

**The
COMMODORE
DISK
and
PRINTER
handbook**

DAVID·BRIDGES

AND

HELEN·NAYLOR

**THE COMMODORE
DISK AND PRINTER HANDBOOK**

**DAVID BRIDGES
and
HELEN NAYLOR**

**Phoenix Publishing Associates Ltd
Bushey, Herts.**

Copyright © David Bridges & Helen Naylor 1984
All rights reserved

First published in Great Britain by
PHOENIX PUBLISHING ASSOCIATES LTD.,
14 Vernon Road, Bushey, Herts. WD2 2JL

ISBN 0 9465 7623 8

Printed in Great Britain by
Billing & Sons Limited, Worcester
Cover design by Ivor Claydon
Graphics by Denis Gibney Graphics, Chesham
Typeset by Prestige Press (UK) Ltd, Chesham

CONTENTS

CHAPTER	PAGE
INTRODUCTION	1
1. INTRODUCING THE DISK DRIVE	3
2. STARTING UP	20
3. THE BASIC COMMANDS	25
4. THE DISK OPERATING SYSTEM	44
5. SEQUENTIAL FILES	73
6. RELATIVE ACCESS FILES	94
7. DIRECT ACCESS FILES	109
8. CHANGING DEVICE NUMBERS	126
9. INTRODUCING PRINTERS	129
10. THE 1526 COMMODORE PRINTER	138
11. USING YOUR PRINTER TO THE FULL	146
12. USING THE EPSON RX-80 PRINTER	176
APPENDIX I ERROR MESSAGES	202
APPENDIX II DISK DRIVE COMMANDS	206
APPENDIX III ABBREVIATIONS	208
INDEX	210

INTRODUCTION

This book is for everyone who wants to know more about the Commodore 1541 disk drive and how to use it in conjunction with a printer. You may be a small business user who has bought your computer, disk drive and printer as a package and are setting out to learn how to use it all from the beginning. You may be a hobbyist who has graduated from the tedious business of cassette storage but is finding using a disk drive very different from handling a cassette recorder. Whatever your motives, you own some powerful, complex, and sophisticated equipment and naturally you want to get the most out of it. This book will show you how.

We have split the book into two parts, one concerning the disk drive and the other covering printers, in particular the Commodore 1526 and the popular family of Epson printers.

The disk drive section starts with a beginner's introduction to the subject. *Clearly* and *simply* it explains how disks work, how to look after them, and what **BASIC** commands you need to know for the everyday routines of data storage. Once you have mastered that, you will be taken at a measured pace through more advanced uses, dealing with the disk operating system and explaining in clear language with plenty of examples how to use each of the commands. We cover in some depth the different types of data storage that are possible with a disk drive and show you how to make the best use of them. There are also some demonstration programs for you to type in which are illustrative and functional, including a particularly useful database program. Finally in this section we round up a collection of useful hints, tips, routines and advice to help you make the most of your disk drive.

The printer section is developed in a similar way to that of the disk drive. It starts with an introduction to computer printers including the technology, connecting up, care and maintenance, in a way designed to be of use to owners of all printers but with particular reference to the CBM 1526. You will learn how to list out programs

very early on so that you will be able to put your printer to practical use almost right away. You will then be led on to more ambitious projects on the 1526 going in detail through the formatting commands and showing how you can exploit your printer's versatility to produce a range of printout formats just as you want them.

For those who have not yet decided which printer to invest in there is a chapter on other printers. The different types such as daisywheel and dot matrix are discussed, including the pros and cons of each and the niceties of which interface you are likely to need. There is also some experienced advice on choosing a printer, and a discussion of the popular Epson range. Finally, we have put together some hints and advice to make using your printer as troublefree as possible.

If the subject is new to you, this book will take you from ignorance to a practical working knowledge and understanding. If you came to it seeking more information, we hope that you will finish by knowing how to make this sophisticated technology really work for you and able to exploit it to its full potential.

1

Introducing the Disk Drive

Using a disk drive with your Commodore 64 is like adding a turbo to your car: suddenly you are driving something that is more powerful, more responsive, and much faster than it ever was before. But like most things of this nature, although you will be impatient to be at the controls straight away, it is worth spending a little time first finding out how it works, how to look after it and how to get the most out of it.

Let's have a look at what you get for your money when you invest in the Commodore 1541 disk drive. First of all, it is a great deal quicker than the alternative method of data storage, the cassette recorder. Programs of average length that may take fifteen or twenty minutes to load from cassette can be loaded from disk in a matter of seconds, or at most just over a minute. Not only is loading faster, but all other data handling operations such as saving files, copying them, deleting them, and so on produce equally impressive savings in time. The hobbyist will have much more time to devote to programming, and the business user could not really seriously consider any other way of accessing his or her information on a regular basis.

The second benefit to come from using a disk drive is storage capacity. People with cassette recorders usually use C12 or C15 cassettes for storing their programs and are limited to around 10k bytes per side. Because the tape gets too thin for reliability on anything longer than a 30 minute tape, really long programs pose all sorts of problems. Not so with a disk drive. On a single disk you have space for 174,000 bytes, or characters, of information and can store up to 144 different programs or data files. Because of the access speed, as well as using disks for larger storage you can also use them to increase the storage capacity of your Commodore beyond its 64K of internal random access memory,

supplementing it by around 170K for each disk you use. For the ambitious, you could link up to five disk drives to your computer at any one time, and then you would have a positively jet-propelled system! Large amounts of data can be stored as a database on disk and used by the computer as a temporary extension of its memory, so that if you want to run a long program that in itself will take up most of your computer's memory you can feed in from disk specific information as data items as and when required to complete the running of the program.

One example of this might be an engineering program for designing gas containers. The computer program would request from its operator the name of the gas to be stored, and it would then select the appropriate one from a database on disk containing details of all the gases and their properties. Working on the parameters supplied, it would then produce a design specification particular to that gas, from a standard program and a standard data file.

The example given above also illustrates the third advantage of a disk drive; flexibility. That engineering application would be totally impractical on a cassette-based system; accessing the information would be variable and extremely slow, you would not have the space to store a good size database, and your only available method of storage, in sequential order, would be clumsy and primitive.

You have the choice of three types of storage on a disk, which we shall be looking at in detail later in the book, and this gives you a great deal of flexibility when it comes to choosing the most efficient type for your particular program.

The final benefit is that by using a disk drive you can carry out interactive processing. That is, large amounts of data can be read from disk into the computer's memory, changed, and written back onto disk quickly and easily as part of the program, such as updating your bank statement, your salary calculations, or your tax records.

FIRST PRINCIPLES

At this point it might be worthwhile to look briefly at the principles of data storage, for those who may have bought their computer and disk drive together and are new to the whole system. Veterans who have been initiated via the rigours of cassette tapes may skip this bit and feel smug.

To put it simply, all the information in the computer's memory, and what you see on the screen, is temporary and only lasts as long as the power is on. As soon as it is switched off, that information is irretrievably lost. In order to keep it for re-use, some form of ancillary storage has to be used, namely magnetic tape or disk. By keying in certain commands, (identified in this book in capitals), you can instruct your computer to **SAVE** the program currently in its memory onto tape or disk in a permanent form, rather as you might tape record a radio program. Then, when you want to use that program again after days or even months, you simply instruct the computer to **LOAD** it back into memory and you can continue using it.

These loading and saving operations involve in effect copying the data from one area to another. If you have just saved a program onto disk, a version still remains in the computer until the power is switched off, so you are still able to change and resave it or otherwise manipulate it if you wish. Similarly, if you have just loaded the program into the computer, it remains on the disk (or cassette) as well, so you do not wipe it clean or in any way affect it by the transfer. Incidentally, if you are dealing with files of data, for example a mailing list, rather than actual programs that do things, we tend to talk of **READING** data from disk into memory and **WRITEing** it from memory onto disk, rather than **LOADing** and **SAVEing**.

The analogy with audio tapes is worth pursuing; in the same way as you would record one piece of music over another if you wanted to, erase a piece you had tired of, or add more tunes to a half-recorded collection, so you would with programs. Disks are even more versatile than this; you are able to do the equivalent of mingling two pieces recorded at different times, changing parts of tunes, and much more. As you progress through the book you will learn how.

SETTING UP

When you get your disk drive home the first thing you want to do is to make sure that everything is there. Here is a quick checklist of what you should have:

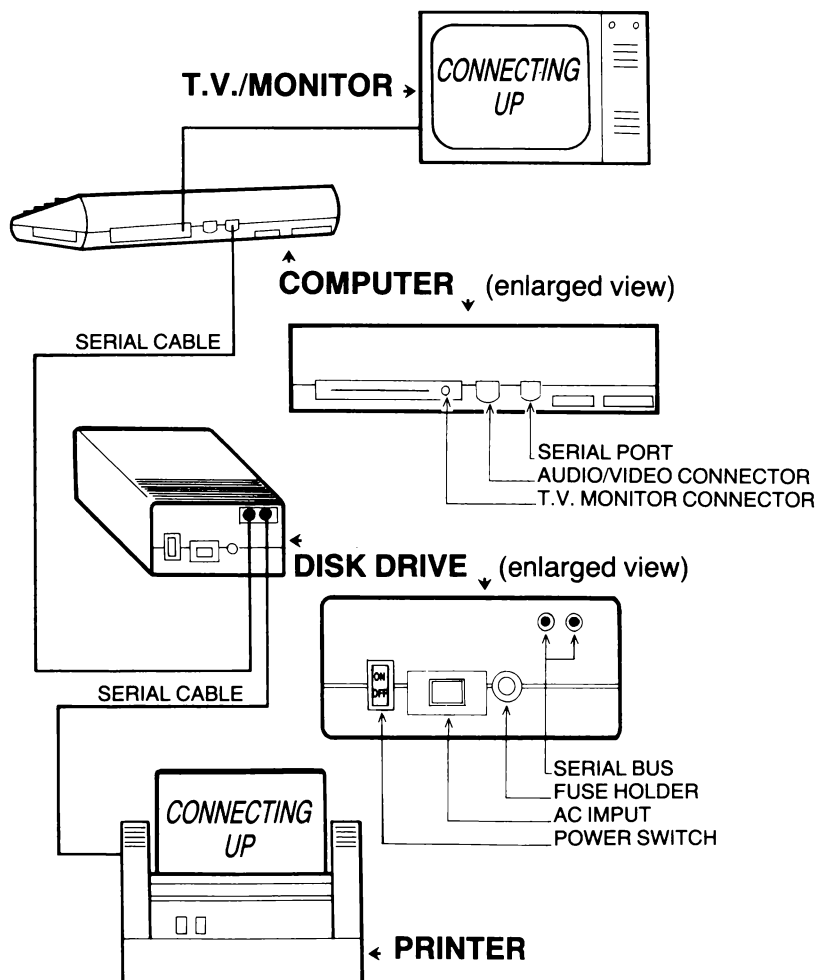
- a disk drive
- a grey lead
- a black lead
- a manual
- a demonstration disk.

The *grey* lead is the mains power supply and is, frustratingly, supplied without a plug. The *black* lead is a serial bus cable for connecting the disk drive to the computer. You will quickly become familiar with the disk drive, inscrutable though it may look on first acquaintance. The main things to remember at first are concerning the two lights on the front panel. The *green* one is a power light and simply indicates that the drive is plugged in and switched on. The *red* light next to it has a dual function: when it is on continuously it signals that the drive is active, that it is **READING** or **WRITEing** data to or from the computer. When the *red* light is flashing it acts as an *error* warning indicating that the drive is *not* functioning properly. On no account should a disk be removed from the drive while the drive-active light is on; you risk damaging the disk and losing all your data. If you get a flashing *red* error light, first check the obvious things like whether it is connected up properly and whether or not, you have forgotten to insert a disk, then if there is an error message on screen consult the Appendix for further advice.

You will find that the drive gets quite warm when it has been on for a while, but that is no cause for alarm, as long as there is a free circulation of air around and particularly over the ventilation grill it won't come to any harm. The other thing that sometimes alarms new users is the mysterious whirring, clunking and grating sounds that often emanates from the disk drive when it is active. Usually these are quite normal, but occasionally they are signals that all is not quite right; you will find that you get to know, like the shakes and rattles on your car, which noises are okay and which ones to worry about.

CONNECTING UP

From the tangle of wires before you, you have to create a tangle in the right order between the right bits, so connect up in the following way but **DO NOT SWITCH ANYTHING ON YET**.



1. Take the *grey* power cable and insert it into the back of the disk drive, the right way up, next to the ON/OFF switch.
 2. Plug the other end of the cable into the power socket but do not yet switch the drive on.
 3. Take the *black* serial bus cable and plug one end into either one of the serial bus sockets at the back of the drive. (The cable is reversible).
 4. Making sure that the computer is not switched on, plug the other end of the serial bus cable into the serial port on the back of the Commodore 64 (the round DIN socket to the left as you sit at the keyboard).
 5. Check that all the other components are correctly connected: television to computer, television to mains, and computer to mains **via power pack**.
-

POWERING UP

It is important that everything is switched on in the right order. If you don't follow the correct sequence you run the risk of losing data or even damaging the drive. So . . .

1. Check that you have not left a disk in the drive.
2. Switch on at the mains.
3. Switch on the disk drive. It will whirr and the *red* light flash briefly as its motor starts up. It will then go silent and the *green* light will come on.
4. Switch on the television.
5. Switch on the computer. Again the drive will flash and whirr a little before going quiet.

You should now have the familiar Commodore welcome message on your screen, and you are ready to go.

POWERING DOWN

Before you do go, however, we had better tell you how to switch off properly when the time comes because it is just as important and for the same reasons: to avoid losing data or damaging your equipment.

1. Check that there is no disk in the drive.
2. Switch off the computer.
3. Switch off the television and disk drive.
4. Switch off at the mains.

THE WORKING DISK DRIVE

By now you are probably curious to know exactly how the disk drive works. Inside the box there are two principle areas of operation: the back half contains a printed circuit board and the chips which enable it to operate electronically, the front half houses the motor, the read-write head and all the mechanical bits.

The electronics on the pcb are really rather sophisticated; equally as complex as the Commodore Vic 20 computer, in fact. It contains chips totalling 16K of ROM which house the DOS (Disk Operating System) software. The DOS works in a similar way to the operating system in the computer, it is the software framework through which the commands that you input are activated, and it controls all the disk functions such as reading and writing, formatting, erasing, and so on. The other chips on the printed circuit board constitute 2K of RAM. This acts as a buffer area into which data is read from the computer before being stored on disk. With some computers the DOS has to be **LOADED** into the computer itself (for example, the BBC Micro) and therefore uses up some of the computer memory for disk processing.

The circuitry within the Commodore disk drive makes it an intelligent device which can carry out all its own processing without taking any memory away from the computer, and thus greatly improves the efficiency of the system. This is especially true when used with the standard VIC-20 with its limited 3.5K of user RAM!

When you insert a disk it slides into position between two horizontal grooves, rather as a drawer does, and as you draw down the shutter you also draw down a clamp to grip the centre of the disk. When the motor starts it spins the disk like a record on a record player. If you look at a disk you will see an oval shaped hole in the outer casing exposing the disk underneath. This exposed area is where the data transfer occurs. There is a read/write head projecting up from the floor of the drive and when it is instructed to

transfer data to or from the disk it comes into contact with the magnetic surface at this point, searching for the correct part while the disk revolves over it. Some other types of disk drive have the head positioned over the disk like a record player stylus, but the Commodore drives work the reverse way using the disk underside as we have just described.

If you don't mind risking invalidating your guarantee it is interesting to remove the top cover of the disk drive and watch its mechanics in operation. You gain a deeper insight into the clankings and whirrings that way!

Now that we are acquainted with the disk drive, let's have a look at the disks themselves. A disk is essentially a circular piece of plastic coated with a magnetic surface. It is worth making the point now that this surface is extremely sensitive, and one of the fundamental principles of disk use is to remember that, like any other magnetic recording surface, it must be treated with great care.

DISK CARE

There are a few rules which it is only common sense to observe for trouble free computing. If you don't follow them, it is quite possible that your disk will be harmed and your data corrupted. The saying goes that it happens to everyone – once. To lose a disk full of precious programs and data causes such heartache that no-one who has learnt the hard way is likely to need telling twice, but if you absorb these do's and don'ts, hopefully you will be spared the anguish!

DO handle the disk gently. Rough handling or forcing may damage it.

DO keep it in its protective paper cover whenever it is not in the drive, and in a plastic library case whenever you are not computing.

DO keep it out of the sunlight and extremes of temperature. Strong ultraviolet light and temperatures outside a range of 10°C to 52°C can cause damage.

DO make sure that you always remove any disk from the drive before switching it on or off.

DO protect it from dust, cigarette ash, coffee, and all the normal hazards of a working environment!

DO hold the disk only by its protective casing.

DO NOT touch the exposed surface with your fingers, you will leave a deposit of grease behind.

DO NOT write on it with a ball point pen; the hard tip will mark it. Use a felt tip pen or, preferably, write the label before fixing it to the disk.

DO NOT leave it anywhere near magnetic fields that could corrupt the data. This includes the television, a ringing telephone, the top of the disk drive, the kettle, the family hairdriers, and curling tongs . . .

DO NOT attempt to remove or insert a disk while the red drive-active light is on.

DO NOT attach anything to the disk with paper clips or staples.

DO NOT attempt to clean the disk. Abrasion will cause the loss of data.

You will probably have noticed that each disk bears a number of notches and holes through its protective covering. The large central hole accommodates the spindle in the disk drive around which the disk revolves. The oval shaped hole, as we have mentioned previously, exposes the area over which the read/write head is active and where the data transfer takes place. The small hole near the centre is required by some types of disk drive to help monitor the speed of revolution, but it is redundant in the case of the Commodore system which uses software for that purpose (known as soft-sectoring).

Finally, the notch on the left-hand edge is known as the write-protect notch. This is fairly self-explanatory: it acts in the same way as the two lugs on the rear edge of a cassette tape, in that if the notch is covered over by a label or some such, it ensures that although data can be read from the disk it cannot be written to it.

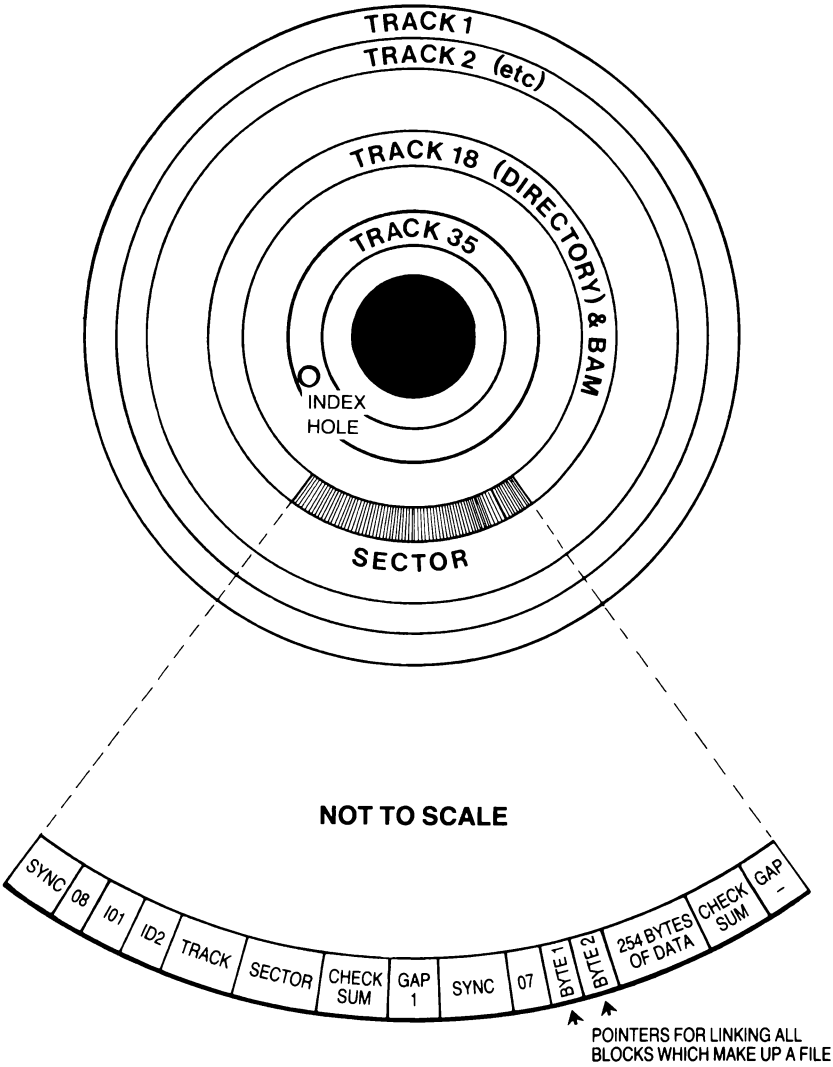
What happens is that once the disk is in the drive, if you instruct the drive to write data onto the disk it first takes the precaution of shining a vertical beam of light onto that spot to detect whether the notch is covered or not. If the light is blocked you will get a flashing red error light from your disk drive and only if the beam of light is detected on the other side of the disk will the drive go ahead with writing data. Commercial disks always have this notch covered over so that you do not inadvertently erase the program by overwriting it, and you can protect your own disks in the same way.

DISK STRUCTURE

Different computers use disks structured in different ways, so when you use a blank disk for the first time you have to format it for use. We will show you how to do this in the next chapter but for now all you need to know is that this process produces a disk with a surface magnetically divided into 35 concentric circles, known as tracks. The tracks are numbered from 1 on the outside to 35 on the inside. One track is reserved for disk administration, and as it is read by the drive before every transaction it is located in the most economic position: the central track, 18.

The disk is also marked out into radiating divisions like the slices of a cake, known as sectors, and into smaller areas bounded by the tracks and sectors known as blocks. There are 21 blocks on the outside track, and as the tracks decrease in size coming inwards the number of blocks falls to 17 on the inside, with a total of 683 blocks on the whole disk.

It is in these blocks that the data is stored and when the drive writes to disk it allocates information to blocks all over the disk, which, when they are all read back together form the complete program or data file. Each block has a maximum capacity of 256 bytes or characters of information. The disk drive works by positioning the read/write head over the appropriate track as the disk revolves, so when the correct block passes the data is read by the head and passed into the computer.



THE BAM

The Block Availability Map, or **BAM**, is the means by which the disk drive organises the position of data in blocks.

We have already said that each block can hold up to 256 characters, so when one block is full the drive has to find another vacant block to continue into. The drive does not always work in a logically sequential way and write data into a series of blocks all next to each other on the same track. Different types of file arrangement can result in data being scattered all over the disk. Also, the drive operates in a most economical way by writing data first to the areas nearest to the central track so that the read/write head travels the minimum distance and access time is as fast as possible. Therefore, if there is an odd empty block on one track the drive will fill that first before going off to another further area of disk, rather than looking for a single large area to take the data all together.

This rather random method of organisation requires a record to be maintained of all the blocks on the disk and their availability, and this is where the **BAM** comes in. The **BAM**, on track 18, sector 0, is in effect a map of all blocks and their status: empty or full. Whenever the drive is instructed to write data to disk, the read/write head first consults the **BAM** for the locations of available blocks so that already full blocks are not overwritten and so that it can find the nearest available area.

After the data has been recorded, the drive returns to the **BAM** and updates it, so that it is always an accurate reflection of the current situation.

THE DIRECTORY

Also situated on track 18, right next to the **BAM**, is the directory. This acts as an inventory of all the files on disk. If you have forgotten which files you have stored on which disk, you can instruct the computer to list out the directory and it will give you all the file names and types for that disk. There is room for 144 entries on the directory, which is the limit to the number of files of any length that can be stored on a disk.

The directory also stores for each file name the location of the first data block, by track and sector, and the number of blocks in the complete file. When the drive is instructed to read from a file it first consults the directory to check that a file of that name is present, and also to find out where it starts. It can then go directly to the appropriate track and sector. If you are using the wrong disk the flashing red error light will tell you that the drive cannot find that file.

You can **LOAD** the directory into memory by using:—

LOAD“\$,8

If you then enter:—

LIST

the names of all the files on the disk are displayed on the screen.

Blocks 1 to 18 of the 18th track on a disk are reserved for the directory. Each block can hold entries for eight files giving a maximum capacity of 144 files per disk, although in practice large files may fill up all the disk storage capacity before this number is reached.

Each file entry is made up of 30 bytes of data followed by 2 spare bytes. The structure of each entry is:—

BYTE	USE
0	FILE TYPE (see below)
1,2	TRACK & SECTOR OF FIRST DATA BLOCK
3-18	FILE NAME (padded with SHIFTed spaces)
19,20	TRACK & SECTOR OF FIRST SIDE SECTOR (REL. FILE ONLY).
21	RECORD SIZE (REL. FILE ONLY)
22-25	UNUSED
26,27	TRACK & SECTOR OF REPLACEMENT FILE WHEN USING OPEN .
28,29	NUMBER OF BLOCKS IN FILE: LOW BYTE, HIGH BYTE.
30,31	UNUSED

The first byte within each entry is used to store the file-type:-

0 = DELETED

1 = SEQUENTIAL

2 = PROGRAM

3 = USER

4 = RELATIVE

When a particular type of file is first **OPENED** the relevant number is stored and when it is **CLOSED** properly 128 is added (or more precisely, **ORED**) to indicate this. Bytes 1 and 2 are used to store the track and sector location of the first block of data in the file. When your file is **OPENED** to **READ**, the **DOS** will first of all read these two bytes before moving the head to the first block so as to **READ** the data within it. Each block contains 256 bytes, of which the first two act as a forward pointer to the next block. The remaining 254 bytes are used for the storage of data. The individual blocks within a file are therefore 'Chained' together by these pointers, each one storing the address of the next block in the file.

Of the remaining space in the directory entry, bytes 3 to 18 contain the file name and **SHIFTED** spaces are used if necessary to pad it out to the full 16 bytes. Bytes 19 – 21 relate specifically to relative files, 22 – 25 are unused, and 26 – 27 have a particular application in connection with the **WRITE** and **REPLACE** command. Finally, bytes 28 and 29 store the total number of blocks within the file, and this is used by the **DOS** as a check that the file is not corrupt. If this number does not correspond with the number of blocks read in an error message will be generated. Within each individual block the data is laid as a series of characters, (or more precisely the **ASCII** code for those characters), with 254 bytes per block storing data and the first two bytes storing the pointer.

DENSITY

Density refers to the amount of data that can be stored on a disk, that is to say, how closely packed the tracks and sectors are. It is possible to buy disks that are single, double, or even quadruple density, depending on the disk drive. The Commodore 1541

requires single density disks. In theory, the greater the density, the greater the storage capacity of a disk, but this is not always true because some disk drives organise their data more efficiently than others.

The other element in the capacity of disks is whether they are single or double-sided. Here there is a direct correlation; double sided disks work literally on both sides and therefore have double the capacity of single-sided ones. Officially, your Commodore 1541 uses only single-sided disks, but some people have found that by reversing the disk in the drive they can also use the second side, having first cut a new write-protect notch on the opposite side of the disk. The only snag with this is reliability. All disks are manufactured in the same way, and the quality of each side is tested thoroughly. Those that have been found to have one side of a higher quality than the other are marketed as single-sided disks. So be warned, if you are tempted to economise by using the reverse side of a disk on your disk drive, do make sure that you are not storing your life's work or something equally irreplaceable on it!

Other terms that specify the capacity of a disk are the size in inches and the number of tracks. The size always quoted is the diameter of the disk, ranging from 8" down to 3". For the Commodore system you need the 5¼" size, and, disks that can be formatted to a density of 35 tracks. To sum up then, the disk to buy for the Commodore 1541 disk drive is single-sided, single density and 5¼ inches.

BACK-UP COPIES

Horror stories from computer buffs abound about their disk drives "crashing" and hours of patient work being wiped from the system in a second. If you have used a cassette based system in the past you have probably had occasions when a tape mysteriously became corrupted and would not **LOAD**. The same is not uncommon on disks and considering their sensitivity to the environment it is often understandable. It is not much consolation, however, when you have just knocked a mug of steaming coffee all over the disk containing your business accounts for the past year to reflect that you kept meaning to make a back-up copy but never got around to it!

Disaster can strike in three different ways: you can lose the information in your computer before you have stored it on disk, your disk drive can malfunction and refuse to retrieve data from a disk, or the disk itself can become corrupted and the information on it become useless.

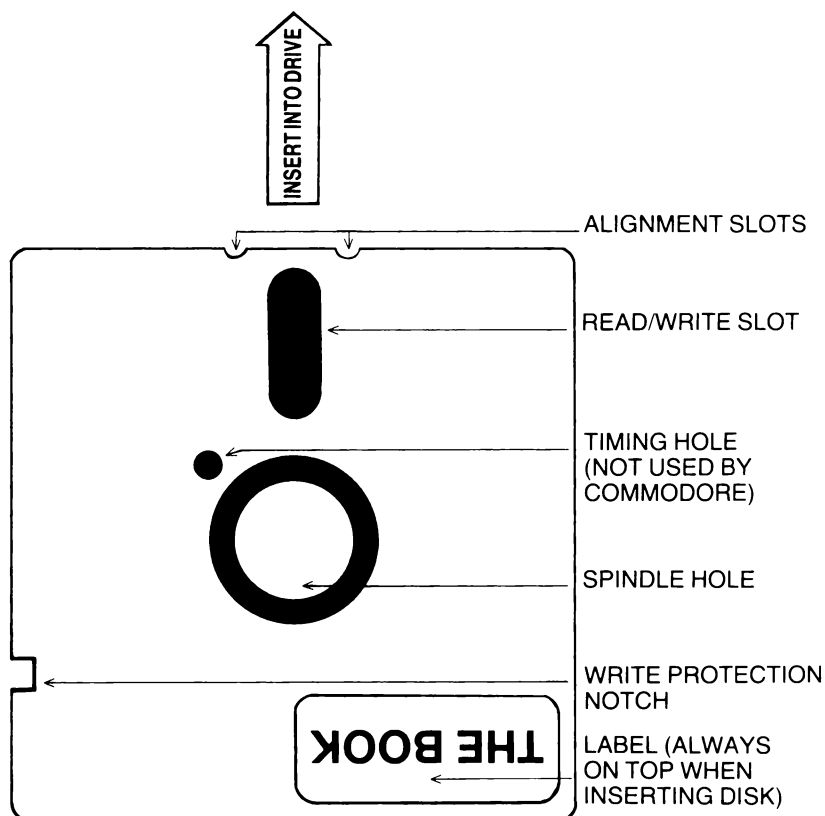
To avoid the first calamity it is best to store all the work that you have been doing on the computer onto a disk at convenient stages and at regular intervals, say every hour. That way, if there is a power cut or you accidentally clear the memory of the computer, you have not lost more than an hour's work.

To minimise the ravages that can be wrought by a malfunctioning disk drive or a corrupt disk, you should always keep a second, back-up copy of every file on a separate disk. If you think there is something wrong with the drive, it is best to get it looked at before you put your second copy in and risk getting that chewed up too. If the disk itself has got damaged, you've no problem if there is a back-up around somewhere. Another advantage of doing this is that if, for some reason, the first copy has not saved properly, you have a second chance to get it right.

Tedious though it may be, you will not regret getting into the habit of making regular copies of your work, and if you are a serious user, particularly a business user, it really is essential.

INSERTING A DISK

The correct way to insert a disk into the drive is with the label uppermost and the write/protect notch to the left. Holding it by its protective casing push it horizontally into the drive until you feel it click into place. Then pull down the drive door (which also brings down the central spindle) and you are ready to go.



To remove a disk, simply push the door latch inwards and upwards and the disk will be released and spring out. Do remember not to insert or remove disks if the red drive active light is on, (not the green light, as some editions of the Commodore manual state).

So here you are, all connected up, disk in drive; now read on to the next chapter and find out how to do something with it.

2

Starting Up

Well, now that you understand the basic principles of how your disk drive works, no doubt you are becoming impatient to actually use it. Most people cut their teeth on commercial software, so we will now show you how to use this before going on, in the following chapters, to how to handle your own programs. Even if you do not use shop-bought programs you are advised to read through this next section because in it we will be introducing and explaining some of the specialist terms used in relation to disk storage.

COMMERCIAL SOFTWARE

There is an excellent selection of commercial software available for all of the range of Commodore computers. It is usually very reasonably priced, and although disk-based software is a bit more expensive than cassette-based, it is well worth the extra outlay for a format that is so very much faster and easier to use. Commercial games can take up to 15 minutes to **LOAD** from tape, while the same version can be accessed from disk in under two minutes. Quite a saving!

Business users will no doubt find that they are compelled to have a disk drive, because very little specialist software is available on cassette. Commercial applications, like stock control or sales ledgers, are only practicable with the sophisticated file handling capabilities of a disk drive, so the question is not whether to buy a disk drive so much as how many to buy. For now we are assuming that you have just one, but later we will show you how to link up several, if you feel that you are going to be needing a larger system.

Loading commercial software is quite easy. In many cases the instructions are on the disk sleeve, or the accompanying instructions sheet, so it is simply a case of inserting the disk carefully into the drive with the label uppermost, drawing down the door, and typing in the magic string of characters supplied by the manufacturers. In your anticipation don't forget to press **<RETURN>** afterwards (of course!), because nothing will happen until you do! The drive will then start to whirr and the drive-active light will come on as your software is **LOAD**ed into the computer. When peace and quiet have been restored you are ready to start.

If you should find that your commercial disk has been supplied without **LOAD**ing instructions, all you need to do, once you have inserted it into the drive, is to type the command **LOAD**, followed by the program name within quotation marks, a comma and a figure eight. Like this:—

LOAD"PROGNAME",8<RETURN>

or, if your program was called TAXCALC for example, then like this:—

LOAD"TAXCALC",8<RETURN>

If for some reason this does not work, check what you have entered very carefully. It is easy enough to omit a comma or a quotation mark and a detail that small will cause a malfunction. The SYNTAX of commands, as this is called, is very important. Every time we introduce you to a new command we will give its syntax, as we have done with **LOAD** above, and although the individual elements will vary according to your needs, it is essential that you follow the given format exactly whenever you use them.

Let's look at how this command is built up. The quotation marks are necessary because they are the means by which the computer recognises the characters within them as a name. When you get to using file names you will find that they also have to be used in the same way.

The comma and eight together form the DEVICE NUMBER, that is, they signal to the computer that the peripheral with which it is communicating is a disk drive. Any device which communicates with the computer for input and/or output, such as a disk drive or a printer, has a number by which the computer identifies it. Commodore disk drives are manufactured having the device

number eight, so whenever you want to instruct the computer to output information to a disk you have to include ,8 in the command for it to be understood. There is a certain amount of flexibility in this system to accommodate set-ups where there are more than one of the same type of device. If you were using two single disk drives, for example, you would be able to change the device number of one of them to allow the computer to distinguish between them. For details of how to do this, see Chapter 8. Whichever device number you use it must be within the range 0–31 and is determined by the particular piece of equipment that is connected to the Computer. 0–11 have already been allocated by Commodore as shown in the table below and 12–31 have been left free to accommodate the expansion of your system.

Number	Device	Direction
0	Keyboard	Input
1	Cassette recorder	Input/Output
2	RS232 interface	Input/Output
3	Video screen	Input/Output
4	Printer	Output
5	2nd printer	Output
6	Plotter	Output
7	2nd plotter	Output
8	Disk drive	Input/Output
9	2nd disk drive	Input/Output
10	3rd disk drive	Input/Output
11	4th disk drive	Input/Output

When we go on to discuss the **BASIC** commands, in the next chapter, we will use dn in the syntax as an abbreviation for device number, so whenever you come across it remember that under most circumstances you will be using ,8. Incidentally, for historical reasons your Commodore computer automatically defaults to one in the absence of a specified device number, so if you forget to include it the computer will assume that it is dealing with a cassette recorder.

There is an optional, final part to the **LOAD** command which you may come across, especially when using commercial software. In some circumstances sophisticated machine code programs have to be **LOADed** into a particular area of the computer's memory, and to achieve this you have to type ,1 at the very end. On Commodore equipment this is known as a 'secondary address', abbreviated here to sa. Do not worry about this, whenever you need to use it the instructions are sure to say so, but we mention it here so that if you do come across it you will understand why it is necessary.

So, bit by bit we have pieced together the **LOAD** command. Its full syntax, as we now know, is:—

LOAD“PROGNAME”,dn,sa

and all you normally need to do is put the name of your program within the quotes, give 8 as the device number, and add ,1 at the end if required.

PREPARING A NEW DISK (FORMATTING)

It would be very nice if you could buy a new box of disks, pick one out, slot it into the disk drive and start storing programs on it straight away, but unfortunately life is not that simple. Virgin disks have a blank magnetic surface that cannot be used for data storage until they have been **FORMATTED**. The reason why you can buy the same disks for use on quite different makes of computer is that this process has to be done by you on your own computer to prepare disks that have the correct magnetic structure for storing your data. Formatting is the process by which the disk drive acts on a new disk and marks it out into the tracks, sectors and blocks that we have already talked about. It also marks out the **BAM** on track 18, indicating that all blocks are free, and it creates the directory, where the disk name is stored and a list of all future files will be held. So, before you even begin using your disk drive for other than commercial software you must first take a fresh disk and format it ready for use.

There are two commands involved in this process: **OPEN** and **NEW**. **OPEN** has the effect of **OPENing** up a communication channel between the computer and the disk drive through which other commands can be sent, and **NEW** effects the physical

formatting of the disk. The syntax is:—

OPEN (filenumber),(dn),(sa),“NEW:diskname,id”

and as an example, to format a disk called Helen 29 you would be:—

OPEN 1,8,15,“NEW:HELEN,29”<RETURN>

It looks quite daunting, doesn't it? It's not as complicated as it may seem, however. Remembr that the principle objective is to name and prepare a disk, so first you must think of a name of less than 17 characters plus two extra characters which will act as an id, or **IDENTIFIER**. Most people use a string of letters as the name followed by an identifier of two numbers because this is easy to remember, but in fact you can use any combination of **PRINTable** characters.

The identifier code functions in everyday file handling as an abbreviation for the file name. When the disk is formatted, it's name and identifier are written into the directory, and the identifier on its own is also written into the header of every block on the disk. So thereafter, every time a block is read from the disk into the computer, the computer will be able to identify it as having come from that particular disk. The advantage of this is that if you are using several disks and inadvertently put the wrong one into the drive, the computer will detect it and warn you with an error message. For this reason it is important that you give every disk not only a unique name, but also a unique id.

The **OPEN** command is actually quite complex and has many more applications than the example given here, but now is not the time to look at it in depth. From the viewpoint of formatting a disk, it is better at this stage simply to follow the example given above, remembering all that we have said about the choice of disk name and identifiers. The disk drive may whirr and clank alarmingly and the red light will flash for a short while, and after 90 seconds or so you will be ready to go, with a freshly formatted disk.

Once you have **OPENed** a channel, for formatting or any other purpose, you must complete the process by **CLOSEing** it afterwards, otherwise you will get all sorts of problems. So after you have **NEWed** your disk the final step is to type in:—

CLOSE 1<RETURN>

3

The Basic Commands

In the last chapter you saw how easy it was to use the varied range of disk-based commercial software that is available for the Commodore computers with an attached disk drive unit. Perhaps this is the only use that you have for your disk drive. After all, it is easier to buy a word-processor package (especially the excellent EASYSCRIPT) than to try writing your own word processing program; games programs **LOAD** very much faster when they are stored on a disk, and you have no doubt read of, and seen the myriad advantages of using the 1541 disk unit with commercial software.

Curiosity killed the cat, so it is said. That was in the days before the electronic revolution and curiosity will have no detrimental effect upon either your computer or its disk drive, although it can quite easily give you many a late night! So, let's ask a few questions. How does a disk store its information? How can you write programs that make the best use of the speed and storage capacity of a disk? The questions are endless, aren't they? Although you may have bought your disk drive solely to use shop-bought software, we are really talking about what could become a fascinating hobby.

In the next few chapters we shall be looking in greater detail at the workings of the 1541 disk unit. From now onwards it is assumed that you have a working knowledge of the **BASIC** language. Although sophisticated, the 1541 is not complicated, and everything will be explained clearly and comprehensively. This is not, however, a beginner's guide to **BASIC**, for that you must look elsewhere.

In this chapter we shall be looking at the **BASIC** Input/Output (I/O) commands that are built into the CBM 64 and VIC-20 computers.

Through the book we deal with each command in a similar way to enable you to understand them quickly and easily. For each command we begin with a summary of its function and show its syntax in a standard format. Then we give a few examples, and finally we explain how it works and go into some detail about how and when to use it. Don't worry if some terms and concepts are new to you, everything will be explained as we go along.

BASIC INPUT/OUTPUT COMMANDS

All the keywords in this section can be described as general **Input/Output** commands. The term **Input/Output**, or **I/O**, is used to describe any transmission of data to or from equipment other than the computer itself. In this case we are looking particularly at the disk drive, but other devices include the cassette player, printer, RS-232 port, modem and so on. Each *peripheral*, as this type of equipment is called, may have its own group of commands to deal with its particular requirements. There is, however, a set of common keywords within the **BASIC** language that can be used with and I/O device and these are described here, with particular reference to their use with a disk unit, but also mentioning other applications that you are likely to come across. In some cases we discuss their use on cassette-based systems first, because the simplicity of cassette recorder operation leads easily on to an understanding of the more complex disk-based system.

SAVE

FUNCTION: To record (**SAVE**) the **BASIC** program that is currently in the memory of the computer onto a storage device, (cassette or disk). The **SAVE** command can be used either directly or within a program. Only the program itself is **SAVED**, any variables in memory are automatically **CLeaRed** by the **SAVE** process.

SYNTAX: **SAVE "PROGRAM NAME",DN [,SA]**

ABBREVIATION: **S<SHIFT>A**

EXAMPLES: **SAVE**

SAVE "TAXCALC"

SAVE "TAXCALC",8

SAVE "TAXCALC",8,1

SAVE "0:TAXCALC",8

We will use TAXCALC as an example program that you might be writing to work out your tax payments. The command **SAVE** is used to record a copy of the program currently in memory onto a disk or cassette for later use. The program is unchanged by using **SAVE** and by storing it thus you can then re**LOAD** it at any later date. Otherwise your program will be lost forever when the computer is turned off! The first example given above, **SAVE** by itself, will **SAVE** the program onto the cassette without giving it a name. This only works using a cassette recorder. If you were **SAVEing** several programs you would have to distinguish between them by using a separate cassette for each one, then the computer would simply re**LOAD** the first program that it came to on each tape.

The second example is very similar to the first. Again, the computer will **SAVE** the program onto a cassette but this time the program name ("TAXCALC") will be recorded at the beginning of the program for identification purposes. This can be particularly useful when more than one program is **SAVED** on the same cassette because the named program can then be selectively re**LOADed** by skipping over the other programs and searching for that name. When you use the cassette recorder for storage, naming your program is optional. A cassette is of variable length; it can be a C12, C60, C90 and so on and the recorder has no way of knowing where on the tape your program is. It just starts at the beginning and works its way sequentially through to the end, finding each program as it comes to it. The disk drive, being far more sophisticated, is able to jump to any part of the disk to find a program quickly. It is vital, therefore, that the program **MUST** be given a unique identifying name. In fact, the disk unit will not allow you to **SAVE** a program if one with the same name is already on the disk. When this is the case, the red light will flash, to indicate

an error condition, and the **SAVE** will be aborted, so choose your names carefully.

The third example, **SAVE "TAXCALC",8** is the first that relates specifically to the disk. As you can see, the syntax is very similar to the previous examples, the difference being the addition of the (,8), which is normally the device number of the disk drive. When you buy your disk drive it is set at the factory to be device number 8 (although it is possible, as you will see later, to alter this). To **SAVE** a program to any device you simply have to specify its device number so that if you had four disk units connected up and numbered **8, 9, 10** and **11**, then you could **SAVE** a program onto any one of them simply by specifying the relevant number.

Up until very recently the cassette recorder was the main method of magnetic storage for home computers. To make life easier for you, the computer assumes that you want to use a cassette recorder if you do not specify any other device number. This is why you can use the (abbreviated) form '**SAVE**' only when **SAVEing** programs to cassette. There is little doubt that the next generation of home computers will default to a disk drive rather than a cassette but for now you have to specify your disk drive number whenever you want to use it.

The final example shown above is:

SAVE "TAXCALC",8,1

and this is a specialized form of the normal '**SAVE** to disk' command. The final '**,1**' actually tells the computer that you want the program to be re**LOADed**, when the time comes, into exactly the same area of memory where it was originally stored. Under normal circumstances a program will be **LOADed** into memory starting at a particular point known as the 'Start of BASIC'

Some programs however, need to be **LOADed** into memory at a different point other than the usual starting position. This is neither the time nor the place to go into the whys and wherefores. It is a sophisticated technique and if you know how to use it then you will already know what the secondary address of **1** means. If you don't know then don't worry. By the time you have learnt assembly language programming you will fully appreciate the benefits of **SAVE "PROGRAM",8,1**.

The final example is also slightly different from usual for a particular reason. Commodore manufacture several different types of disk drive and for compatibility reasons they all use the same command structure. Some units contain a single disk drive, (the 1541 for instance), whilst others (the CBM 4040 and CBM 80 80) are dual disk drives within a single box. With the latter the drives are numbered 0 and 1 within each device number and each drive is accessed by specifying its number. For example **SAVE "1:TAXCALC",8** will save the program on drive number 1 of device number 8. With single disk units the drive is always number 0, so this specification is optional. However, Commodore programs and disks are highly interchangeable throughout the product range, so if you think it likely that your program may be used on a twin drive commercial system then you are recommended to use the format that specifies the drive number. Note that two single drive 1541 units are not the same thing as a dual drive unit. In the first case you will have devices 8 and 9 whilst in the second case you will have device 8, drives 0 and 1.

SAVE AND REPLACE

Although we have just said that you cannot **SAVE** a program if one with the same name already exists on disk, there is one exception to this. There is a modified version of the **SAVE** command that you can use which **SAVES** a new program under an existing name and replaces the old program at the same time. The syntax for this is:—

SAVE "C0:TAXCALC1",8

or

SAVE "C:TAXCALC",8

The 0 can be optional with a single disk drive. Using this form of **SAVE** can be very convenient, especially if you are developing a program and you want **SAVE** versions of it regularly to guard against crashes.

There are two points to bear in mind when using the **SAVE** and **REPLACE** option. Firstly, early models of the 1541 disk drive contained a bug in this routine that occasionally and unpredictably corrupted the disk, losing not only the **SAVEd** program but often several other files on the disk as well. Commodore reassuringly

state that this bug has now been corrected and the routine works perfectly, but be warned. The author once lost all the records for his annual business accounts by **SAVE** and **REPLACE**ing a program in that way! (Of course, there should have been a back-up disk, but)

A second point to watch out for if you do use the **SAVE** and **REPLACE** option is to make sure that you don't accidentally erase a program that you want to keep which you had forgotten just happens to have the same name. The program name can be upto 16 characters long so you have plenty of choice when naming files.

VERIFY

FUNCTION: To compare the program in the memory of the computer with the program file on the specified storage device to ensure that both are identical. When using a disk drive, the purpose of **VERIFY** is to make sure that your program has been **SAVE**d properly, so you should always **VERIFY** immediately after **SAVE**ing and before any alterations are made to the program in memory.

SYNTAX: **VERIFY "PROGRAM NAME",DN**

ABBREVIATION: **V<SHIFT>E**

EXAMPLES: **VERIFY**

VERIFY "TAXCALC"

VERIFY "TAXCALC",8

VERIFY A\$,8

VERIFY "*",8

The first two examples are used to **VERIFY** a program that has been **SAVE**d onto a cassette recorder. In the first case, the computer checks the first program that it comes to on a tape, whilst in the second case it searches for a program called "TAXCALC" before starting to **VERIFY**. As with the **SAVE** command, it is not necessary to specify a device number when you are using the cassette recorder.

By specifying device number **8**, as in the third example, you instruct the computer to operate with the disk drive rather than the cassette recorder.

It is possible to use a string variable instead of your program name, as in the fourth example. You could, for instance, use:—

A\$="PROGRAM1"

VERIFY A\$,8

The final example displays the use of the pattern matching character **"**"**. If you use the form:—

VERIFY "",8**

immediately after **SAVEing** your program then it will be **VERIFYed**. In this case the **"**"** tells the disk-drive to **VERIFY** the last program that was **SAVEed**, whatever it's name.

When you enter a **VERIFY** instruction, if a program file with the given name is found it can be checked against that program in memory, but if not, a 'FILE NOT FOUND ERROR' message will be displayed on the screen. If the program has been found and the **VERIFY** is satisfactory, then you can be sure that your program has been correctly **SAVEed** onto the disk. A '**VERIFY ERROR**' message indicates that for some reason the two programs are not identical. The most common reason for this message is you! If you add, delete or in anyway alter a single character in your program between **SAVEing** it and attempting to **VERIFY** it then you will not be able to **VERIFY** properly because the two versions will no longer be the same. If you are using pattern matching and any other disk operation is carried out between **SAVEing** and using this form of **VERIFY**, then the result will be almost certainly be a '**VERIFY ERROR**' because the disk drive will only keep track of the last operation that it has performed. Assuming that you are actually trying to **VERIFY** the correct program using the proper name, then the only other reason for a '**VERIFY ERROR**' message is that the program has not been **SAVEed** correctly. This can occur from time to time for a number of reasons, such as a momentary loss of power, or a bad disk. If this should happen to you then remember that your program has not yet been **SAVEed**, so the first thing you should do is try to **SAVE** it again using a different name and preferable a different disk. Only when you have correctly **VERIFYed** your program can you feel secure in the knowledge that it is safely stored on disk.

LOAD

FUNCTION: To transfer a copy of a program from a storage device into the memory of the computer. Unlike the **SAVE** command, which only **SAVEs BASIC** programs, **LOAD** can be used to load machine code programs, or any other area of memory.

SYNTAX: **LOAD "PROGRAM NAME",DN[,SA]**

ABBREVIATION: **L<SHIFT>O**

EXAMPLES: **LOAD**

LOAD "TAXCALC"

LOAD "TAXCALC",8

LOAD "TAXCALC",8,1

LOAD "TAXC*",8

LOAD "",8**

As with the **SAVE** and **VERIFY** commands, the first two examples will **LOAD** a program from a cassette recorder. The first example **LOADs** the first program that occurs on the tape, while the second will search for a program called "TAXCALC" and **LOAD** it.

The third example adds the device number **8** to the command and so instructs the computer to **LOAD** a program from the disk drive. Whilst a program name is optional when using the cassette recorder, a name *MUST* be specified when using the disk storage system. You can, of course, use a string variable for the program name, just as with the commands **SAVE** and **VERIFY** and you can also use the pattern matching and 'wild card' characters '*' and '?', as in examples five and six. Example five demonstrates a **LOADing** short cut. You simply abbreviate your program name by entering just enough characters to identify it from any other files on disk, followed by an asterisk, and the computer will **LOAD** it. Example six is a similar short cut, but it only works if you want to access the first, or the only program on the disk, because it simply **LOADs** the first program on the directory. This is the simplest method of **LOADing**, and is often used with commercial software because there is usually only one program on a disk.

LOAD at first appears to be a very simple command; it **LOADs** a program into the computer's memory. However, there are several

undocumented attributes of this innocent looking keyword that can be teamed up with the speed and flexibility of the 1541 disk drive to enable you to produce very professional programs, if you are willing to put in the necessary programming skills and time.

The secrets of **LOAD** are threefold. The first is really a facet of the disk drive rather than the **LOAD** command itself: the power of pattern matching. We have already seen a couple of examples. Another, particularly useful instance is when you want to **LOAD** the directory to see what is stored on disk. To **LOAD** the complete directory into memory the instruction is:—

LOAD "\$",8

However it is not always convenient or necessary to **LOAD** the whole directory, so pattern matching can be used to identify particular files. The instruction:—

LOAD "\$GAME*",8

makes the computer search through the directory and **LOAD** into memory the names of all files beginning "GAME . . .". As a single disk can store up to 144 files (subject to total capacity) this can be a good way of manipulating the computer to pick out a mini-directory of those files you are particularly interested in, ignoring the rest. To read the directory you must, of course, then **LIST** it onto the screen.

The second of the little known features of the **LOAD** command is the facility for **LOADing** one program from within another, and even for handing over variables from one program to the next. There are certain rules that have to be followed when using this technique if the whole thing is not to end in disaster. The variables are stored in the space in memory immediately after the end of a program, so if a longer program is **LOADed** the variables will be overwritten by the second program. When **LOAD** is used within a program it performs a **LOAD** and then **RUNs** the new program from the beginning, but without **CLearRing** the variable list and without resetting the **BASIC** pointers in zero-page. So if the second program is longer than the first it will corrupt itself by storing variable values over the extra piece of program. The rule to remember when using this technique is that the larger program must be **LOADed** first; even if this means padding out the first program in the series so that it becomes larger than the second one. Similarly, when handing over variables values, especially

numeric variables, the rule is that the second program must be smaller than the first.

In theory at least, you can create a hyper-mega-program made up of many smaller programs. Each individual program could be treated as a large subroutine to be pulled off the disk and into memory whenever it is needed by a core, driver program. Such a collection of programs could collectively be as large as 160K bytes. What a program! I leave it to you as an exercise to write a 160K program – we have the rest of this book to write!

On a more mundane but practical level, this third attribute of **LOAD**, the facility to **LOAD** programs from within a program, can be used to **LOAD** machine code programs that have previously been **SAVED** using a machine code monitor. In this case, the syntax:–

LOAD "M/C PROG NAME",8,1

will **LOAD** the program into the original memory locations. Since we are not now **LOADing** a **BASIC** program it is unfortunate that the **LOAD** routine **reRUNS** the program. In theory the following routine should work:–

10 LOAD "M/C NAME",8,1

20 REM

30 THE REST OF THE PROGRAM

Line 10 **LOADs** the piece of machine code and then the program should carry on to line 20 and onwards. Unfortunately it doesn't. After performing the **LOAD**, the program **reRUNS**, starting at line 10 again. The machine code is **LOADed** once more, the program restarts, and so on in an endless loop!

The accepted method of sidestepping this irritating quirk is to use the following:–

10 IF A=0 THEN A=1: LOAD "M/C NAME",8,1

20 REM

30 THE REST OF THE PROGRAM

On the first cycle, **A=0** so the **LOAD** is performed and then the program restarts. On the second cycle, however, the **IF . . . THEN** condition is not fulfilled (because **A=1**), so the program will now carry on to line 20 and the rest of the program.

The syntax of the **OPEN** statement is quite long by comparison with some of the other commands, but it is such an essential command that it is worth persevering with. The first parameter after the **OPEN** instruction sets the *logical file number*, which is a unique identification for that file. This can be any number within the range 0 to 255. File numbers greater than 127 (ie. 128–255) have a special purpose and are used with non-Commodore printers and when communicating with other computers so they should not normally be used.

The second parameter specifies the device that is to be communicated with. Device numbers must be within the range 0–31.

The third parameter defines the *secondary address*, or secondary command, for the device. The **OPEN** statement is very much a general-purpose command, but at this point it needs quite specific identification of the peripheral because each one will interpret a secondary command in a different way. (You can read in more detail about all the functions of secondary commands with the 1526 printer in Chapter Eleven. For information on other devices, consult the relevant user guide.) For the disk drive, the secondary command must be within the range 2–15. The values 0 and 1 are reserved for use by the **LOAD** and **SAVE** commands. Secondary command values within the range 2–14 will effect the **OPENing** of a **DATA** channel to the disk drive and allocate a 256 byte **DATA** buffer for use by that channel. A value of 15 **OPENS** the disk command channel (see Chapter Four – **DOS** commands).

The final parameter is the text string, that is, everything between the quotation marks. To make it possible to access **DATA** files each file on a disk must have a unique identifying name, just like a program. The file name can consist of up to 16 characters. Following that, you must specify the file **TYPE** after a comma. Your disk-drive supports four different types of file:–

1. **SE**quential
2. **PROG**ram
3. **USER**
4. **REL**ative

and you must indicate one of these: **SEQ**, **REL** and so on. The initial letter of each type is sufficient identification for the computer, but many people use the fuller format because it is easier to relate to. Finally, when **OPENing** a **SEQ**quential, **PROG**ram or **USER** file, you must specify the mode or direction of communication as either **READ** or **WRITE** (or their initials).

Let's look at an example to make things a bit clearer:—

10 OPEN 8,8,8,“STAMPS,S,W”

This statement **OPENS** a channel (8) to the disk drive to write a file called **STAMPS** onto the disk, using file number 8. The **OPEN** statement simply creates the file and if you wanted to fill it with **DATA** such as the details of your stamp collection, you would use the **PRINT#** command.

Similarly:—

20 OPEN 7,8,9,“BOOKS,S,R”

OPENS channel 9 for access to the file called **BOOKS**, from where **DATA** will be **READ** into file number 7 (using **GET#7 AND/OR INPUT#7**).

There are a few rules that must be followed when using I/O files:—

1. The computer does not allow more than 10 files to be **OPEN** simultaneously. This applies with all devices, not just the disk drive. So if you should need to use more than 10 files, **CLOSE** some of those that are already **OPEN** to allow space for the new ones. It is good programming practice always to **CLOSE** files as soon as you finish using them, so this limit should rarely be a constraint.
2. Within this overall maximum of 10, the disk drive allows a maximum of three files to be **OPEN** simultaneously.
3. When using **SEQ**quential, **PROG**ram or **USER** files, you can **OPEN** a channel to either **WRITE** or **READ** to a file but not to do both at the same time. If necessary, **CLOSE** the file after **WRITEing** and then re**OPEN** it for **READING**. The only type of file that allows concurrent **READING** and **WRITEing** is a **REL**ative file, which we deal with fully in Chapter 6.

WRITE AND REPLACE

We saw in the section on **SAVE** that it is possible to use a **SAVE** and **REPLACE** option when saving a program onto disk. In just the same way, you can **OPEN** a file to **WRITE** and **REPLACE** an existing file. The format for this is:—

10 OPEN 8,8,8,"C:STAMPS,S,W"

or

10 OPEN 8,8,8,"C0:STAMPS,S,W"

On the face of it this command has a number of advantages over the normal **WRITE** file format, not the least being that you can update your files quite easily by just completely overwriting the existing **DATA**. In fact, however, there is one big disadvantage with using this command: as soon as you **OPEN** the **WRITE** and **REPLACE** file, all the **DATA** in the original file of that name is wiped off the disk. From then on your **DATA** is held only in the computer's memory, assuming that you have previously **READ** it. From the point of view of data security this is an extremely unattractive position to be in because it leaves you so vulnerable to circumstances such as a power cut, which could clear your memory and leave you without a back-up on disk!

With important **DATA** always ensure that you keep a copy on another disk, just in case of accidents.

You will see in the chapter on **DOS** commands that there are several available which allow you to manipulate the disk files. You should therefore create a second file and fill it with your new data. Then delete the original file and rename the new one as the original.

PRINT#

FUNCTION: To write **DATA** items to a logical file. The file number must be the same as that used to **OPEN** it, and in the case of a data file it must have been **OPENed** for **WRITEing**.

SYNTAX: **PRINT#<FILE NUMBER>,<DATA LIST>**

ABBREVIATION: **P<SHIFT>R**

EXAMPLES: **PRINT#8,"HELLO"**
 PRINT#5,A\$;X;B\$
 PRINT#4,A\$;"";B\$;"

The **PRINT#** statement writes **DATA** to the designated logical file in exactly the same way as the **PRINT** statement writes **DATA** to the screen. That is to say, the **PRINT** modifiers (the semi-colon and comma) have the same effect upon the **PRINT#** command as they have upon **PRINT**. When you **PRINT#** data to a disk file it is important that you use the modifiers correctly so that later the **DATA** can be correctly **READ** using the command **INPUT#**. (see Chapter 5 for more details and examples).

INPUT#

FUNCTION: To read whole variables from a logical file. The file number must be the same as that used to **OPEN** the file, and in the case of a data file it must have been **OPENed** for **READING**.

SYNTAX: **INPUT#<FILE NUMBER>,<VARIABLE LIST>**

ABBREVIATION: **I<SHIFT>N**

EXAMPLES: **100 INPUT#8,A\$**
 110 INPUT#4,A,B,C\$

Just as with the **INPUT** command, **INPUT#** cannot be used in the direct mode, (typed in from the keyboard without a line number), but may only be used as part of a program.

The **INPUT#** command performs for **I/O** devices in a similar way to which **INPUT** works for keyboard inputs. It reads data into the declared variable until it reaches a delimiter which denotes the end of one variable and the beginning of the next. When using **INPUT** (from the keyboard) the delimiters are the carriage return (**CHR\$(13)**) and the comma (**CHR\$(44)**). **INPUT#** also uses these characters, plus the semi-colon (**CHR\$(59)**) to delimit variables in a list. (See Chapter 5 for more detail.) Note that if a numeric variable is expected (ie **INPUT#8,X**) and string **DATA** is received then your program will stop with a '**FILE DATA ERROR**' message (not '**BAD DATA ERROR**' as claimed in the CBM User Guide, Appendix L). The opposite is not the case however; numeric **DATA** can be read into a string variable and indeed this is the preferred method.

INPUT# can **READ** strings of up to 80 characters long and beyond that a '**STRING TOO LONG ERROR**' message will be generated.

GET#

FUNCTION: To read a single character from a logical file and store it as a variable. The file number must be the same as that used to **OPEN** it, and in the case of a disk data file it must have been **OPENed** for **READING**.

SYNTAX: **GET#<FILE NUMBER>,<VARIABLE LIST>**

ABBREVIATION: **NONE**

EXAMPLES: **GET#8,A\$**
 GET#4,A,B,A\$,B\$

The **GET#** statement reads characters one at a time from the specified device or file into computer memory. It works in the same way as the **GET** statement, except that the **DATA** comes from somewhere other than the keyboard. If no character is received, the variable is set to an empty string (equal to "" or **CHR\$(0)**), or to zero for numeric variables. Note that if a numeric variable is expected (ie **GET#8,X** and string **DATA** is received then your program will stop with a '**SYNTAX ERROR**' message. In fact there is no use for numeric variables within the **GET#** statement with

disk files, because all numeric data is converted to string data by the disk drive (as if a **STR\$** were performed) before it stores the data on disk with **PRINT#**.

GET# can only be used within a program, not in direct mode which will give you an '**ILLEGAL DIRECT ERROR**' message.

CMD

FUNCTION: To redirect primary output from the screen to the file specified. The file could be an Output device, like a disk, tape, or printer, or an I/O device such as a modem, or other RS232 facility.

SYNTAX: **CMD<FILE NUMBER>[,STRING]**

ABBREVIATION: **C<SHIFT>M**

EXAMPLES: **CMD 8**

CMD 8,"TITLE"

CMD 8,A\$

The **CMD** command switches primary output from the screen to a designated file. This must have previously have been **OPENed** to a device that is able to accept output, (not, for instance, the keyboard), and if a disk file, must be **OPENed** for **WRITEing**. Whilst **CMD** is in action the results of **PRINT** statements and **LIST** commands will not be displayed on the screen, but will be sent in the usual format to the file. To return to the normal output mode, on a television screen, you must send a **PRINT#** to the **CoMmanD** file.

When it is used as part of a program the **CMD** statement can be somewhat inconsistent regarding error reports. Systems errors such as '**SYNTAX ERROR**' will cause output to return to the screen. However, the disk drive will still be 'listening' for output so you must send a **PRINT#** immediately after this type of error. On the other hand, user errors such as '**EXTRA IGNORED**' or '**STRING TOO LONG**' in response to an incorrect **INPUT** are not generated at all and your program will continue in blissful ignorance of the mistake! The **CMD** command can be very useful, but if you use it within your own programs be careful. Only do so in situations where it is impossible for an error message to be generated.

The main application for **CMD** is in **LISTing** programs. The way to do this with a printer is:-

OPEN 4,4 <RETURN>

CMD 4,"TITLE": LIST <RETURN>

When the printer has finished you should redirect output back to the screen by typing:-

PRINT#4 <RETURN>

You will find that your user manual states that you must also **CLOSE** the file. This is not actually the case; a **PRINT#** command is all that is necessary to return output to the screen. You can also make a listing to a disk file. This can be very useful if you want to incorporate program listings into documents using a word-processor. An **ASCII** file of the listing can be made using **CMD**, and then the listing file can be merged into your document using the facilities of the word-processor. For other uses, however, **CMD** should be used with caution. It isn't much fun to crash out of a program with a '**SYNTAX ERROR**' half-way through writing important **DATA** onto a disk, with the file still listening for more **DATA**!

CLOSE

FUNCTION: To **CLOSE** a communications channel between the computer and the I/O device.

SYNTAX: **CLOSE <FILE NUMBER>**

ABBREVIATION: **CL<SHIFT>0**

EXAMPLE: **CLOSE 8**

N=2:CLOSE N

The **CLOSE** command quite literally **CLOSEs** the specified file. When you are using disk or cassette files it is very important that you **CLOSE** them properly, especially **WRITE** files. When a **WRITE** file is **OPENed**, an input buffer is created into which data is **PRINT#ed** and when the buffer becomes full its contents are recorded onto the disk (or cassette). The **CLOSE** command forces

the computer to record the contents of buffer, even if it's only partly full, onto the disk file, before updating the **BAM** and closing the file in the directory. As we said earlier, the **BAM** is used by the Disk Operating System to keep a record of which blocks are full and which are available for storage of new data, so if you don't **CLOSE** your datafile properly the **DOS** will not know that it is there and will quite happily corrupt it by recording over it. **READ** files should also be **CLOSEd** as soon as they are finished, but luckily there is no danger of losing **DATA** by not **CLOSEing** them. You can only have three files **OPEN** simultaneously so in any case it is always a good idea to **CLOSE** those that are not in use.

CLOSEing the command channel (channel 15) works slightly differently. Any file that is **OPEN** is automatically **CLOSEd** before the command channel itself. So if you are unsure exactly which files are **OPEN**, at the end of your program you can use the following line:—

10000 CLOSE 15: OPEN 15,8,15: CLOSE 15

The first statement **CLOSEs** file 15 and is there in case file 15 is being used for another device or for a data file. Then the command channel (channel 15) is **OPENed** and immediately **CLOSEd** again, forcing any other **OPEN** disk file to **CLOSE**.

4

The Disk Operating System

In the last chapter we looked at the **BASIC** commands that can be used with the disk drive. Although they are vitally important these commands are rather limited in function and they really only scratch the surface of the power and versatility of a disk system.

Commodore disk drives are 'intelligent' peripherals. The 1541, for instance, contains 16K of **ROM** memory, 2K of **RAM** and its own 6502 micro-processor. In terms of computing power, it is almost as powerful as the VIC-20 computer itself. In fact, the most significant difference between a computer and a disk drive is that while the computer can be programmed to perform a variety of tasks, the disk drive is dedicated to just one: the storage, retrieval and manipulation of data on a floppy disk.

To allow the disk drive to carry out this task there is a set of programs built into its circuitry called the Disk Operating System, or **DOS** for short. Just as the computer has a set of **BASIC** keywords that can be used to instruct it to perform specific functions, so the disk drive has a set of **DOS** keywords that command it to perform specific functions upon the disk. These **DOS** commands are what we shall be dealing with in this chapter. They can be divided into two categories: disk maintenance, which we shall look at first, and disk manipulation.

THE DOS SUPPORT PROGRAM

Before discussing the individual commands let us look first at the **DOS** support program that is one the demonstration disk supplied with your drive. Its object is to facilitate use of the **DOS**, mainly by

allowing you to abbreviate the commands quite considerably, and also by providing a facility for obtaining error messages when something has gone wrong.

There are, in fact, two programs on the disk: one each for the VIC-20 (VIC-20 WEDGE) and CBM 64 (C-64 WEDGE). When you **LOAD** and **RUN** the appropriate program you will find that it simplifies the use of many of the disk commands. For example, we have already seen that to format a disk the formula is:—

OPEN 15,8,15,“NEW:DISK NAME,ID”:CLOSE 8

The **DOS** support program allows you to abbreviate the command to:—

<NEW:DISK NAME,ID

which, I am sure you'll agree is much less typing. Many other commands can be used just as economically.

The other useful feature of the **DOS** support program is that it provides you with a means of reading the disk error channel. Whereas normally you just get a flashing red light and a blank screen to inform you of an error condition when you are programming, if you are using the support program and type in either an **C** or a **>** you will get an appropriate error message on the screen. If you are working in direct mode this is the only way to interpret an error signal from the disk drive, and it is a great aid to bug-haunted programmers.

One thing to remember about the **DOS** support functions is that they can only be used in the direct mode so you must still use the full version of each command within a program. Nevertheless, this utility can be very useful during the development of your own programs and it is worthwhile keeping a copy of it on your most frequently used disks. VIC-20 owners should have no trouble doing this, you simply **LOAD** the 'VIC-20 WEDGE' program from the demonstration disk and then immediately **SAVE** it onto your own before **RUN**ning it. However, Commodore 64 owners will find that you cannot follow the same procedure because the 'C-64 WEDGE' is actually a loader program for the main machine code program '**DOS 5.1**'. Here is what you have to do to **SAVE** it:—

1. LOAD and RUN the program 'C-64 WEDGE'

2. Type in the following **POKEs**:-

POKE 44,204:POKE 43,0 <RETURN>

POKE 46,207:POKE 45,89 <RETURN>

3. Now insert your work disk into the drive, and **SAVE** the program using:-

SAVE "DOS 5.1",8,1 <RETURN>

4. **VERIFY** the **SAVE** using:-

VERIFY "*",8,1 <RETURN>

5. If the **VERIFY** is OK then the program has been properly **SAVED**, if not, you will have to start again from the beginning.

6. The **DOS** support can be re**LOAD**ed at any time by entering:-

LOAD "DOS 5.1",8,1 <RETURN>

and re**RUN** by typing:-

SYS 52224 <RETURN>

DISK MAINTENANCE COMMANDS

When they are used as part of a program, all **DOS** commands have to be sent down the command channel (channel 15) in the form of a **COMMAND STRING**, that is, the proper syntactical format for that command enclosed in quotation marks. The command string should be sent using either the **OPEN** statement or a **PRINT#** command. For example:-

OPEN 15,8,15,"COMMAND STRING":CLOSE 15

or

OPEN 15,8,15

PRINT#15, "COMMAND STRING"

CLOSE 15

You can use any file number for the command file but it is most convenient to use the same one as the command channel, that is 15. As a general rule, disk maintenance commands should not be used if there are any **OPEN** data files on the disk. Because they

are performing 'housekeeping' duties **OPEN** files run the risk of becoming corrupted, so always **CLOSE** them.

NEW

FUNCTION: To prepare (format) an unused disk so that it is ready to store data and programs from your computer.

SYNTAX **"NEW:DISK NAME,ID"**
OR **"N:DISK NAME,ID"**

EXAMPLES: **OPEN 15,8,15,"NEW:MY DISK,84"**
 PRINT #15,"N:MY DISK,84"

The **NEW** command is used to **FORMAT** a blank disk ready for use. It can also be used to re**NEW** an old disk. Every disk must be processed in this way before any data files or programs can be stored on it. The formatting process magnetically marks out all the tracks and sectors for the first time, including the **BAM** and the directory.

The 'disk name' can be up to 16 characters long and will be stored in the directory along with the identifiers as a unique name for that disk. The 'id' or identifier can be any two characters you choose, but as a rule they should be a unique combination for each disk. As well as appearing in the directory, the identifier code is recorded on every block of the disk, so that every time the **DOS** accesses a block it can check that the correct disk is in the drive. As an example of how important this is, imagine a situation where you are **WRITE**ing a datafile to a disk. When the file is first **OPEN**ed, the current **BAM** is read by the disk drive to ensure that it will only **WRITE** the file to blocks that are available. When you get round to **CLOSE**ing the channel the **BAM** will be updated to accommodate the new file. In the meantime the **DOS** checks that the identifier of the disk is correct, relying on it as a form of shorthand for the full name. So if you were accidentally to replace that disk while the file was still **OPEN** with another disk having the same identifier the **DOS** would not register the change. It would then quite happily corrupt your new disk, working on it as if it were the original one. The corruption would be at its worst when you **CLOSE**d the file and the **BAM** from the original disk would be written right over that of the replacement!

This type of situation can be avoided quite easily by ensuring that you always use a unique code for each disk. The **DOS** will then spot any changeover and report an **ERROR** rather than carry on **WRITEing**. There are over 2000 combinations of a two character code, so you have no excuse for using your initials or 85 (for 1985) for all your disks! The **DOS** is there to help you so give it all the help that you can.

There is another use of the **NEW** command. You will probably find at some time that you have a disk full of programs and data files that you no longer need. In this case you can reformat it to provide you with the equivalent of a fresh disk again. As it has already been **FORMATted** once, the disk can be re**NEW**ed using an abbreviated format:—

OPEN 15,8,15,“N:DISK NAME”

or

PRINT #15,“N:DISK NAME”

Because you haven't specified an identifier in this case, a speedier form of formatting can be carried out. The tracks and sectors are already marked out and each block already carries an identifier. All this short form of **NEW** does is to clear the directory to show that there are no files stored on the disk and to clear the **BAM** so that all blocks are indicated to be free for future use. This takes about two seconds compared with around one and a half minutes for full formatting, and produces a recycled disk that you can use as new.

Although it is probably obvious, it must be stressed that you should be extremely careful about using this command. There is no faster way to lose up to 170K of your most useful programs and data files than to **NEW** your favourite disk by mistake! It only takes a few seconds to check that you are using the correct disk but it will take many hours to key-in your lost programs again. It is a mistake that you will only make once at most, but if you think carefully you shouldn't make it at all.

SCRATCH

FUNCTION: To delete a file permanently from a disk.

SYNTAX: **"SCRATCH:FILE NAME"**
 "S:FILE NAME"

EXAMPLES: **OPEN 15,8,15,"SCRATCH:STAMPS"**
 PRINT #15,"S:BOOKS"

The **SCRATCH** command is used to erase a file (data or program) from a disk. The syntax is quite simple: you specify the file to be deleted, preceded by '**SCRATCH:**' or the abbreviated form '**S:**'. The wild-card and pattern-matching characters, '?' and '*' can be used as a convenient way to **SCRATCH** multiple files if you are having a blitz. Using pattern-matching can be particularly useful when developing a program. For instance, you can **SAVE** the various stages of your program as **VERSION1**, **VERSION2**, **VERSION3** and so on, and the finished product as **FINAL VERSION**. Then you can quickly remove the development versions in one fell swoop using:-

OPEN 15,8,15,'S:VERSION*'

which will **SCRATCH** all files that begin with **VERSION**. If you read the error channel afterwards you will find that the number of files **SCRATCHed** will be given.

A word of warning: you can not discriminate when using pattern-matching between files that you want to keep and those you want to erase. "**S:T***" will **SCRATCH** every file beginning with that letter indiscriminately, so do check first in the directory that there are no valuable files also starting with **T**. What's more, the **SCRATCH** function is a permanent erasure. There is absolutely no way of recovering data once you have **SCRATCHed** it, so when you are using this command don't rush it, and double check everything first. Incidentally, it is only when you realise that, despite all precautions, you have inadvertently wiped out the wrong program, that you really have to answer for not having kept a backup copy!

RENAME

FUNCTION: To change the name of an existing file on a disk. The file must be properly **CLOSEd** before it is **RENAMED**.

SYNTAX: **“RENAME:NEWNAME=OLDNAME”**

“R:NEWNAME=OLDNAME”

EXAMPLES: **OPEN 15,8,15,“RENAME:FINAL=**
VERSION4”

PRINT #15,“R:FINAL=VERSION4”

The **RENAME** command is used to alter the name of a file that is already on the disk directory. The file, (program or data) must be properly **CLOSEd** before using this command, otherwise the **DOS** will refuse to carry it out. The operation is very fast because only the name in the directory is changed and no alterations are made to the file itself. Note that this command only changes the file name; it cannot be used to change the type. **RENAME** can be used effectively to update important data where it is imperative for security reasons that there is always a copy on disk. In these cases it is a much more reliable method than using **SAVE** and **REPLACE** or **WRITE** and **REPLACE**, and although a bit more laborious it is wellworth while. For instance, if you have a datafile that you wish to **READ** into memory, update and then **WRITE** back onto the disk for future use, then the safest procedure to follow is this:—

1. **READ** in the data file, for example **“RECEIPTS”**.
2. Update the contents of the file, by adding extra receipt information or whatever.
3. **WRITE** the amended data to a new file called, for instance, **“NEW RECEIPTS”**.
4. You now have two data files on the disk: one that is up to date, and the other now superceded – a nice safe position to be in. More importantly, you have never been in the position of having **NO** data on your disk.
5. Now delete the original file, **“RECEIPTS”** using **“SCRATCH:RECEIPTS”** or the abbreviation **“S:RECEIPTS”**.

6. Finally, change the name of the updated file **NEW RECEIPTS** to **"RECEIPTS"**, using **"RENAME:RECEIPTS=NEW RECEIPTS"** or the abbreviated form.
7. You can now begin again using the latest file of data as the starting point and at no time have you been in danger of losing anymore than the current update of the file.

Note that you cannot change the name of a file to a name that already exists in the directory, just as you cannot **SAVE** a program under a name that already exists.

COPY

FUNCTION: To make a copy of any program or data file onto the same disk as the original.

SYNTAX: **"COPY:NEWFILE=OLD FILE"**

"C:NEWFILE=OLDFILE"

EXAMPLES: **OPEN 15,8,15,"COPY:BACKUP=RECEIPTS"**

PRINT #15,"C:BACKUP=RECEIPTS"

The **COPY** command is used to create a second identical version of a file. The name you give the copy must, as always, be unique and not already in existence on the disk directory. This command cannot be used to transfer copies between physically separate devices (two CBM 1541s, for instance) but it can **COPY** between drives on a dual drive unit, (such as the CBM 4040). When using a single drive disk unit you are restricted to **COPYing** onto the same disk as the original file is on. This can provide a limited though very valuable degree of backup security for your important programs and files. Two copies of a file are better than one after all, but for total security the copies should be on separate disks, just in case an entire disk becomes corrupted.

COPY AND CONCATENATE

The **COPY** command has an added feature which can often be more useful than its primary purpose. You can combine up to four individual files into one, new, larger file with a single **COPY** command. This is known as **CONCATENATION** and the syntax is:—

**“COPY:NEW FILE=OLDFILE1,OLDFILE2,OLDFILE3,
OLDFILE4”**

You can use the abbreviated form here too. If you want to concatenate less than four files then just specify their relevant names. Any number of files can be joined together by repeating this function several times. For example, to combine 6 files, named **OLDFILE1** to **OLDFILE6** you could use:—

**PRINT#15,“C:NEWFILE1=OLDFILE1,OLDFILE2,
OLDFILE3”**

**PRINT#15,“C:NEWFILE2=OLDFILE4,OLDFILE5,
OLDFILE6”**

PRINT#15,“BIGFILE=NEWFILE1,NEWFILE2”

It is not possible to insert a file in the middle of another, but you can combine files in such a way that the newest data is at the beginning of the concatenated datafile by using the construct:—

“BIGFILE=NEWFILE,OLDFILE”

Whatever you put first after the equals sign will take precedence in the **COPY**ing operation, and material from the other files will be ordered accordingly. This can be very useful when storing data about, for instance, VAT returns or sales and costs figures. In that case, the most recent data chronologically is also the most important, and by putting it at the beginning it can be accessed most quickly. Earlier data can then be read in afterwards. If the most recent information is put at the end of the file then it will be accessed last. This could be quite a time consuming process with large datafiles and would detract from the professionalism of your program.

You can use the **COPY** and **CONCATENATE** structure to combine program files as well as datafiles. This does not, however, ‘merge’ two programs together because the end-of-

program marker for the first program prevents this.

INITIALIZE

FUNCTION: To reset the disk drive as if it had just been turned on. There must be a disk in the drive when this command is used.

SYNTAX: **"INITIALIZE"**
 "I"

EXAMPLES: **OPEN 15,8,15,"INITIALIZE"**
 PRINT #15,"INITIALIZE"
 PRINT #15,"I"

The command **INITIALIZE** is used to return the disk drive to its power-on condition. This means that all the buffers are cleared and any data in them is lost. Also any pointers that are in use will be reset and therefore these two will be lost.

INITIALIZE does just as it says. It literally initializes the disk drive to the disk in it, that is, it reads in the identifier code and the current **BAM** so that it knows which disk it is dealing with and which blocks are free for the allocation of data. On earlier models of Commodore disk drives this process was compulsory every time a different disk was inserted. Those disk drives had what is known as **DOS** version 1. The 1541, which includes the built-in disk unit in the SX-64 portable, contains an upgraded **DOS**, version **2a**, which automatically initializes a disk on the first **OPEN** or **SAVE** command. The **INITIALIZE** statement is really rather redundant on the 1541 but it is there to provide compatibility for programs written for other Commodore disk units. Under normal circumstances it is not necessary to use **INITIALIZE** for this purpose.

It does have a useful function, though. At times there may be an error condition on the disk which is preventing you from continuing. Using **INITIALIZE** will free the disk and drive from this 'hang-up' and return control to the computer (and you!). But be careful, any files that are **OPEN** at the time for **WRITE**ing will not be **CLOSE**d properly and the **BAM** will not be updated to

accommodate them. This is a very dangerous situation because it leaves a file vulnerable to being overwritten by subsequent operations and therefore corrupted. Furthermore, you can only tidy the disk up again by performing a '**VALIDATE**' or '**SCRATCH**' upon it, or by deleting the **OPEN** files. Whichever way, you stand to lose all of the data that is in these un**CLOSE**d files. The **INITIALIZE** command therefore, should only be used with great care and as a last resort.

VALIDATE

FUNCTION: To reorganize file storage on a disk so as to make best use of the available space.

SYNTAX: "**VALIDATE**"

EXAMPLES: **OPEN 15,8,15,"VALIDATE"**

PRINT #15,"V"

After a disk has been in use for some time the continuous **SAVE**ing and **SCRATCH**ing of files leaves pockets of blocks that are empty but are too short to be of any practical use. Moreover, when a file is **SCRATCH**ed, although it no longer shows up when the directory is listed its entry is still stored there, and a pointer is set to indicate to the **DOS** that it has been deleted. The **VALIDATE** command cleans up the disk, removing deleted file names completely from the directory and reorganizing those files still on the disk to optimise the space available.

This command should never be used with a disk that contains direct access files, because blocks that have been allocated for use will be de-allocated by the operation of the **VALIDATE** statement.

READING THE ERROR CHANNEL

Unless you use the **DOS** support program it is not possible to read the disk error channel in direct mode. You have to **INPUT#** data from the error channel and this can only be performed within a program. Here is a useful **BASIC** routine for reading the error channel:—

```
1000 OPEN 15,8,15
```

```
1010 INPUT#15,EN,EM$,ET,ES
```

```
1020 PRINT EN;EM$,ET,ES
```

The variable names have been chosen to remind you of their uses. **EN** inputs the Error Number, **EM\$** reads the Error Message whilst **ET** and **ES** will contain the Track and Sector where the error occurred. When you are using files, it is important that you keep track of any possible errors that may occur and, of course, take appropriate action. There is a full list of the disk error messages in Appendix II.

DISK MANIPULATION COMMANDS

The advantage of these commands is that they allow you direct access to the disk surface and memory buffers, and by using them, you can wield complete control over the structure of your fields, records and files. However, because they are very powerful instructions they should be used with great care, especially in the early stages when you may not always be sure exactly what is happening.

In the following pages we shall be looking at each command individually, and then in Chapter 7 we shall consider using them in combination applied to direct access files. Until you have some experience of this type of file and the commands that go with it, you are strongly recommended to use a disk that does not contain any important information and to get plenty of practice on it before you try serious applications.

OPEN (see also Chapter 3)

FUNCTION: To allocate a buffer to a communication channel for use with a direct access file.

SYNTAX: **OPEN FILE NUMBER,DN,CHANNEL
NO "# [BUFFER NO]"**

EXAMPLES: **OPEN 2,8,2,"#1"**

In this more specialized application, the **OPEN** command allocates a buffer within the disk drive **RAM** to a channel, for use with direct access commands. A channel of communication must be established before anything can be **PRINT**ed, **INPUT**#ed or **GET**#ed from the buffer. You can specify a particular buffer by number if you want to, or failing that the **DOS** will allocate the next available one. You can then immediately find out which buffer has been allocated by **GET**#ing its number from the command channel. The following routine demonstrates this:—

```

10 REM *****
12 REM * OPEN A BUFFER *
14 REM * DEMO PROGRAM *
16 REM *****
20 OPEN 15,8,15: REM COMMAND CHANNEL
30 OPEN2,8,2,"#":REM ANY BUFFER
40 GET#2,A$
50 PRINT"YOU ARE ALLOCATED BUFFER NO. ";
60 PRINT ASC(A$+CHR$(0))
70 CLOSE 2: CLOSE 15

```

Note that you must perform the **GET**# before any other instruction to the nominated channel otherwise the character will be lost. You should then set a variable to the buffer number and use that variable whenever a reference to that buffer is required. This feature can be very useful when you are unsure which buffers are available at a particular point within a program.

BLOCK-ALLOCATE

FUNCTION: To allocate or reserve a specified block on a disk for your own use.

SYNTAX: “BLOCK-ALLOCATE:“DR;T;S
 “B-A”DR;T;S

EXAMPLES: PRINT#15,“BLOCK-ALLOCATE:”
 0;1;1
 T=1:S=1:PRINT#15,“B-A”0;T;S

The **BLOCK-ALLOCATE** command can be used to effectively set aside a given block on a disk for your own purposes. First, the **BAM** checks to ensure that the required block is available. If it is free the **BAM** is updated to indicate that it is in fact in use, so that any data you then put into that block will be protected from corruption by inadvertent overwriting. If, however, the specified block has already been allocated, either by a previous **BLOCK-ALLOCATE** instruction or by the storage of data, then the error message no. 65:

“NO BLOCK T,S”

will be returned. It is very important that you read the error channel every time you use this command. There are two reasons for this. Firstly, if a block is already in use you run the risk of corrupting the data stored there by your subsequent operations. Secondly, the numbers **T** and **S** which are returned with error message 65 give the track and sector address of the next highest available block. So if your chosen block is not free for some reason you can use this address as a safe substitute. The following demonstration program allocates 10 blocks on a disk and **PRINTS** an error report onto the screen after every operation so that you can see what is happening. Try typing it in and **RUN**ning it two or three times:—

```

10 REM *****
12 REM *      BLOCK-ALLOCATE      *
14 REM *      DEMO PROGRAM        *
16 REM *****
20 OPEN 15,8,15
30 OPEN 2,8,2,"#"
40 FOR T=1 TO 10
50 PRINT#15,"B-A"0;1;T
60 GOSUB 1000
70 IF EN=65 THEN PRINT#15,"B-A"0;ET;ES
80 NEXT T
90 CLOSE2 : CLOSE15 : END
1000 REM *****
1002 REM *  READ THE ERROR  *
1004 REM *    CHANNEL      *
1006 REM *****
1010 INPUT#15,EN,EM$,ET,ES
1020 PRINT EN;EM$;ET;ES
1030 RETURN

```

On the first **RUN** of the program you should find that there is no error message and each block can be allocated. When you **RUN** it a second time, however, error 65 is displayed and the track and sector address that is returned is allocated by line 70. If you were to **RUN** the program about 65 times you would eventually receive the error message:

“65 NO BLOCK 00 00”

This indicates that no higher blocks are available.

Note that if you happen to use the “**VALIDATE**” command it will automatically de-allocate any blocks that have been reserved using **BLOCK-ALLOCATE**. So never **VALIDATE** a disk that contains user-allocated blocks or direct access files – they will immediately be wiped out!

BLOCK-FREE

FUNCTION: To de-allocate a specific block that has been reserved using the **BLOCK-ALLOCATE** command.

SYNTAX: **"BLOCK-FREE:"DR;T;S**
 "B-F"DR;T;S

EXAMPLES: **PRINT#15,"BLOCK-FREE:"0;1;1**
 T=1:S=1:PRINT#15,"B-F"0;T;S

The **BLOCK-FREE** command is the opposite of **BLOCK-ALLOCATE**. It frees an allocated block to become available for **DOS** use for storing datafiles or programs. It operates by updating the **BAM** to declare that block free. There are no associated error message with this command so you must always ensure that you identify the correct block or blocks; freeing those that have been assigned by the **DOS** to datafiles or programs will invite corruption of these files by subsequent writing operations.

BLOCK-READ

FUNCTION: To move the contents of a block of data from the disk into the buffer of a selected channel. The channel must previously have been **OPENed** and assigned a memory buffer.

SYNTAX: **"BLOCK-READ:"CH;DR;T;S**
 "B-RMCH;DR;T;S

EXAMPLES: **PRINT#15,"BLOCK-READ:"2;0;1;1**
 T=1:S=1:PRINT#15,"B-R"2;0;T;S

The **BLOCK-READ** command will literally **READ** data from a specified block into the memory buffer of a given channel. Having filled the buffer, you can then use **GET#** and **INPUT#** statements to read the data into the computer.

This command can **READ** up to an entire block of 256 characters into the buffer, but it will stop **READING** when the buffer pointer stored within the block indicates that it is finished.

The following demonstration will allow you to **READ** any block on your disk and **PRINT** the contents onto the screen. **READING** blocks of stored program will probably not make a lot of sense because this data is stored in tokenized form, so a simple printout may not always be readable. However, datafile blocks should be recognisable.

```

10 REM *****
12 REM *          BLOCK-READ          *
14 REM *          DEMO PROGRAM        *
16 REM *****
20 OPEN 15,8,15
30 OPEN 2,8,2,"#"
40 PRINT CHR$(147)
50 INPUT "WHICH TRACK 1-35";TRACK
60 IF TRACK<1 OR TRACK>35 THEN GOTO 40
70 IF TRACK<=17 THEN B=20
80 IF TRACK>17 AND TRACK<=24 THEN B=18
90 IF TRACK>24 AND TRACK<=30 THEN B=17
100 IF TRACK>30 THEN B=16
110 PRINT:PRINT"WHICH BLOCK 0-";B;
120 INPUT BLOCK
130 IF BLOCK<0 OR BLOCK>B THEN GOTO 110
140 PRINT#15,"B-R"2;0;TRACK;BLOCK
150 FOR BYTE=0 TO 255
160 GET#5,CHARACTER$
170 IF STATUS=0 THEN BLOCK$=BLOCK$+CHARACTER$
180 IF STATUS=64 THEN BYTE=255:REM END OF DATA
190 NEXT BYTE
200 PRINT BLOCK$
210 CLOSE 2 : CLOSE 15
220 INPUT"ANOTHER BLOCK Y/N";REPLY$
230 IF REPLY$="Y" THEN RUN
240 IF REPLY$<>"N" THEN GOTO 220
250 END
READY.

```

When you use this type of routine within your own programs you will no doubt wish to manipulate the data that has been **READ** in rather than simply **PRINTing** it onto the screen.

BLOCK-WRITE

FUNCTION: To move the contents of a given channel buffer into a specific block on the disk.

SYNTAX: "BLOCK-WRITE:"CH;DR;T;S

"B-W"CH;DR;T;S

EXAMPLES: PRINT #15, "BLOCK-WRITE:"2;0;1;1

T=1:S=1:PRINT #15, "B-W"2;0;T;S

This command has the reverse function of **BLOCK-READ** and transfers data in the opposite direction. First the buffer is filled by **PRINT** #ing data through the specified channel, (channel 2 in the examples above). As the buffer fills up the **DOS** keeps a pointer to show how many characters there are and when the **BLOCK-WRITE** is performed, this pointer is also recorded onto the block.

It should go without saying that **BLOCK-WRITE** overwrites the current contents of a block, so it should be used with caution to avoid corrupting important data.

BLOCK-EXECUTE

FUNCTION: To read the contents of a specified block into the buffer of a given channel and then run it as a machine code program. This program *MUST* end with an **RTS** to return control to the main program.

SYNTAX: "BLOCK-EXECUTE:"CH;DR;T;S

"B-E"CH;DR;T;S

EXAMPLES: PRINT #15, "BLOCK-EXECUTE:"
2;0;1;1

T=1:S=1:PRINT #15, "B-E"2;0;T;S

This command is very similar to the **BLOCK-READ** in that it reads a specific block into the channel buffer. It differs, however, in that it then **EXECUTES** the buffer contents as a machine code program, just as **SYS** will run a machine code program in the computer's memory.

The object code should previously have been written to the block using **BLOCK-WRITE**, and it must end with an **RTS (ReTurn from Subroutine)** to restore control to the main calling program. In both the examples above, the contents of track 1, sector 1 are loaded into channel 2 and then **EXECUTED** as machine code.

Failure to include an **RTS** in your routine will cause the program to continue through the nominated buffer and then precariously through higher memory, effectively crashing the whole system. The advantage of this command is that when you use a machine code routine stroed within a block, you do not have to **WRITE** the data to the block every time you want to use it. Once it has been stored, it can be called whenever necessary using **BLOCK-EXECUTE**.

The following demonstration program fills block 1,1 with a machine code program that duplicates the contents of buffer no. 1 in buffer no. 2 whenever the **BLOCK** is **EXECUTED**. Although this is only a demonstration routine it can be used to good effect within your own programs to quickly duplicate data files while the files are still **OPEN** and without having to **WRITE** the data twice to the disk.

```

10 REM *****
12 REM *      BLOCK  EXECUTE      *
14 REM *      DEMO  PROGRAM      *
16 REM *****
20 OPEN 15,8,15:REM COMMAND CHANNEL
30 OPEN 2,8,2,"#0"
40 OPEN 3,8,3,"#1"
50 OPEN 4,8,4,"#2"
60 FOR LOOP=1 TO 14
70 READ MACHINE CODE
80 PRINT#2,CHR$(MACHINE CODE);
90 NEXT LOOP
100 PRINT#15,"B-W"0;1;1
110 INPUT" BUFFER 1,PLEASE";B1$
120 PRINT#15,"B-P"3;1
130 PRINT#3,B1$
140 INPUT" NOW BUFFER 2";B2$
150 PRINT#15,"B-P"4;1
160 PRINT#3,B2$
170 PRINT

```

```

180 PRINT"PRESS ANY KEY TO PERFORM BLOCK
-EXECUTE "
190 GET KEY$: IF KEY$<>" " THEN GOTO 190
200 GET KEY$: IF KEY$="" THEN GOTO 200
210 PRINT#15,"BLOCK-EXECUTE"2;0;1;1
220 PRINT#15,"BUFFER-POINTER"3;1
230 PRINT#15,"BUFFER-POINTER"4;1
240 INPUT#3,A$
250 INPUT#4,B$
260 PRINT" BUFFER 1=";A$
270 PRINT
280 PRINT" BUFFER 2=";B$
290 CLOSE2 : CLOSE3 : CLOSE4 : CLOSE15
300 END
1000 REM *****
1002 REM *   DATA FOR MACHINE   *
1004 REM *   CODE ROUTINE       *
1006 REM *****
1010 DATA 162,0,189,0,4,157,0,5,232,224,
0,208,245,96

```

BUFFER-POINTER

FUNCTION: To position the pointer within the specified buffer to the nominated position. The next **PRINT#**, **INPUT#** or **GET#** statement will start from that position.

SYNTAX: **"BUFFER-POINTER:"CH,POS**
 "B-P"CH,POS

EXAMPLES: **PRINT#15,"BUFFER-POINTER"3;1**
 CH=3:POS=1:PRINT#15,
 "B-P"CH,POS

Whenever a **PRINT#**, **INPUT#**, or **GET#** command is performed to the buffer of the specified channel, the **DOS** keeps a record of the current position within that buffer. **BLOCK-WRITE** commands write the pointer position along with the block to the disk and **BLOCK-READ** commands will only read a block into the buffer up

to the position of the previously saved pointer. All of this is performed automatically by the **DOS**. However, by using the **BUFFER-POINTER** command you can control the position of the pointer yourself. In the **BLOCK-EXECUTE** routine shown in the previous section, you will see that the buffer pointer is set to character 1 before any other operation. Each block on a disk holds 256 bytes or characters of data and the **BUFFER-POINTER** must be in the range 0-255.

BUFFER-POINTER, then, allows the pointer to be positioned anywhere within the buffer so that subsequent operations will begin from that point. For instance, up to 256 separate pieces of single byte data can be stored and accessed within a single block by firstly positioning the pointer (with **B-P**) and then using **PRINT#** to write the next byte or **GET#** to read the next byte. Using **INPUT#** will read in everything up to a carriage return (**CHR\$(13)**), a comma (**CHR\$(44)**) or a semi-colon (**CHR\$(59)**), up to a maximum of 80 characters. If you are unsure of the length of the data to be read in, it is better to use **GET#**. You should note that the pointer for a buffer is reset whenever a new block of data is read into that buffer. On the other hand, the command only relates to the assigned buffer of the specified channel and the previously set pointers to the other buffers are unaffected.

The next demonstration program shows the effect of the **BUFFER-POINTER** command. You are first asked to enter a word or phrase (UP TO 40 CHARACTERS). This sequence is then written into buffer no 1. **B-P** is used to position the pointer successively one character backward within the buffer and read back the result (and **PRINT** it onto the screen), reversing the phrase.

```

10 REM *****
12 REM *      BUFFER-POINTER      *
14 REM *      DEMO PROGRAM        *
16 REM *****
20 OPEN 15,8,15:REM COMMAND CHANNEL
30 OPEN 2,8,2,"#1"
40 PRINT" PLEASE TYPE A WORD OR PHRASE"
50 INPUT PHRASE$
60 PRINT:PRINT PHRASE$
70 COUNT=LEN(PHRASE$)

```

```

80 IF COUNT>40 THEN PRINT"NO MORE THAN 4
0 CHARACTERS": GOTO 50
90 PRINT#15,"B-P"1;1
100 PRINT#2,PHRASE$
110 FOR NUMBER=COUNT TO 1 STEP -1
120 PRINT#15,"B-P"1;NUMBER
130 GET#2,CHARACTER$
140 PRINT CHARACTER$;
150 NEXT NUMBER

```

In this example the data is not actually placed onto the disk but obviously in a real-life situation you would want to read and/or write the information onto permanent storage. You would, no doubt, also want to perform more useful functions with the data!

USER1 or USERA

FUNCTION: To read a complete block from the disk into the buffer of the specified channel.

SYNTAX: "USER1:"CH;DR;TRACK;BLOCK
 "U1"CH;DR;TRACK;BLOCK
 "USERA:"CH;DR;TRACK;BLOCK
 "UA"CH;DR;TRACK;BLOCK

EXAMPLES: PRINT#15,"USER1:"3;0;1;1
 TR=1:SC=1:PRINT#15,"U1"3;0;TR;
 SC
 PRINT#15,"USERA:"3;0;1;1
 TR=1:SC=1:PRINT#15,"UA"3;
 0;TR;SC

The **USER1** (or **USERA**, the two forms can be used interchangeably) command is a special form of **BLOCK-READ**. **USER1** reads the specified block into the buffer of the nominated channel, just as **BLOCK-READ** does. With **BLOCK-READ**, the

read is performed until the current buffer-pointer, recorded along with the block contents, indicates that the data in the block is complete. The **USER1** command also reads in a block from the disk but with one important difference: the buffer pointer is first pushed to position 255 so that the entire block will be read.

USER1 should be used in preference to **BLOCK-READ** when non-text data is stored within a block. On the other hand, if you are storing text, names and address for instance, it would be faster to read only the stored contents of the block rather than the entire 256 bytes. See also **USER2**, below.

USER2 or USERB

FUNCTION: To write the entire buffer of the specified channel to the nominated block.

SYNTAX: “**USER2:**”CH;DR;TRACK;BLOCK

 “**U2**”CH;DR;TRACK;BLOCK

 “**USERB:**”CH;DR;TRACK;BLOCK

 “**UB**”CH;DR;TRACK;BLOCK

EXAMPLES: PRINT #15,“**USER2:**”3;0;1;1

 TR=1:SC=1:PRINT #15,“**U2**”3;0;TR;
SC

 PRINT #15,“**USERB:**”3;0;1;1

 TR=1:SC=1:PRINT #15,“**UB**”3;
0;TR;SC

The **USER2** command is a special form of **BLOCK-WRITE**. Again **USER2** and **USERB** can be used interchangeably. When using **BLOCK-WRITE** the contents of the specified channel buffer are written onto the nominated block up to the position of the buffer-pointer. The **USER2** command, on the other hand, writes the entire 256 byte contents of the buffer onto the disk without disturbing the previously recorded **BUFFER-POINTER**.

USER2 and **USER1** should be used, in preference to **BLOCK-WRITE** and **BLOCK-READ**, in any situation where data is being written into the buffer in a non-sequential manner. For instance, the following example will write two words into the buffer. Starting at position 8 and **PRINT#**ing "DAVE", then coming back to position 1 in the buffer to **PRINT#** the word "HELLO". If the buffer is **WRITE**n to a disk and then **READ** back again you would expect to be able to get back from the buffer "HELLO DAVE". Try this example program and notice the difference in execution between the four commands.

```

10 REM *****
12 REM *      USER 1 AND USER 2  *
14 REM *      DEMO PROGRAM      *
16 REM *****
20 OPEN 15,8,15:REM COMMAND CHANNEL
30 OPEN 2,8,2,"#2"
40 PRINT#15,"B-P"2;8
50 PRINT#2,"DAVE "
60 PRINT#15,"B-P"2;1
70 PRINT#2,"HELLO "
80 PRINT"THE POINTER IS NOW AT POSITION
6"
90 PRINT#15,"B-W"2;0;2;2
100 PRINT:PRINT"BLOCK READ GIVES:-";
110 PRINT#15,"B-R"2;0;2;2
120 INPUT#2,A$,B$:PRINT A$;B$
130 PRINT" USER 1 GIVES   :-";
140 PRINT#15,"U1"2;0;2;2
150 INPUT#2,A$,B$:PRINTA$;B$
160 PRINT
170 PRINT" USING 'USER2', THE SECOND WOR
D IS NOW  OVERWRITTEN BY 'FRED'"
180 PRINT#15,"B-P"2;8
190 PRINT#2,"FRED "
200 PRINT#15,"U2"2;0;2;2
210 PRINT:PRINT"BLOCK READ GIVES:-";
220 PRINT#15,"B-R"2;0;2;2
230 INPUT#2,A$,B$:PRINTA$;B$
240 PRINT" USER 1 GIVES   :-";
250 PRINT#15,"U1"2;0;2;2
260 INPUT#2,A$,B$:PRINTA$;B$
270 PRINT

```

```

280 PRINT" USING 'BLOCK-WRITE' THE SECON
D WORD IS      NOW OVERWRITTENBY 'BILL'"
290 PRINT" AND THE POINTER IS NOW AT 12
"
300 PRINT#15,"B-P"2;8
310 PRINT#2,"BILL "
320 PRINT#15,"B-W"2;0;2;2
330 PRINT:PRINT"BLOCK READ GIVES:-";
340 PRINT#15,"B-R"2;0;2;2
350 INPUT#2,A$,B$:PRINTA$,B$
360 PRINT" USER 1 GIVES  :-";
370 PRINT#15,"U1"2;0;2;2
380 INPUT#2,A$,B$:PRINTA$,B$
390 CLOSE2:CLOSE15

```

You should be able to see from the example that the position of the pointer within the buffer is important when using the commands **BLOCK-READ** and **BLOCK-WRITE**, but has no role effect when using the **USER** commands. In the first part, for instance, the **B-R** only reads into the buffer the word 'HELLO' so the **INPUT#** of **A\$** and **B\$** (line 120) only returns **A\$="HELLO"** and **B\$** is set to the null character **CHR\$(0)**. When using **U1** to read the block, as in line 140, the entire block is read into the buffer allowing the **INPUT#** of line 150 to read both words. It is a good idea if you follow the flow of the program through with this demonstration, noticing the way that the instructions effect the buffer-position and the contents of the buffer.

MEMORY-READ

FUNCTION: To read the contents of the nominated memory locations within the disk memory.

SYNTAX: **"M-R"CHR\$(LOW BYTE)CHR\$(HIGH
BYTE)CHR\$(NO. OF BYTES)**

EXAMPLES: **PRINT#15,"M-R"CHR\$(10)CHR(0)
CHR\$(20)**

**HI=0:LO=10:CHAR=20:PRINT#15,
"M-R"CHR\$(LO)CHR\$(HI)CHR\$(CHAR)**

The **MEMORY-READ** command is used to read any of the memory locations within the disk drive. The address of the starting location must be in the range **0–65535** although not all of the addressable space is actually used on the 1541.

The address has to be sent to the disk drive in the standard 6502 format of low byte, high byte, each as a character. To calculate these values from your address use:–

$$\text{LOW BYTE} = \text{INT}(\text{ADDRESS}/256)$$

$$\text{HIGH BYTE} = \text{ADDRESS} - (\text{LOW BYTE} * 256)$$

For example, if you want to **READ** the contents of memory location 1026 you would use:–

$$\text{LO} = \text{INT}(1026/256) = 4$$

$$\text{HI} = 1026 - (4 * 256)$$

$$= 1026 - 1024$$

$$= 2$$

PRINT #15,"M-R"CHR\$(2)CHR\$(4)CHR\$(1)

MEMORY-READ reads the contents of the specified locations into the command channel and you then have to **GET#** them from there, a single character at a time. With a single instruction you can read between 1 and 256 locations by specifying the **CHAR** variable between **CHR\$(1)** up to **CHR\$(255)** and **CHR\$(0)** representing the highest number that can be read, 256 bytes. You should note that you have to take all the specified bytes of data from the command channel before any other action can be performed through that channel.

MEMORY-WRITE

FUNCTION: To write bytes into the memory of the disk-drive.

SYNTAX: **"M-W"CHR\$(ADDRESS LOW)CHR\$(ADDRESS HIGH)CHR\$(NO OF CHARACTERS) DATA**

EXAMPLES: PRINT#15,"M-W"CHR\$(0)CHR\$(3)CHR\$(1)CHR\$(96)"HELLO"

A\$="HELLO":PRINT#15,"M-W"CHR\$(0)CHR\$(3)CHR\$(6);A\$

The **MEMORY-WRITE** command is used to **POKE** bytes within the **RAM** memory of the disk drive, from 0 to 2047. You must specify the start address in low-byte/high byte form as with **MEMORY-READ**. In the examples above, the start address is $0 + (3 \times 256) = 768$. You must also state the number of bytes to be written into memory, up to a maximum of 34. The data itself can be in the form of **CHR\$(x)** for each character or, if the characters are printable then you can use them within **QUOTE MARKS**. You can also use a string variable within the instruction, the variable having been previously set to the string of characters that you want to write, as in the second example above, or even a numeric variable. The data could be a machine code routine that you may wish to use.

Alternatively, you may wish to use **MEMORY-WRITE** as an alternative to **BUFFER-POINTER** for writing data into the buffer. The demonstration program shows how this can be achieved:-

```

10 REM *****
12 REM *   MEMORY-WRITE   *
14 REM *   DEMO   PROG   *
16 REM *****
20 OPEN 15,8,15: REM COMMAND CHANNEL
30 OPEN 2,8,2,"#0"
35 A$="HELLO"
40 PRINT#15,"M-W"CHR$(0)CHR$(3)CHR$(6);A$
$
50 PRINT#15,"B-P"2;0
60 INPUT#2,B$
70 PRINT B$
80 CLOSE 2 : CLOSE 15

```

In this example the pointer is set to position 0 before using **INPUT#** to read back the data. Although using **MEMORY-READ** is faster than positioning the **BUFFER-POINTER** and then **PRINT#**ing to the buffer, the reverse is not the case. It is actually quicker to **INPUT#** data from the channel buffer than to perform

successive **MEMORY-READs** and **GET#s**. As an exercise you might like to alter lines **50** and **60** so that the data is read by using **MEMORY-READ**.

MEMORY-EXECUTE

FUNCTION: To **RUN** a machine-code routine that starts at a specified address.

**“M-E:”CHR\$(LOW BYTE)CHR\$(HIGH
BYTE)**

**“MEMORY-EXECUTE:”CHR\$(LOW BYTE)
CHR\$(HIGH BYTE)**

EXAMPLES: PRINT#15,“M-E”CHR\$(10)CHR\$(0)

HI=0:LO=10:

PRINT#15,“M-E”CHR\$(LO)CHR\$(HI)

The **MEMORY-EXECUTE** command is used to **RUN** a machine code program that is in the memory of the disk drive. The routine can be either in the **DOS**, which is in **ROM**, or in the 2k of **RAM** that is available. By far the most frequent **RAM** based use for **MEMORY-EXECUTE** is the re-execution of a routine that has already been loaded into a buffer using **BLOCK-READ** or, more usually, **BLOCK-EXECUTE**. In such a case, of course, **MEMORY-EXECUTE** will be considerably faster than **BLOCK-EXECUTE** because the code does not have to be reloaded.

CLOSE

FUNCTION: To end the association between a channel and a buffer used for direct access.

SYNTAX: CLOSE x

EXAMPLES: CLOSE 2

N=2: CLOSE N

See also the section on **CLOSE** in Chapter 3. In the particular case of **CLOSE**ing files that have been used for direct access, this **CLOSE** command does nothing more than disassociate the channel and the buffer to free them both for further use. Any data left in the buffer is not wiped out until the next **PRINT#** instruction is performed to that channel. This fact can be used to good effect because there are a total of eight buffers available on the 1541, although only three can be assigned at any one time. The manual actually states that five channels can be **OPEN**ed simultaneously, but this is apparently not true on the 1541, the maximum is three. The first three 256 byte blocks of memory, 0–255, 256–511 and 512–767, are used by the **DOS** and you are advised not to interfere with those unless you are sure you know exactly what you are doing. The next five blocks from 768 onwards are user buffers for your use. When you are using machine code routines, for example, you can **OPEN** a channel to a particular buffer, use **BLOCK-READ** to fill it with your routine and then **CLOSE** the channel. In this way you can use **MEMORY-EXECUTE** to **RUN** the machine code routine whilst at the same time leaving three channels free for other purposes. The alternative would be to use **BLOCK-EXECUTE** which would, of course, mean that a channel must be left **OPEN**. Using **MEMORY-EXECUTE** allows you an extra channel which one day you just might need. However, under other circumstances, the **CLOSE** command has no particular effect when **CLOSE**ing direct access files; note that it does not force the **DOS** to **WRITE** the contents of the buffer onto the disk. Under the circumstances this is hardly surprising; the direct access commands that we have looked at in this chapter allow you to completely bypass the **DOS** routines so you can't expect any help from the **DOS** at the last moment!

All the direct access commands require you to use your best programming techniques to keep track of your own data, so use them carefully and wisely. It is very satisfying to know not only what data is on a disk but where it is as well, and to be able to exercise complete control over the system. A wiser person than I said in the 19th century that 'power tends to corrupt, and absolute power corrupts absolutely'. With these direct access commands you have absolute power over your disk drive and you also have the power to absolutely corrupt your disk, so please be careful. Read Chapter 7, about direct access files first, but above all use an unimportant disk while you are experimenting. Good luck!

5

Sequential Files

Over the last two chapters we have examined each of the disk commands individually to show its structure and use. In the three chapters that follow we shall be taking things a step further and looking at these same commands in their natural environment: controlling and manipulating data. We shall be referring to them as they occur in the demonstration programs but without detailed explanation, because we are now concentrating on the practical application of these commands and the development of file handling techniques. So, if you haven't already done your homework by reading Chapters Three and Four then go to the bottom of the class.

No matter how diverse the applications may be, all computer programs are essentially concerned with the same thing: the manipulation of data. Our problem, as mere mortal programmers and operators, is to control the computer's tremendous appetite for number crunching and produce the sort of program that allows us to achieve the following general objectives:—

1. Enter the relevant information efficiently and economically.
2. Process that information in the way that we want.
3. Display the results quickly and clearly.
4. Store important material safely for future re-use.
5. Discard superfluous material.
6. Do all the above as fast as possible.

All these objectives can be achieved with good **BASIC** programming practices. However, the correct choice of file structure can often mean the difference between a successful and

useful program and a pathetically slow one that does nothing more than demonstrate that computers can do clever things – given enough time! The way in which information is stored on the disk is fundamental to the operation of your program, and it is important to be aware of the different file structures, their advantages and their disadvantages, and how to make the best choice for your purposes.

In computer terms a file is any collection of related data. The disk drive stores the contents of a file on a disk, though not necessarily all in the same physical place, and gives it a filename so that it can be found again later. In this sense a disk is rather like a traditional filing cabinet; another collection of related data. This is an analogy that we shall be coming back to because it is a useful way of illustrating both the potential and the limitations of disk storage. There are five different types of file structure that you can use with your Commodore disk drive:

1. **SEQ**quential
2. **US**eR
3. **DIR**ect access
4. **REL**ative
5. **PR**oGram

Let us look first of all at the easiest and most often used: the sequential file.

HOW THEY WORK

The sequential file, or more properly the sequential access file, is the most elementary file structure available on a disk drive. It is also the only type that can be used with a cassette recorder.

Its fundamental principle is that the data is stored literally in a single sequence on the storage medium. When the computer (via your program) **WRITEs** a file to the disk it does so by **WRITEing** the bytes sequentially one after another, to form a complete string of data. To **READ** the data back again it has to be accessed in the same order until the required information is found. An obvious

example is the **LOADing** of programs from cassette tape. There is no other way to access a program in the middle of a collection than by **READING** in everything from the beginning of the tape until the appropriate point is reached. You just cannot jump into the middle of a file to pick out the particular piece you are interested in, you must always start at the beginning and **READ** through, byte by byte, to the end.

An analogy can be drawn between this type of storage and a particularly haphazard method that you might use of keeping a record of all of your friends' names and addresses. The best method of recording the data on paper would be by keeping the names in alphabetical order. However, suppose that you were too lazy, and wrote them down in any old order, just adding new ones at the end of your growing database, since this is what you are creating. There would be no logical order to the records, so when you tried to find an address or name you would have no option but to start at the very beginning and read every entry until you either found the right one or came to the end. A sequential file, on disk or cassette; operates in exactly the same way; you have to **READ** or **WRITE** the whole file at a time and there is no means of random selection.

As with most things, there are advantages and disadvantages with using sequential files. Let's look at the benefits first.

ADVANTAGES

1. The sequential file is by far the easiest type to use. Its simplicity can be a great boon when you are writing small programs, where it would not be worthwhile to use more sophisticated and more difficult options. For simple everyday data storage you should always consider using sequential files first.
2. A sequential file can store variable length records economically without any special preparation. Because of the simplicity of the system this is often not appreciated. It means that data can be stored on disk or cassette efficiently and compactly, with each record beginning immediately after the other and no space wasted in allowing for extra long records.

3. Although **READ**ing a file into memory is very slow, once it is there the speed of access to the data is actually very fast. When you have a program that makes extensive use of its data it would be a sensible choice to use a sequential file, and you will probably find that the overall speed of operation of your program will be greater than with any other file-type.

DISADVANTAGES

1. Because you must **READ** an entire file into memory before processing, the first problem is one of memory size. This is not too bad when using a Commodore 64 with its spacious 38K of user memory, but it can be an acute problem if you have a standard VIC-20 with only 3.5K available to the user. In either case, the size of your database is effectively limited to the size of the available memory in your computer. However, this problem can be bypassed quite easily by **READ**ing the data in but not storing all of it, and letting each new lot of information displace the last.

2. You will then come up against the second disadvantage: speed of access from disk. To **READ** 30K of data from a disk will take a minimum of 2-3 minutes, longer if you are manipulating it at the same time. Using a cassette recorder the same operation takes about 15 times longer, and can take anything up to an hour.

USING SEQUENTIAL FILES.

The whole basis of access to a sequential file is the filename. As we know, when you first create a file you must give it a name that is unique on that disk. It is possible to use the same name for files on different disks but this could still lead to confusion and it is not advisable. The filename can be 16 characters or less, consisting of any character that can be typed in from the keyboard (except the wild cards * and ? for obvious reasons).

A sequential file can be **OPEN**ed either for **WRITE**ing or for **READ**ing, but not for both at the same time. (Relative files, which we shall be looking at in the next chapter, are more versatile and

can be **OPEN**ed for **READ**ing and **WRITE**ing simultaneously, but sequential files are limited to one or the other at a time.) Since we can't read data until it has been written onto the disk we shall look first at how to create a sequential file and write information into it.

A file is created by **OPEN**ing a channel to the disk drive, specifying a new filename and identifying it as a **WRITE** file. For example, if you wanted to **OPEN** a file called 'ADDRESSES' you would use the following line:—

10 OPEN 8,8,8,“ADDRESSES,,S,W”

where **S** is the abbreviation for **Sequential** and **W** is short for **WRITE**. For simplicity we will use **8** for the file, device, and channel numbers.

The **OPEN** statement creates the file but does not actually write anything into it. To do this there is only one command that you can use: **PRINT#**. It is quite a simple operation and the only points to watch for are to be sure that you **PRINT#** data to the correct file, and also that it is done in such a way that it can easily be **INPUT#**ed again at a later time. **PRINT#** must always be followed by a logical file number. When you **OPEN** a file for use, you assign it a temporary number so that future references to it can be made simply by using that file-number. In the example above we used file number **8**, so to write data to that file we just use **PRINT#8**. You can choose any file number you wish (between 1-127 generally, but see also Chapter Three) but you must always use the same one for **OPEN** and **PRINT#** commands to that file. The same principle applies to reading data with the **INPUT#** and **GET#** statements.

DEMONSTRATION PROGRAM

As an example of a sequential file application, we are going to look at a program to store the names, addresses and telephone numbers of a list of people. They could be friends, customers or what you like, this type of database program is always useful. We shall be using it in the next two chapters as well, to demonstrate to you the practical differences between the various file-types.

First of all, before we start coding let us consider the various tasks that a program like this has to perform. The main function can usually be broken down into logical steps, or subsections, which can then be developed as a series of programming building blocks. This process not only makes the final program easier to read and understand, it is also an aid to efficient programming. Additional modules can easily be added at a later date if you should wish.

Our database program will have six basic tasks and therefore six sections. These are:—

1. An initial facility to allow the user to choose their route through the program, and specify which function they want to use, eg. entering a new address. This is best presented as a multiple-choice menu of options.
2. A means of displaying the information held in the database, when called up selectively by the user.
3. A facility for entering new information so that the database can be kept up to date.
4. A routine to read the file from disk into memory, where it can be manipulated by the user to give the information that they want.
5. A means of writing the data back to the disk.
6. A routine that **ENDs** the program properly, making all the necessary safety checks to minimise the risk of losing data.

When producing a program such as this it is always worthwhile laying out the basic structure before attempting detailed coding. In this example, for instance, we shall be using six blocks, so we begin the program by setting out the line numbering like this:—

```

10 REM      *****
12 REM      * SEQUENTIAL FILE *
14 REM      *   DEMO PROG 1   *
16 REM      *****
100 REM     *****
102 REM     *                      *
104 REM     *   MAIN MENU      *
106 REM     *                      *
108 REM     *****

```

```

1000 REM *****
1002 REM *
1004 REM *   FIND ADDRESS   *
1006 REM *
1008 REM *****
2000 REM *****
2002 REM *
2004 REM *   ADD NEW DATA   *
2006 REM *
2008 REM *****
3000 REM *****
3002 REM *
3004 REM *   READ FILE FROM   *
3006 REM *   THE DISK       *
3008 REM *
3010 REM *****
4000 REM *****
4002 REM *
4004 REM *   WRITE FILE TO    *
4006 REM *   THE DISK       *
4008 REM *
4010 REM *****
5000 REM *****
5002 REM *
5004 REM *   END OF PROGRAM   *
5006 REM *
5008 REM *****

```

That's an awful lot of **REMs** and not much else, isn't it? Now we can add some meat to the bare bones. We are going to be storing and recalling names and addresses which we shall store in the computer's memory as a series of string arrays. Before using an array it has to be **DIMensioned**: line 30 **OPENS** the command channel and lines 40 and 50 **DIMension** our arrays. Let's make provisions for up to 100 entries:—

```

30 OPEN 15,8,15
40 DIM NAME$(100),FIRSTNAME$(100),HOUSE$(
  (100),STREET$(100)
50 DIM CITY$(100),COUNTY$(100),PCODE$(10
  0),TELE$(100)

```

MAIN MENU

Now that is out of the way we can get down to the first major routine of the program, the main menu. You don't have to use this approach, of course, but offering the operator a simple choice from a display of options is a sensible and 'user-friendly' way of controlling the flow of a program. In our case we are offering four options and using the function keys to make the choice:—

```

110 PRINT CHR$(147); "                                MAIN
MENU"
120 PRINT:PRINT"  THIS PROGRAM ALLOWS YO
U TO SAVE THE      NAMES, ADDRESSES ";
130 PRINT"AND TELEPHONE NUMBERS      FOR U
P TO 100 PEOPLE"
150 PRINT
160 PRINT" YOU ARE ABLE TO :-"
170 PRINT:PRINT"      F1. FIND AN ADDRESS
"
180 PRINT:PRINT"      F3. ENTER A NEW ADD
RESS"
190 PRINT:PRINT"      F5. SAVE NEW DATA
"
195 PRINT:PRINT"      F7. END THE PROGRAM
"
200 GET A$: IF A$<>" THEN GOTO 200
210 PRINT:PRINT CHR$(29);CHR$(18); "    MA
KE YOUR CHOICE BY FUNCTION KEY    "
220 GET A$: IF A$<CHR$(13) OR A$>CHR$(1
36) THEN GOTO 220
225 A=ASC(A$+CHR$(0))
230 ON (A-132) GOSUB 1000,2000,4000,5000
240 GOTO 100

```

The routine is mostly **PRINT** statements until we reach line 200. This checks that no keys are being pressed; if one is then the line loops around itself until the key is released. This type of action is very useful in programs that are liable to be used by inexperienced operators because it prevents random key-pressing from interfering with the smooth flow of the program. Lines 220-230 control the operation of the program. 220 **GETs** a character from the keyboard and checks whether it is one of the function keys. Assuming that it is,

line 225 sets the variable **A** equal to the **ASCII** code of the key that is being pressed. You may not have come across the **ON...GOSUB** construct of line 230 before, but it allows the program to branch to one of several different lines depending upon the value of the variable. In this case it is **A** but even a formula could be used. If the value of **A** is **zero** or less then the program will ignore the **ON...GOSUB** completely and carry on to the next line. If **A** has a value of **1** then the first line number in the list will be chosen, a value of **2** will select the second number, and so on. If **A** has an integer value greater than the total number of line numbers in the list then, again, the whole **ON...GOSUB** is ignored and the program proceeds to the next line. Line 230, then, controls the flow of all the program that is to follow.

FIND ADDRESS

This is the section of the program that actually displays the information on the screen. Of course, this routine cannot be used until all the data has been read from the disk and stored into the various arrays. Type in the following:-

```

1040 REC=1
1050 PRINTCHR$(147);      RECORD NO:REC
1055 PRINT$URNAME        ;NAME$(REC)
1060 PRINT$FORENAME      ;FIRSTNAME$(REC)
1070 PRINT$HOUSE NO.     ;HOUSE$(REC)
1080 PRINT$STREET        ;STREET$(REC)
1090 PRINT$TOWN          ;CITY$(REC)
1100 PRINT$COUNTY      ;COUNTY$(REC)
1110 PRINT$POST CODE    ;PCODE$(REC)
1120 PRINT$TEL NO.      ;TELE$(REC)
1130 WAIT 197,64:WAIT 197,191
1140 GETA$:IF A$=CHR$(133) AND REC>1 THE
N REC=REC-1
1150 IF A$=CHR$(134) AND REC<N THEN REC=
REC+1
1160 IF A$=CHR$(136) THEN RETURN
1999 GOTO 1050

```

Whenever this routine is called, the array data is **PRINTed** onto the screen by lines 1055 to 1120, starting with the first element of each. Pressing function key 3 will allow the next set of addresses to be shown, while function key 1 allows you to read the previous set, if there is one. To exit from the routine press function key 7. The control of this routine is carried out in lines 1140 to 1160, which test which key is pressed, ignoring all but the three mentioned.

ADD NEW DATA

Working logically and following the operator's steps, the next section to add is one that will allow the entry of data from the keyboard. Yes, the hard facts are that if you want to store the details of your 100 favourite friends on a disk database you first have to type it all into the computer!

In our program this routine lies within line numbers 2000-2210:-

```

2010 PRINT CHR$(147)
2020 N=N+1
2050 PRINT"   ADDRESS BOOK                RECO
RD NO.";N
2060 PRINT:PRINT"   PLEASE ENTER THE FOLL
OWING DETAILS"
2070 PRINT:INPUT"SURNAME          ";NAME$(N)
2075 IF NAME$(N)="" THEN NAME$(N)="-"
2080 PRINT:INPUT"FORE NAME        ";FIRSTNAM
E$(N)
2085 IF FIRSTNAME$(N)="" THEN FIRSTNAME$(
N)="-"
2090 PRINT:INPUT"HOUSE NUMBER ";HOUSE$(N)
)

```

```

2095 IF HOUSE$(N)="" THEN HOUSE$(N)="-"
2100 PRINT:INPUT"STREET NAME  ";STREET$(
N)
2105 IF STREET$(N)="" THEN STREET$(N)="-"
"
2110 PRINT:INPUT"CITY          ";CITY$(N)
2115 IF CITY$(N)="" THEN CITY$(N)="-"
2120 PRINT:INPUT"COUNTY        ";COUNTY$(
N)
2125 IF COUNTY$(N)="" THEN COUNTY$(N)="-"
"
2130 PRINT:INPUT"POST CODE     ";PCODE$(N
)
2135 IF PCODE$(N)="" THEN PCODE$(N)="-"
2140 PRINT:INPUT"TELEPHONE     ";TELE$(N)
2145 IF TELE$(N)="" THEN TELE$(N)="-"
2150 PRINT:PRINT CHR$(18);"PRESS SPACE B
AR TO ENTER ANOTHER RECORD ";
2160 PRINT"  OR RETURN TO RETURN TO THE
MAIN MENU  "
2170 GET A$:IF A$(">") THEN GOTO 2170
2180 GET A$
2190 IF A$=CHR$(32) THEN GOTO 2000
2200 IF A$ = CHR$(13) THEN RETURN
2210 GOTO 2180

```

This section asks for the various details associated with each record. The variable **REC** is the record number and this is increased at line 2020 every time we go through this routine. A check is then made to ensure that the 100th record has not yet been reached. The actual information is stored in parallel string arrays whose names have been carefully chosen to give an indication of their contents. The choice of a particular name for a variable is not critical but appropriate names do help to make a program more readable and understandable. Only the first two characters of a name are recognised by the computer, the others are ignored so long as there is not a keyword embedded within it. (See your User Manual for more details.)

All the line numbers ending in 5 in this section perform a check to ascertain whether a carriage return has been entered as data (ie. nothing else has been typed in). If no other information has been entered then that variable is set to a "-". This is done for a particular reason; it is very difficult to record 'nothing' onto a disk. You have to write at least one character in so that there is then something to read back again.

READ FILE FROM DISK AND WRITE FILE TO DISK

Now let's type in the two most important sections of the program, those that read and write the record data to the disk. There aren't too many lines so we can do it in one go:-

```

60 GOSUB 3000
3020 OPEN 8,8,8,"ADDRESSES,S,R"
3030 INPUT#8,N
3040 FOR REC=1 TO N
3050 INPUT#8,NAME$(REC)
3060 INPUT#8,FIRSTNAME$(REC)
3070 INPUT#8,HOUSE$(REC)
3080 INPUT#8,STREET$(REC)
3090 INPUT#8,CITY$(REC)
3100 INPUT#8,COUNTY$(REC)
3110 INPUT#8,PCODE$(REC)
3120 INPUT#8,TELE$(REC)
3130 NEXT REC
3140 CLOSE 8
3999 RETURN

4020 OPEN 9,8,9,"TEMP. ADDRESSES,S,W"
4030 PRINT#9,N
4040 FOR REC=1 TO N
4050 PRINT#9,NAME$(REC)
4060 PRINT#9,FIRSTNAME$(REC)
4070 PRINT#9,HOUSE$(REC)
4080 PRINT#9,STREET$(REC)

```

```

4090 PRINT#9,CITY$(REC)
4100 PRINT#9,COUNTY$(REC)
4110 PRINT#9,PCODE$(REC)
4120 PRINT#9,TELE$(REC)
4130 NEXT REC
4140 CLOSE 9
4150 PRINT#15,"S:ADDRESSES"
4160 PRINT#15,"R:ADDRESSES=TEMP. ADDRESSES"
4999 RETURN

```

The first point to notice is the similarity between the two routines. The section that reads the data from disk does so in exactly the same order as it was originally recorded. If you compare these routines you are sure to see the correlation between them. The main point to notice is that the data must be read from the file in *exactly* the same order as it was written, even though the variable names need not be the same.

Let's look in more detail at the routine that **WRITES** the data into the file, lines 4000 onwards. Line 4020 is the first of importance and this **OPENS** a file. The number (**N**) of records is stored in the file. Then the details of each record are entered within the **FOR...NEXT** loop of lines 4040 to 4130, using individual **PRINT#** statements for each item. When all the records have been saved the file is **CLOSED** in line 4140, ensuring its security. Lines 4150 and 4160 perform a **SCRATCHING** and **RENAMEING** function.

The file is a temporary one, '**TEMP. ADDRESSES**', and we have used this format because it provides greater security for your information. You could use a **WRITE** and **REPLACE** function (ie. **OPEN 9,8,9,"C:ADDRESSES"**) to store the new data but this has a disadvantage in that it is not always totally reliable, as we have mentioned before. By using **RENAME** within our routine instead there is always a secure file of data on the disk, no matter how temperamental the power supply may prove. (For more about

RENAME see Chapter Four.) What we have done in these lines is:—

1. Write all new data to a second, backup file and **CLOSE** it. At this point there are two files on the disk including the first, still under its original name.
2. Remove the original data file by **SCRATCHING** it, in line 4150, leaving us the security of still having one copy of the file left.
3. **RENAME** this file after the original, line 4160. The latest data now becomes the original data of step 1.

When you are writing programs that utilise valuable disk files, the importance of data security cannot be overemphasised. The overall objective of this subroutine is to **PRINT#** into a file so that in the event of an unexpected crash only the minimum amount of information will be lost. Within the section that actually **PRINT#**s the data you can have more or less free choice over which information is stored and in which order, but the cardinal rule remains that data must always be read back in *exactly* the same order as it was written. This should be fairly obvious by now, but when you are writing complicated programs that seem to be producing unexpected results, incorrect reading of data is often to blame!

If you now look at the routine that reads the information back into the computer you should be able to see a clear correlation between the order in which the data is written and read. First the total number of records is read from the file, followed by the **FOR...NEXT** loop which reads the individual records, a total of **N**. Simple, isn't it? Only a minimal amount of forward planning is necessary to use sequential files successfully.

END OF PROGRAM

We have now reached the last routine of our program, which provides an orderly exit. Key in these lines and our demonstration program will be complete:—

```

5010 PRINT CHR$(147)
5020 PRINT" HAVE YOU SAVED YOUR NEW FILE  "
5030 PRINT"  ONTO THE DISK  Y/N?"
5035 WAIT 197,64:WAIT 197,191
5040 GET A$:IF A$="" THEN 5040
5050 IF A$="N" THEN GOSUB 4000:GOTO 5110
5060 IF A$<>"Y" THEN 5040
5070 PRINT:PRINT"      ARE YOU SURE  Y/N?"
5075 WAIT 197,64:WAIT 197,191
5080 GET A$:IF A$="" THEN 5080
5090 IF A$="N" THEN GOSUB 4000:GOTO 50110
5100 IF A$<>"Y" THEN 5080
5110 CLOSE 15: END

```

This section ensures that the program is not **ENDED** without a new file having been recorded first. This particular routine is not totally foolproof but it does offer you a second opportunity to save your new data if you had previously forgotten to do so. You may like to improve upon it, you should know enough **BASIC** to do so. The only lines that may be puzzling are 5035 and 5075. The **WAIT** commands are used to compel the computer to **WAIT**; firstly until no keys are pressed and then again until a key *is* pressed, to prevent any accidental key-presses from interfering with the orderly process of the program!

Our demonstration program is now complete so **SAVE** it first and then play around with it for a while. You will have to enter about 20 names and addresses before the slowness of disk access becomes particularly noticeable. In the next chapters we shall be performing the same task using a relative file and a direct access file structure so that true comparisons can be made between the merits of each type of file structure. Before that, you may like to add one or more of your own routines to this demonstration program. You could, for instance, add something that allows the computer to search the records for a particular surname and then display the details of that person. The more ambitious amongst you might even like to write a routine that sorts all the records into alphabetical order!

USER FILES

Before we finish the subject we must briefly say something about user files. This type of file is also available on the 1541 disk drive, and they can be used interchangeably with sequential files. The two are, in fact, exactly the same type of file.

The availability of user files allows compatibility between your disk drive and earlier versions of the Commodore **DOS**, where only program and user files were supported. In the dark past of microcomputers (circa 1979), these were the only two file-types available: program files literally to store programs and user files to store data for the user. Programs that were written for the earlier types of disk drive should still work with the 1541, thanks to Commodore's long established policy of providing compatibility across their product range wherever possible.

You should be consistent in your use of one or the other type of file, because you cannot change them around within a program. For instance, if you have specified a sequential file in the **OPEN-to-READ** statement you cannot **READ** from it as a user file, or vice versa. Thanks to this flexibility the 1541 can also **READ** disks used by the Commodore range of business machines and is the only home computer kit available in the U.K. able to do so.

SUMMARY

The advantages and disadvantages of using sequential files can be summed up as follows:—

ADVANTAGES

1. Ease of use. When writing your own programs all you have to remember is that the data must be read from the file in the same order that it was written to it.
2. Compact and efficient use of storage space.
3. Fast manipulation of data is possible once it has been read into the computer's memory.
4. Variable length data can be stored without any special preparation.

DISADVANTAGES

1. The file must be written/read from the beginning to the end. It is not possible to alter a small part of the data without rewriting the entire file.
 2. Slow access to data on disk. This disadvantage can often be offset by the fast access once the data is in the memory of the computer.
 3. The size of each data file is more or less limited to the size of the computer's memory. This could be circumvented but the slow access times resulting would be unacceptable.
-

PRINT MODIFIERS, TERMINATORS, DATA STORAGE

When you use the **PRINT** statement to the screen there are two 'print modifier' characters, the semi-colon and the comma. Both are used to prevent the **PRINT** statement from performing its customary carriage return **CHR\$(13)**, to start a new line. The semi-colon causes the next **PRINT** to start **PRINT**ing where the previous one finished, whilst the comma causes the start of the next **PRINT** to be **TAB**bed to the beginning of the next 10-column wide block on the screen. These two characters have exactly the same effect when used with the **PRINT#** command, although the effect is more subtle. The best way to show this is to use a small demonstration program that firstly writes onto the disk and then reads back again, to display the results on the screen.

Type in the following and you will see how the semi-colon modifies the **PRINT#** command when storing two string variables onto a disk file (and also to a cassette file, if that is what you are using):—

```

10 REM *****
12 REM *      DEMO PROG      *
14 REM * PRINT# MODIFIER *
16 REM *****
20 OPEN 8,8,8,"C:DEMO FILE,S,W"
30 PRINT#8,"HELLO","DAVE"
40 CLOSE 8
50 OPEN 8,8,8,"DEMO FILE,S,R"
60 INPUT#8,A$,B$
70 PRINT"      A$ HOLDS ";A$
80 PRINT"B$ HOLDS ";B$
90 CLOSE 8

```

Here we have stored two words into the demo file and, having first **CLOSEd** the file for writing and re**OPENed** it for reading, tried to read the two words back into the variables **A\$** and **B\$**. However, when you **RUN** the program you will see that the two words are read only into **A\$**, **B\$** is left as a null. The reason for this is that the **INPUT#** command will read every character from the specified file until a terminator is reached. Just as with the **INPUT** command, 'the carriage return' and comma are counted as terminators and with the **INPUT#** a semi-colon can also be used.

Now, this might seem a bit confusing at first because there appear to be two different uses for these characters, doesn't there? The distinction is a simple one, if perhaps not immediately obvious. When the comma or semi-colon is used to modify a **PRINT#** statement then it will do just that. When either of these characters is **PRINT#ed** to the file then it will be used as a delimiter for later **INPUT#** commands.

Using the program shown above we can, with a few easy modifications, show the various effects upon data stored in a file, of these characters. When you **RUN** the program as it stands you should see on the screen:—

A\$ HOLDS HELLO

B\$ HOLDS

DAVE

and the data is stored on the disk file thus:—

DATA	H E L L O										D A V E cr eof													
BYTE	0	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	2	2	2	2		
											0	1	2	3	4	5	6	7	8	9	0	1	2	3

The spaces between the two words result from using the comma in line 30. **INPUT#**ing **A\$** reads in both words and the space between them, up to the carriage return character immediately after "**DAVE**". This is very wasteful of disk space since there are 10 spaces between each word. Altering line 30 so that the comma

and this is the most compact and easily readable format to use.

The commands **INPUT#** and **GET#** are used to read data from a (disk) file into the computer's memory and the systems variable **STATUS** can be used to detect the **End-Of-File** marker. **GET#** reads a single character at a time whilst **INPUT#** is used to read entire variables, up to a delimiter. As we have just seen, the carriage return **CHRS(13)** is used as a delimiter, but we can also use a comma or a semi-colon. Unlike with their format or modifier function, when used as delimiters these characters must be **PRINT#**ed to the file in order to be picked up again by **INPUT#**. For instance, changing line 30 in our demonstration program (and deleting line 35) to :-

```
30 PRINT#8, "HELLO",";", "DAVE"
```

will still allow the two words to be **INPUT#**ed separately into **A\$** and **B\$**. They are now on the disk thus:-

DATA	H E L L O , D A V E cr eof																									
BYTE	0	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	2	2	2	2			
											0	1	2	3	4	5	6	7	8	9	0	1	2	3		

Similarly, if you write a semi-colon instead of the comma the effect will be the same. To the **INPUT#** command the three characters, carriage return, comma and semi-colon all have the same purpose and you can use whichever one you wish. The only difference occurs when you use **GET#** to read the file, because this command will read every character, one at a time, including these separators. This makes **GET#** very useful for examining files of unknown format because you can **GET#** every character and examine it individually.

The systems variable **STATUS**, or **ST**, is used by the computer to show the completion **STATUS** of the last Input/Output operation. When using a diskdrive you can use the current **STATUS** to detect the **End Of File** marker (**EOF**). If **ST** has a value of 64 then the end of the file has been reached, otherwise there is still some data to be read. Using our same demo program changing lines 60 to 80

will show how to **GET#** data from a file, using **ST** to detect the end:—

```
60 GET#8,A$:B$=B$+A$
```

```
70 IF ST<>64 THEN GOTO 60
```

```
80 PRINT B$
```

Try **RUN**ning the program and you should see that all of the characters stored in the file are read and printed onto the screen.

6

Relative Access Files

In the last chapter we looked at the structure and uses of the sequential file. The drawbacks of this type, such as slow access and limited file size, are offset to quite a large extent by the advantages of versatility and simplicity. In the chapter after this one we shall be looking by contrast at the direct access file which provides great flexibility but at the cost of simple programming. For now, in this chapter, we shall investigate a practical compromise between the two: the relative access file.

On Commodore disk drives the relative access file is in many ways the most sophisticated of the three structures available. The combination of easy use with powerful and fast access to data makes this the logical choice for most operations, especially large database applications.

So what is a relative access file? The very name of this file type provides the key to its function. Using relative files you can create a file of data that takes the form of a number of records, each one rather like an individual card in a card-index system. You can access any one of these 'cards' or records even though you do not know exactly where that card is in the system. The **DOS** automatically keeps track of the physical location of each record and acts rather like a filing clerk asked to get a record from the cabinet. All you need to know is the *relative* position of each. You could, for instance, create a relative file that contains 100 records. To access, say, the 50th record you would simply have to tell the **DOS** to position the head over the 50 record and it would do so without your knowing exactly where on the disk that record was.

HOW IT WORKS

Before going any further we should first of all consider the structure of a relative file and the meaning of the few new words that we shall be using. A file, as we have said before, is no more than a collection of related data. To use our old example of names and addresses, a file is the structure within which all this data of a similar type is assembled and organised, like an address book. Within this structure there is a clear breakdown into subsections of the information relating to each individual: their name, address and so on. Each subdivision is known as a record and this is the basic building block of the relative file. Breaking the material down even further, each element within the record, such as surname or house number is called a 'field'. Thus a number of fields make up a record and a number of records are required to produce a relative file.

The **DOS** maintains its own housekeeping file containing the location of each record within the main file. Therefore records can be accessed individually, unlike sequential files where only blanket access to data is possible. This selectivity gives rise to a number of advantages but, as with most aspects of computer programming there are also drawbacks to be taken into consideration.

ADVANTAGES

1. The database contained within a relative file always remains on the disk itself. Because access to records is reasonably easy, there is no need to read the entire file into the core memory of the computer. This means that the size of your database is limited not by the amount of memory in your computer but by the much greater capacity of the disk, approximately 176k! Even users of the standard VIC-20, with its 3.5k of user memory, can have access to a massive amount of storage capacity in this way and with good programming can write some very powerful programs.

2. A great number of records can be stored within these files which are limited only by the available disk space and the size of each record. It is possible to have up to 65535 records on one disk, each one quickly and individually accessible.
3. Access to each record is efficient and fast. It is not as fast as having all the data within the computer's memory, but with even a large database the access time is very acceptable.
4. Relative files allow you to both read and write to the **OPEN** file, which contributes to their overall flexibility and speed.
5. You can alter individual records, or even particular fields within a record, independently of the other information in the file. This makes for fast corrections and additions to your database.
6. A unique feature of relative files is that you can easily 'reserve' space on the disk for the entire database that you are planning to create. If, for instance, you wanted to create a file to hold the names and other details of 200 customers you could reserve enough space on the disk to hold this number even though you might only have 15 people at present. This is a very useful feature because it then ensures that the disk you are currently using will not run out of storage space at some point in the future. You can write into your program '**please insert the disk labelled ADDRESSES**', for instance, safe in the knowledge that it will hold all the information you want it to.

For every silver lining there has to be a cloud, so here are the principal disadvantages that you are likely to encounter when using relative files. They are surprisingly few and certainly they do not outweigh the many advantages.

DISADVANTAGES

1. If the total capacity for a very large database is taken up it can make manipulation of the data within the file extremely difficult and slow. For instance, sorting 1000 records into alphabetical order will take a very long time and require some clever programming. As soon as the database becomes

larger than the available computer memory then in a sense the problem becomes academic because there is no other solution.

2. Relative files require a rigidly defined record structure. When you first create the file, you have to define the number of characters that will be contained within each record, up to a maximum of 254 bytes. This can, at times, be extremely limiting.
3. Because of these restrictions relative files are often very wasteful of disk storage. If, for example, you were to specify a record length of 200 bytes then all 200 bytes would be reserved even though you might only store 20 bytes of data within a particular record. The unused space in such cases is wasted and cannot be used for any other purpose.

RELATIVE ACCESS FILES IN THEORY

More than any other operation of the disk drive, relative access files cause great confusion whenever they are mentioned. So before we go into the mechanics of using them let's investigate in plain language how they work.

Before you can actually use a relative file, you must first create it. You do this in a similar manner to creating a sequential file, by using the **OPEN** command. When you **OPEN** the file you must also specify the length that each record will be, up to a maximum of 254 characters. At this point the **DOS** sets six blocks or sectors aside for its own use. These *side sectors* as they are known are used by the **DOS**, in a way completely transparent to the user, to keep its own record of where on the disk the data is being stored. In subsequent operations, whenever you add to the data in your file the **DOS** will automatically update this list. The six side sectors can hold a list of up to 720 pointers in total. It is important to remember that these pointers do *not* point directly to the beginning of each record. They do, in fact, point to the 720 blocks or less that are being used to store the data. This is the first limitation to the file size: there are only 664 blocks free for use on a single sided, single density disk, such as that used by the CBM 1541 disk drive, not 720. Of course, a partially filled disk will have

room for an even smaller relative file. There are two more limitations: 65536 is the maximum number of records that you can have in any one file (numbered from 0 to 65535), and each record must be less than or equal to 254 bytes, ie. a single block. Within these three restrictions, you can create a file to suit your own requirements.

USING RELATIVE FILES

As an example of using relative files we shall create a similar program to the address database that we used in the last chapter. There were eight fields for each record and we shall use the same number here. With a sequential file, each record can be of variable length but relative files require that the maximum length of each record be set initially. Within individual records each field can be a different length so long as the total size is less than the record maximum. For example, you could have:

FIELD NAME	LENGTH
SURNAME	15
FIRST NAME	15
HOUSE NUMBER	5
STREET NAME	25
TOWN NAME	20
COUNTY NAME	20
POSTCODE	10
TELEPHONE NUMBER	15
TOTAL RECORD SIZE	125

The field sizes shown above are for guidance only, although once specified no single field should exceed its nominal size and in total they cannot exceed the 125 bytes that we are going to use as our record size. When you are setting up your own file you should always remember that a delimiter character must be included in the calculation for each field. This is the carriage return or **comma** character that allows **INPUT#** to read individual fields and

variables easily. Remember that the **semi-colon** does not act as a delimiter when using relative access files.

125 bytes is slightly under half a block of disk storage, so, as a rough calculation we will be able to store around 1300 records when using a blank disk. A freshly formatted disk has 664 blocks free and the file requires six side sectors for its own use, leaving 658 for the data. The total number of records is, therefore:—

$$\mathbf{658\ BLOCKS\ *\ 254\ BYTES\ /\ 125\ BYTES\ =\ 1337\ RECORDS.}$$

That should be plenty. If you have more than 1300 friends then you will probably be too busy going out enjoying yourself to find the time to keep a database up to date! On the business side, if you have more than 1300 customers then you should be using a more powerful (and expensive) professional computer. No doubt there will be circumstances where you want to keep a record of large numbers of individual items, be it VAT invoices and receipts or a catalogue of your stamp collection, and the relative access file demonstration program in this chapter should provide you with sufficient ammunition to allow you to write your own custom program. Use the sections that follow within your own programs and explore the fascination of programming as a hobby by adding routines that you write yourself to provide that 'custom-made' feel to your program.

This demonstration database program is somewhat different from the one using a sequential file, although the structure is generally similar. We now have ten sections to the program, which are:—

1. Initial set-up
2. Main menu
3. Find an address
4. Add new data
5. Read a record from the file
6. Write a record to the file
7. End the program
8. Read the error channel
9. Create the relative file
10. Set the record pointer

As always, the best thing to do first is to set out the program into sections with **REM** statements, so that it is easily understandable.

```

10 REM *****
12 REM *   RELATIVE FILE   *
14 REM *   DEMO PROG      *
16 REM *****
100 REM *****
102 REM *                   *
104 REM *   MAIN MENU      *
106 REM *                   *
108 REM *****
1000 REM *****
1002 REM *                   *
1004 REM * FIND ADDRESS    *
1006 REM *                   *
1008 REM *****
2000 REM *****
2002 REM *                   *
2004 REM * ADD NEW DATA   *
2006 REM *                   *
2008 REM *****
3000 REM *****
3002 REM *                   *
3004 REM * READ A RECORD   *
3006 REM * FROM THE DISK  *
3008 REM *                   *
3010 REM *****
4000 REM *****
4002 REM *                   *
4004 REM * WRITE A RECORD *
4006 REM * TO THE DISK   *
4008 REM *                   *
4010 REM *****
5000 REM *****
5002 REM *                   *
5004 REM *   END PROGRAM   *
5006 REM *                   *
5008 REM *****

```

```

6000 REM *****
6002 REM *
6004 REM * READ ERROR *
6006 REM * CHANNEL *
6008 REM *
6010 REM *****
7000 REM *****
7002 REM *
7004 REM * CREATE THE *
7006 REM * RELATIVE FILE *
7008 REM *
7010 REM *****
8000 REM *****
8002 REM *
8004 REM * SET THE *
8006 REM * RECORD POINTER*
8008 REM *
8010 REM *****
READY.

```

The initializing process for this program is much simpler than that for use with sequential files. There is no need to **DIM**ension any array because it is not necessary to read the data from disk into an array. This is because each record can be accessed repeatedly and independently of the others, resulting in huge savings in memory. In fact, the only initializing we need to carry out is to **OPEN** a command channel to the disk drive in line 30 by adding these two lines:—

```

30 OPEN 15,8,15
40 GOSUB 7000

```

CREATE THE RELATIVE FILE AND READ THE ERROR CHANNEL.

The next piece of program to be written is the routine that creates the file, lines 7000 onwards:—

```

7020 OPEN8,8,8,"REL.ADDRESSES"
7030 GOSUB 6000
7040 IF EN=62 THEN CLOSE8:GOSUB 7200:GOT
O 7000
7050 X=101:P0=1:GOSUB 8000
7060 INPUT#8,CURRENTREC$
7070 CURRENTREC=VAL(CURRENTREC$)
7120 RETURN
7200 OPEN 8,8,8,"REL.ADDRESSES,L,"+CHR$(
125)
7210 X=101:P0=1:GOSUB 8000
7220 GOSUB 6000
7230 IF EN=52 THEN PRINT CHR$(147);"SORR
Y, THERE IS NOT ENOUGH DISK SPACE"
7240 IF EN=52 THEN CLOSE8: END
7250 PRINT#8,"0"
7260 CLOSE 8
7270 RETURN

```

At this point we must add the routine that reads the error channel:—

```

6020 INPUT#15,EN,EM$,ET,ES

```

```

6030 RETURN

```

So, from line 40 the program now **GOSUBs** to the 'create a file' routine. This routine has two functions. Looking first at lines 7000 to 7120, we can see that the file is **OPENed** in line 7020. A call to the error routine, in line 7030, is then followed by a check to determine whether the file exists. Error number 62 will be returned if the file has not yet been created, although of course this will only occur on the first **RUNning** of the program. Lines 7200 to 7270 are the routine that creates the relative access file initially and we shall deal with this shortly. Assuming that the file is on the disk the only task remaining is to read in the value for the number of records in the file, a job performed by lines 7050 and 7060. We have already decided that we shall store the details for up to 100 people, so the number of records that have actually been stored can safely be held as record number 101. Lines 7120 then **RETURNs** us back to the main program.

When the program is **RUN** for the first time a file has yet to be created and this is done by lines 7200 to 7270. Line 7200 **OPENS** the file and specifies that each record shall have a length of 125 characters. We can then force the **DOS** to allocate sufficient space

for the file by writing to the 101st record. We do this with a **zero** because there are no records in the file at present. A check to the error channel beforehand, lines 7220 to 7240, determines whether there is sufficient space available on the disk and if this is the case then the **zero** is written to the 101st record and the program flow **RETURNS** to the calling routine to allow the file to be **OPENed** properly. If you get an error message 52, however, you are short of space, in which case the program will **END** and allow you to find another disk before **reRUNning**. If there is sufficient space then the zero is written to the 101st record and the program flow then **RETURNS** to the calling routine, to allow the file to be properly **OPENed**.

MAIN MENU

So, back to the main program. The next section is the main menu, which is added so that the program is easy for the operator to use. There is **BASICally** (pardon the pun) nothing new in this section, so just key in the following and we shall move straight on to the routines that read and write data to the disk file.

```

110 PRINTCHR$(147);"                                MAIN M
ENU"
120 PRINT:PRINT" THIS PROGRAM ALLOWS YOU
    TO SAVE THE      NAMES, ADDRESSES ";
130 PRINT"AND TELEPHONE NUMBERS      FOR U
P TO 100 PEOPLE"
150 PRINT
160 PRINT" YOU ARE ABLE TO :-"
170 PRINT:PRINT"      F1. FIND AN ADDRESS
"
180 PRINT:PRINT"      F3. ENTER A NEW ADD
RESS"
190 PRINT:PRINT"      F5.
195 PRINT:PRINT"      F7. END THE PROGRAM
"
200 GET A$:IF A$<>" THEN GOTO 200
210 PRINT:PRINT CHR$(29);CHR$(18);"    MA
KE YOUR CHOICE BY FUNCTION KEY  "
220 GET A$: IF A$<CHR$(133) OR A$>CHR$(1
36) THEN GOTO 220

```

```

225 A=ASC(A$+CHR$(0))
230 ON (A-132) GOSUB 1000,2000,3000,4000,5000
240 GOTO 100

```

FIND ADDRESS

The next routine, lines 1030 to 1999, is concerned with **PRINT**ing the data for the records that have been found as a result of a search using the subroutine in lines 3000 and onwards. It also relates to the section of code between lines 2010 and 2220 which allows the user to enter data via the keyboard. Lines 1150-1170 determine the action of this routine: pressing function key 3 will access the next record from the disk, function key 1 will access the previous record and function key 7 will allow the user to exit to the main menu. In the same way, lines 2160 to 2220 allow the user either to enter the next set of data or **RETURN** to the menu. Key in the following, but don't **RUN** the program yet:-

```

1030 REC=1:PO=1
1040 X=REC:GOSUB 3000
1050 PRINTCHR$(147); "      RECORD NO: "REC
1055 PRINT"SURNAME      "NAME$
1060 PRINT"FORE NAME    "FIRSTNAME$
1070 PRINT"HOUSE NO.    "HOUSE$
1080 PRINT"STREET       "STREET$
1100 PRINT"TOWN/CITY    "CITY$
1110 PRINT"COUNTY       "COUNTY$
1120 PRINT"POST CODE    "PCODE$
1130 PRINT"TEL NO.      "TELE$
1140 WAIT 197,64:WAIT 197,191
1150 GETA$:IF A$=CHR$(133) AND REC>1 THE
N REC=REC-1
1160 IF A$=CHR$(134) AND REC<CURRENTREC
THEN REC=REC+1
1170 IF A$=CHR$(136) THEN RETURN
1999 GOTO 1040

```

```
2010 PRINT CHR$(147)
2020 CURRENTREC=CURRENTREC+1
2050 PRINT"  ADDRESS BOOK                      RECO
RD NO.";CURRENTREC
2060 PRINT:PRINT"  PLEASE ENTER THE FOLL
OWING DETAILS"
2070 PRINT:INPUT"SURNAME          ";NAME$
2075 IF NAME$="" THEN NAME$="-"
2080 PRINT:INPUT"FORE NAME        ";FIRSTNAM
E$
2085 IF FIRSTNAME$="" THEN FIRSTNAME$="-"
"
2090 PRINT:INPUT"HOUSE NUMBER ";HOUSE$
2095 IF HOUSE$="" THEN HOUSE$="-"
2100 PRINT:INPUT"STREET NAME    ";STREET$
2105 IF STREET$="" THEN STREET$="-"
2110 PRINT:INPUT"TOWN/CITY      ";CITY$
2115 IF CITY$="" THEN CITY$="-"
2120 PRINT:INPUT"COUNTY         ";COUNTY$
2125 IF COUNTY$="" THEN COUNTY$="-"
2130 PRINT:INPUT"POST CODE      ";PCODE$
2135 IF PCODE$="" THEN PCODE$="-"
2140 PRINT:INPUT"TELEPHONE      ";TELE$
2145 IF TELE$="" THEN TELE$="-"
2150 X=CURRENTREC : GOSUB 4000
2160 PRINT:PRINT CHR$(18);"PRESS SPACE B
AR TO ENTER ANOTHER RECORD ";
2170 PRINT"  OR RETURN TO RETURN TO THE
MAIN MENU  "
2180 GET A$:IF A$(">") THEN GOTO 2180
2190 GET A$
2200 IF A$=CHR$(32) THEN GOTO 2000
2210 IF A$=CHR$(13) THEN RETURN
2220 GOTO 2190
```

READ AND WRITE A RECORD TO DISK

Lines 3000 to 3999 make up the subroutine to read, **OR INPUT#**, a single record from the file and this section is called from line 1040:—

```

3015 PO=1:GOSUB 8000
3020 INPUT#8,NAME$
3025 PO=16:GOSUB 8000
3030 INPUT#8,FIRSTNAME$
3035 PO=31:GOSUB 8000
3040 INPUT#8,HOUSE$
3045 PO=36:GOSUB 8000
3050 INPUT#8,STREET$
3055 PO=61:GOSUB 8000
3060 INPUT#8,CITY$
3065 PO=81:GOSUB 8000
3070 INPUT#8,COUNTY$
3075 PO=101:GOSUB 8000
3080 INPUT#8,PCODE$
3085 PO=111:GOSUB 8000
3090 INPUT#8,TELE$
3999 RETURN

```

Before we explain further you should also key in the parallel routine that **PRINTs** the data into each record. Notice first the close similarity between these two, the only significant difference being that **INPUT#** statements in the first routine are replaced by **PRINT#** statements in the second. In fact it would be much quicker and easier carefully to change the lines that you have just typed in than to key in this whole new section from scratch:

```

4015 PO=1:GOSUB 8000
4020 PRINT#8,NAME$
4025 PO=16:GOSUB 8000
4030 PRINT#8,FIRSTNAME$
4035 PO=31:GOSUB 8000
4040 PRINT#8,HOUSE$
4045 PO=36:GOSUB 8000
4050 PRINT#8,STREET$
4055 PO=61:GOSUB 8000
4060 PRINT#8,CITY$
4065 PO=81:GOSUB 8000

```

```

4070 PRINT#8,COUNTY$
4075 PO=101:GOSUB 8000
4080 PRINT#8,PCODE$
4085 PO=111:GOSUB 8000
4090 PRINT#8,TELE$
4999 RETURN

```

SET THE RECORD POINTER AND END THE PROGRAM

Two more routines to go and then the demonstration program is complete! Lines 8000 to 8060 are possibly the most important in the program because they set the record pointer to the correct position within the relative file. This subroutine is dual purpose in that it is used for both reading from and writing to the file.

```

8030 XH=INT(X/256)
8040 XL=X-XH*256
8050 PRINT#15,"P"CHR$(8)CHR$(XL)CHR$(XH)
CHR$(PO)
8060 RETURN

```

The action is performed by line 8050, where the variables **XL** and **XH** are the low byte and high byte respectively of the record number. These values are calculated in lines 8030 and 8040. The **CHR\$(PO)** positions the pointer to a particular character position within the record, in this case to the start position for each field. When you look at the routine for writing data to the file, (lines 4000-4999), you can see that before each field is **PRINT#**ed, **PO** is set to the relevant position and then a call is made (**GOSUB 8000**) to set the pointer. Again, working back to the original calling routine where the data is entered from the keyboard, (lines 2000-2220), you can see that in line 2150 the variable **X** is set to the current record number, **N**, before **GOSUB**ing to 4000. Thus, when **PRINT#**ing data to the file the record number is held in **X** and the position within the record is held in **PO**. It is then simply a case of setting the pointer before writing the data. We leave it to you to follow through the process of **INPUT#**ing data from the file. It is exactly the same: first set the pointer to the required position within the file and then read the data into a variable.

The final section of code is the '**end-of-program**' routine which in the case of relative files is much simpler than the equivalent routine for sequential files:—

```
5010 PRINT CHR$(147)
5020 X=101:PO=1:GOSUB 8000
5030 PRINT#8,CURRENTREC
5040 CLOSE 8: CLOSE 15
5050 END
```

Lines 5000-5050 store the current value of **N**, (the number of records filled), in record number 101 before **CLOSE**ing all files and **ENDING** the program.

Now that you have the complete program typed in you should **SAVE** it onto disk before you try to **RUN** it, just in case you have made any mistakes that crash it. When you do **RUN** it you will be able to store and recall details for up to 100 people. This is only a demonstration program, so the facilities on offer are rudimentary, but it can be used to form the basis for your own database program. You could, for instance, add a routine that allows you to specify a particular record and view the contents. You could even write a routine that searches through all the records looking for a particular field-match, displaying only those that meet the given requirements.

Hopefully this chapter has demonstrated to you that the relative file structure available on Commodore disk drive systems can be powerful and versatile yet at the same time easy to use. For many database applications this file type should be your first choice, especially where the data to be recorded is of a standardised nature and needs to be stored in large amounts. The single overwhelming disadvantage of relative files is their limited record size of 254 bytes. By careful planning and good programming even this can be overcome, however. The trick is to create two or more files and use them in parallel; thus if you wanted to store 500 bytes of information per record you could **OPEN** two files, writing the first 250 bytes to the first file and the rest to the second. But that is another aspect of file handling and we shall not enter into the serene world of multi-files in this book! The important thing to remember when writing programs that use relative files is to plan first and code second. That way you will never lose any of your precious data — well, almost never!

7

Direct Access Files

The availability of direct access files on the Commodore disk drive enables you, the user, to control the reading and writing of data directly onto disk. Using them you will be able to access any block on the disk, including the systems track 18. This type of file structure is often erroneously referred to as 'random access'. The word *random* implies a total lack of control, but all files must have some form of organisation and need to be accessed in an orderly way. To access files truly at random would be unhelpful, to say the least, because you would have no control over the information transferred. However, to be able to access the individual bytes of storage space on a disk directly, rather than randomly, can be extremely useful in many circumstances, and this is precisely what direct access files allow.

On the Commodore drive disk, the commands that control this direct access to data also allow you to bypass completely the built-in programs and routines of the **DOS**. In certain circumstances this freedom can be used to good advantage, but it also means, that the responsibility for the safety of your data lies entirely with you the programmer, and you can expect no assistance from the **DOS**. For this reason alone, direct access files should be the last type to be considered when you are planning out your program, and for general data storage either sequential or relative files should be chosen in preference.

The structure of the direct access commands allows such a high degree of flexibility that it is possible to simulate through program control either of the other two structures. However, the end result of such a complicated simulation will seldom be worth the effort, either in terms of the elegance of the program or the speed of its execution. Later in this chapter we shall be looking at a direct

access version of our database program that we used in the previous two chapters. It must be stressed here that it is only a demonstration version, and we cannot recommend that you use it for serious database purposes because of the inherent possibilities of data corruption.

ADVANTAGES

1. You are able to bypass the usual **DOS** command structure.
2. In so doing, any application can be programmed.
3. Machine code routines and programs can be stored on the disk for recall at a later time. They can even be used to modify or improve the existing set of **DOS** routines if you are a sufficiently experienced programmer.

DISADVANTAGES

1. All the work of data storage and retrieval must be accomplished purely by your programming, with no help from the **DOS**.
2. The risk of data corruption is much higher than with either sequential or relative files.
3. Perhaps most importantly, disk blocks that are allocated for use by direct access file are completely unprotected.

Whilst they will not be overwritten by normal file handling techniques, if you were to **VALIDATE** the disk you would immediately wipe out all your direct access files.

DIRECT ACCESS COMMANDS

There is a set of seven commands built into the disk drive operating system which allows you to exercise complete control over the positioning of data on a disk. These are **BLOCK-ALLOCATE**, **BLOCK-FREE** etc. which are all discussed in detail in Chapter 4 to where you should refer back for individual explanations. Let us examine here the way that these commands complement each other when used together.

When we were building up a data file in the previous two chapters, the work of actually storing the data on disk was carried out by the **DOS**, without us needing to know the exact physical location of the data because the DOS could always be relied upon to find the required snippet of information on request. When using the **direct access** structure you, the programmer, must take over the role of the **DOS**. So, your first task is to identify a free block on the disk and tell the **BAM** to reserve it for use.

The **BLOCK-ALLOCATE** command is used to mark a block as being in use, and the opposite action is performed by **BLOCK-FREE**. You must, of course, always allocate the exact blocks that you are using. If you don't, any subsequent **SAVE**ing operation may overwrite your data, thus corrupting your file. You use the commands **BLOCK-READ** and **BLOCK-WRITE** to do just that: **READ** and **WRITE** data to a block. The **BUFFER-POINTER** command can be used to set the position within a block, so that individual bytes can be influenced. Finally, **USER1** and **USER2** are special versions of the **BLOCK-READ** and **BLOCK-WRITE** commands and can be used to good effect in certain circumstances.

Now let us look at our familiar database of names and addresses, this time in the format of a direct access file. As always we should first of all set out the routines that are to be used, by laying down a core of **REMs** for each section:—

```

10 REM *****
12 REM *   DIRECT ACCESS   *
14 REM *         FILE      *
16 REM *   DEMO PROG      *
18 REM *****
100 REM *****
102 REM *                  *
104 REM *   MAIN MENU      *
106 REM *                  *
108 REM *****
1000 REM *****
1002 REM *                  *
1004 REM * FIND ADDRESS    *
1006 REM *                  *
1008 REM *****

```

```

2000 REM *****
2002 REM *
2004 REM * ADD NEW DATA *
2006 REM *
2008 REM *****
3000 REM *****
3002 REM *
3004 REM * READ A RECORD *
3006 REM * FROM THE DISK *
3008 REM *
3010 REM *****
4000 REM *****
4002 REM *
4004 REM * WRITE A RECORD*
4006 REM * TO THE DISK *
4008 REM *
4010 REM *****
5000 REM *****
5002 REM *
5004 REM * END PROGRAM *
5006 REM *
5008 REM *****
6000 REM *****
6002 REM *
6004 REM * READ ERROR *
6006 REM * CHANNEL *
6008 REM *
6010 REM *****
7000 REM *****
7002 REM *
7004 REM * READ IN THE *
7006 REM * SEQUENTIAL *
7008 REM * FILE *
7010 REM * FROM DISK *
7012 REM *
7014 REM *****
8000 REM *****
8002 REM *
8004 REM * WRITE THE *
8006 REM * SEQUENTIAL *
8008 REM * FILE *
8010 REM * TO DISK *
8012 REM *
8014 REM *****

```

```

9000 REM *****
9002 REM *
9004 REM * CREATE THE
9006 REM * SEQUENTIAL
9008 REM * FILE
9010 REM *
9012 REM *****
10000 REM *****
10002 REM *
10004 REM * ALLOCATE THE
10006 REM * NEXT FREE
10008 REM * BLOCK
10010 REM *
10012 REM *****
10500 REM *****
10502 REM *
10504 REM * FORCE END OF
10506 REM * PROGRAM IF
10508 REM * DISK IS FULL
10510 REM *
10512 REM *****

```

Before we get down to the actual coding, it is worth going quickly through the sections marked out for this program. Most of them are quite familiar by now, aren't they? For instance, we have to have a main menu to allow the operator to control the flow of the program, there has to be a routine to find an address from within the database and there also has to be a means of adding new data to the file. Similarly, we need routines to **READ** and **WRITE** the data to the disk when required, a means of **ENDING** the program in an orderly fashion and some lines to **READ** the error channel. All these subroutines were equally necessary with the relative files in the last chapter and we are going to simulate the same structure in this program. There are some routines added from lines 7000 onwards, which will be explained in due course.

INITIALIZATION

The first operation that has to be effected is to **OPEN** the command channel and **DIM**ension any arrays that are to be used within the program. As you can see from the next listing, we are going to use two arrays: one to hold the track address of each record on disk, and a second to store the sector address of each record.

Type in the following:—

```
30 DIM TRACK(100)
```

```
40 DIM SECT(100)
```

```
50 OPEN 15,8,15
```

```
60 OPEN 5,8,5,"#"
```

```
70 GOSUB 7000
```

In this example we will be using an entire sector (254 bytes) to store each record. If you were to write a more sophisticated version you might choose to store several records within each block. In this case you would have to keep a parallel array to store the buffer pointer position for every record, so that the pointer can be correctly positioned in each individual record. This would take the form of, for instance:—

```
45 DIM POSITION(100)
```

You should note however that, unlike relative files where records can be of *any* pre-specified length, with direct access files the entire record or records must fit within each individual block. You cannot have a record that overlaps blocks on the disk except by dint of extremely difficult programming. Therefore, you have certain limitations as to the record length. One record per block will allow up to 254 bytes per record including carriage returns. With two records per block you are allowed a maximum of 127 bytes per record, and so on.

MAIN MENU

```

30 DIM TRACK(100)
40 DIM SECT(100)
50 OPEN 15,8,15
60 OPEN 5,8,5,"#"
70 GOSUB7000

110 PRINTCHR$(147);"                                MAIN M
ENU"
120 PRINT:PRINT" THIS PROGRAM ALLOWS YOU
    TO SAVE THE      NAMES, ADDRESSES ";
130 PRINT"AND TELEPHONE NUMBERS      FOR U
P TO 100 PEOPLE "
150 PRINT
160 PRINT" YOU ARE ABLE TO :-"
170 PRINT:PRINT"      F1. FIND AN ADDRESS
"
180 PRINT:PRINT"      F3. ENTER A NEW ADD
RESS"
190 PRINT:PRINT"      F5.
195 PRINT:PRINT"      F7. END THE PROGRAM
"

200 GET A$:IF A$(>)" THEN GOTO 200
210 PRINT:PRINT CHR$(29);CHR$(18);"    MA
KE YOUR CHOICE BY FUNCTION KEY    "
220 GET A$: IF A$(CHR$(133) OR A$>CHR$(1
36) THEN GOTO 220
225 A=ASC(A$+CHR$(0))
230 ON (A-132) GOSUB 1000,2000,3999,5000
240 GOTO 100

```

FIND ADDRESS

```

1030 REC=1
1040 GOSUB 3000
1050 PRINTCHR$(147); "      RECORD NO: "REC
1055 PRINT"SURNAME      "NAME$
1060 PRINT"FORE NAME    "FIRSTNAME$
1070 PRINT"HOUSE NO.    "HOUSE$
1080 PRINT"STREET       "STREET$
1100 PRINT"TOWN/CITY    "CITY$
1110 PRINT"COUNTY       "COUNTY$
1120 PRINT"POST CODE    "PCODE$
1130 PRINT"TEL NO.      "TELE$
1140 WAIT 197,64:WAIT 197,191
1150 GETA$:IF A$=CHR$(133) AND REC>1 THE
N REC=REC-1
1160 IF A$=CHR$(134) AND REC<CURRENTREC
THEN REC=REC+1
1170 IF A$=CHR$(136) THEN RETURN
1999 GOTO 1040

```

Again this routine is very similar to the equivalent sections for the other file structures. This is probably not surprising considering that all it does is to **PRINT** the results of the 'read from disk' routines in lines 3000-3999. Notice the differences in lines 1030 and 1040, however; in this example we are not setting the pointer to any particular position within the block, but are instead **READING** the block from the first character onwards, which is the default value.

ADD NEW DATA

```

2010 PRINT CHR$(147)
2020 CURRENTREC=CURRENTREC+1:GOSUB 10000
2050 PRINT"  ADDRESS BOOK                      RECO
RD NO.";CURRENTREC
2060 PRINT:PRINT"  PLEASE ENTER THE FOLL
OWING DETAILS"
2070 PRINT:INPUT"SURNAME          ";NAME$
2075 IF NAME$="" THEN NAME$="-"
2080 PRINT:INPUT"FORE NAME        ";FIRSTNAM
E$
2085 IF FIRSTNAME$="" THEN FIRSTNAME$="-"
"
2090 PRINT:INPUT"HOUSE NUMBER ";HOUSE$
2095 IF HOUSE$="" THEN HOUSE$="-"
2100 PRINT:INPUT"STREET NAME   ";STREET$
2105 IF STREET$="" THEN STREET$="-"
2110 PRINT:INPUT"TOWN/CITY     ";CITY$
2115 IF CITY$="" THEN CITY$="-"
2120 PRINT:INPUT"COUNTY        ";COUNTY$
2125 IF COUNTY$="" THEN COUNTY$="-"
2130 PRINT:INPUT"POST CODE     ";PCODE$
2135 IF PCODE$="" THEN PCODE$="-"
2140 PRINT:INPUT"TELEPHONE     ";TELE$
2145 IF TELE$="" THEN TELE$="-"
2150 REC=CURRENTREC : GOSUB 4000
2160 PRINT:PRINT CHR$(18);"PRESS SPACE B
AR TO ENTER ANOTHER RECORD ";
2170 PRINT"  OR RETURN TO RETURN TO THE
MAIN MENU  "
2180 WAIT 197,64:WAIT 197,191
2190 GET A$
2200 IF A$=CHR$(32) THEN GOTO 2000
2210 IF A$=CHR$(13) THEN RETURN
2220 GOTO 2180

```

These lines too are almost identical to the relative access file version. The only difference is the **GOSUB 10000** added in line 2020. This calls the subroutine that allocates a specific block for storing the data that is to be entered from the keyboard. As always, a check is made after each **INPUT** of data (in the lines ending with 5) to ensure that at least one character is assigned to each variable.

READ A RECORD FROM THE DISK

```

3012 PRINT#15,"B-R: "5;0; TRACK(REC); SECT(
REC)
3020 INPUT#5,NAME$
3030 INPUT#5,FIRSTNAME$
3040 INPUT#5,HOUSE$
3050 INPUT#5,STREET$
3060 INPUT#5,CITY$
3070 INPUT#5,COUNTY$
3080 INPUT#5,PCODE$
3090 INPUT#5,TELE$
3999 RETURN

```

This is the first routine in the program that is substantially different from the previous versions, so let's take a closer look at it. Line 3012 is the line that utilises the particular capabilities of the direct access commands by reading one block of data from the disk into the buffer, through channel no. 5. When using this type of file structure to store individual records we have to keep a separate file containing the track and sector addresses of each record in the main file. In this example we are using two arrays, **TRACK** and **SECT**, each of which contains 100 elements, that is one element per record. So, when this subroutine is called successively (from line 1040), the value of the variable **REC** determines which record is to be accessed. Looking back a bit, the value of **REC** is controlled by lines 1150 to 1170, so that pressing function key 1 will decrease its value by 1 and pressing function key 3 increases it by 1. Checks are, of course, made to ensure that the value of **REC** stays within reasonable limits. A call to this subroutine (line 3000) will then pull the required data from the disk.

Now that the **BLOCK-READ** instruction of line 3012 has been carried out data is in the disk buffer. Note that the data is still within the disk system; it has not yet been transferred to the computer. In fact we now have to **INPUT#** it, in the correct order as always, via the allocated channel. This task is performed by lines 3020 to 3090. Notice that, unlike the relative file version of this routine, there is no need to set the buffer pointer between each read of the block. In fact, each variable can be read sequentially from the buffer, provided that the information was initially put in there in a similar manner.

WRITE A RECORD TO THE DISK

```

4020 PRINT#5,NAME$
4030 PRINT#5,FIRSTNAME$
4040 PRINT#5,HOUSE$
4050 PRINT#5,STREET$
4060 PRINT#5,CITY$
4070 PRINT#5,COUNTY$
4080 PRINT#5,PCODE$
4090 PRINT#5,TELE$
4100 PRINT#15,"B-W: "5;0;TRACK(REC);SECT(R
EC)
4999 RETURN

```

This routine complements the one above; notice the similarity of action. The mode of action is slightly different, for reasons that should become obvious. This routine is called from line 2150, where the value of the current highest record number is firstly handed over to the variable **REC**. The data is then **PRINT#**ed to the disk buffer, (not to the disk itself), in lines 4020 to 4090. Once this is done the data must then be transferred to the disk, and this is performed by the **BLOCK-WRITE** in line 4100. Notice that the channel number specified within this command must be the same as that used by the **PRINT#** commands, otherwise only an empty buffer will be stored on the disk! As before, the track and sector address of the block to be used is found by using the two arrays **TRACK** and **SECT**.

Notice the flow of these last two routines. To write a block of data you must first of all fill the **RAM** buffer with the appropriate data and then **WRITE** the whole thing onto disk. To read data from the disk, on the other hand, you must first **READ** it into the **RAM** buffer before transferring it from there into the computer. In both cases the contents of the buffer are most important; you can only **WRITE** to the disk what has first been put into the buffer, and similarly you cannot **READ** data from the disk unless that too has been put into the buffer first.

END PROGRAM

```
5010 PRINT CHR$(147)
5020 GOSUB 8000
5030 CLOSE 5:CLOSE 15
5040 END:OF PROGRAM
```

Well, the program has to end at some point. Because we have tried to keep to the same line numbers for each routine, adding any new coding to the end of the program as necessary, the result is an 'End Of Program' section right in the middle of our program! Of course it makes no difference to the computer where each individual routine is placed, but if you were writing your own program independently then this routine would be better situated more logically at the end.

These lines simply **CLOSE** all the files that have been used within the program. Before doing so, however, a call is made to the routine at line 8000, which (as we shall see) stores the current **TRACK** and **SECTOR** arrays onto the disk, ready for the next use of the program. Note that line 5040 shows up an undocumented feature of Commodore **BASIC**. If you look at this line you will probably think that there is a syntax error in there, when in fact, there is a **REM** statement missing. However, the computer will perform the **END** statement and **END** the program without ever reaching the offending piece of code, so it will not therefore throw up a **SYNTAX ERROR** message.

READ THE ERROR CHANNEL

```
6020 INPUT#15, EN,EM$,ET,ES
6030 RETURN
```

Line 6020 reads the error channel and stores the error number in **EN**, the error message in **EM\$**, and the track and sector of the offending error in **ET** and **ES**. This is a general-purpose routine that can be called whenever an error report is required. Since only the channel is **READ**, the calling routine must perform any necessary actions resulting from particular errors.

READ IN THE SEQUENTIAL FILE

```

7020 OPEN 8,9,8,"RECORD DATA,S,R"
7030 GOSUB 6000
7040 IF EN=62 THEN CLOSE 8: GOSUB 9000:G
OTO 7000
7050 INPUT#8,CURRENTREC
7060 IF CURRENTREC=0 THEN CLOSE 8: RETUR
N
7070 FOR REC=1 TO CURRENTREC
7080 INPUT#8,TRACK(REC)
7090 INPUT#8,SECT(REC)
7100 NEXT REC
7110 CLOSE 8
7120 RETURN

```

In many ways this routine and the next are the most important in this program. We are reading the contents of a sequential file to obtain the **TRACK** and **SECTOR** arrays that contain the addresses of each block of data. These arrays, as we have already seen, are used to position the **READ/WRITE** head over the appropriate block when **READING** or **WRITEing** individual records. This routine is called at the beginning of each **RUN** of the program, from line 70.

The first task is to **OPEN** a file for **READING**, which is done by line 7020. A call must then be made to the '**read the error channel**' routine and on its return a check is made (line 7040 to ensure that the file already exists on the disk. If it does not (error number 62) then the channel is **CLOSEd**, and a call made to the routine at 9000, to enable the creation of a file. This provision must be added because on the very first **RUN** of the program there will be no file and no database. If this is not checked then the program will crash on its very first **RUN**, which is not very friendly!

Another way of doing this might be to add a separate option to the main menu which would allow the operator to create a new file. If you wanted to have several databases on the same disk this would be advantageous, but for a single file the process might just as well be automatic, as above.

The first item of data read from the file is the total number of records that have been filled to date, in line 7050. Again, if this has a value of **zero** it means that there are no records on the disk so the file is **CLOSED** and we **RETURN** to the main program. If, however, some records are already there we need to know their location. This is found by the loop of lines 7070 to 7100, where the arrays **TRACK** and **SECT** are **READ** in from the file. When complete, the file is **CLOSED** for **READING** and a **RETURN** is carried out.

WRITE THE SEQUENTIAL FILE

```

8020 OPEN 8,9,8,"@:RECORD DATA,S,W"
8030 PRINT#8,CURRENTREC
8040 FOR REC = 1 TO CURRENTREC
8050 PRINT#8,TRACK(REC)
8060 PRINT#8,SECT(REC)
8070 NEXT REC
8080 CLOSE 8
8090 RETURN

```

At the end of each session with the computer it is important to update the sequential file of **TRACK** and **SECTOR** addresses, otherwise the latest set of records will not be available for your next session. This is performed by the routine above. Quite simply, it records all the necessary information in the same order as the previous routine read it. First of all, the current total of records is recorded, followed by the individual items of data within the loop of lines 8040 to 8070. The routine is called from line 5020 and, at the end of its assigned tasks, that is to where it **RETURNS**.

CREATE THE SEQUENTIAL FILE

```

9020 OPEN 8,9,8,"RECORD DATA,S,W"
9030 CURRENTREC=0
9040 PRINT#8,CURRENTREC
9050 CLOSE 8
9060 RETURN

```

On the first **RUN**ning of the program, there will be no sequential file already there, so this routine is called to create one. All that it does is to create a file name in the directory and on the disk and then to record a single piece of information in the file, to the effect that the current number of filed records is zero. The file is then **CLOSED** again before the program **RETURNS** to line 7040.

ALLOCATE THE NEXT FREE BLOCK

```

10020 PRINT#15,"B-A:"0;1;0
10030 PRINT#15,"B-A:"0;1;0
10040 GOSUB 6000
10050 IF EN=65 AND ET=0 AND ES=0 THEN GO
TO 10500
10060 IF EN=65 THEN PRINT#15,"B-A:"0;ET;
ES
10070 TRACK(CURRENTREC)=ET
10080 SECT(CURRENTREC)=ES
10090 RETURN

```

This type of routine is at the heart of the direct access file structure. It is most important that you allocate a block before storing any data in it. When a block is allocated this indicates to the **DOS** that the block is in use and should not be used for any other purpose. So, if you do not allocate your block properly you stand a very good chance of having your database corrupted by any subsequent **SAVE**ing or **WRITE**ing operations. Actually, it is less a chance than a certainty, unless you are extremely lucky!

When an attempt is made to **ALLOCATE** a block, it will only be successful if that block is not already in use. If it is, the **DOS** will generate an **error message** that is particularly useful. The numbers that are generated as the track and sector (**ET** and **ES** in our example) are, in fact, the address of the next highest available free block on the disk. The error number is 65 and this is checked in lines 10050 and 10060. Looking at line 10060 first, you can see that all that is now necessary is to **ALLOCATE** the relevant block, the **DOS** having done all the work for you. The whole purpose of this routine is to find a free block to store a new record in and lines 10070 and 10080 store the address of the block in the arrays **TRACK** and **SECT** so that the record can be accessed at any time in the future.

Let us now go back to line 10050. If the **error number** 65 should be generated with **TRACK** and **SECTOR** addresses of **zero**, this signifies that there are no higher blocks available. Since we started **ALLOCAT**ing at the first block on the disk (**track 1, sector 0**), this implies that the disk must therefore be full. In such a case, it is only polite to inform the operator that this has happened, by **GOTO**ing 10500. A similarly unusual situation can arise when the program is **RUN** for the first time with a relatively empty disk.

The first time that an attempt is made to **ALLOCATE** track 1, sector 0 it will be successful, that is the block will be available and no **error message** will be generated. The easiest way to overcome this hiccup is deliberately to **ALLOCATE** the block twice, as in lines 10020 and 10030. At the second attempt the block will no longer be free and the required information can then be **READ** from the **error channel**.

FORCE THE END OF THE PROGRAM

```

10520 PRINTCHR$(147):PRINT
10530 PRINT"          SORRY, THE DISK IS F
ULL"
10540 PRINT:PRINT"          NO MORE RECORDS C
AN BE ENTERED"
10550 CURRENTREC=CURRENTREC-1
10560 REM  GOSUB 8000
10570 CLOSE 5 : CLOSE 15
10580 END:OF PROGRAM

```

In the unfortunate event of a disk becoming full during record entry, it is important that the operator is informed, which is the function of this routine. The last record cannot, of course, be **ENTERED** so line 10550 reduces the value of the current record by 1. At this point an attempt is made, by **GOSUB**ing to 8000, to **SAVE** the current sequential file onto the disk, but this is unlikely to succeed because the disk is full! If you are planning a serious application you should make provision for this possibility. The easiest solution is to use a new disk solely for your database. The example program allows for storing 100 items of data in 100 blocks so there would be no possibility of the disk filling up completely in this case. A few simple calculations before you start your own database can save a lot of anguish before you find that you have wasted hours of keying in because your disk is full. A more sophisticated but equally practicable approach would be deliberately to **ALLOCATE** sufficient blocks, the first time that the

program is **RUN**, to allow the sequential file to be **SAVEd** in the event of the disk becoming full. Before carrying out such a **SAVE** you would have to remember to de-**ALLOCATE** the aforementioned blocks using the command **BLOCK-FREE**.

THE PROGRAM IN USE

Now that the whole program has been keyed in (correctly?) you should **SAVE** it onto a disk before you **RUN** it, just in case. The other thing you should do before **RUNning** it is to ensure that the disk has plenty of free space on it, say, at least 200 blocks. When you have done this you can then **RUN** the program. The format for using it is exactly the same as with the relative access file demonstration program. From the main menu you can choose to enter a new record, sort through existing records or end the program, just as before. When you are **ACCESSing** records from the disk you can press function key 3 to display the next record in the series and function key 1 to display the previous one. Function key 7 will **RETURN** you to the main menu. After **ENTERing** new data into a record you can press either the **SPACE** bar to enter another record or the **RETURN** key to return to the main menu.

The whole flow of this program is almost exactly the same as with the relative file demonstration program. Notice that the speed of execution is very similar also. This is hardly surprising since it is the same file structure as that which the current demonstration program is based upon and is simulating. We could just have easily chosen to emulate a sequential access file to store all the record data, or even created our own, unique file structure. If you have particular and unusual requirements for a database you might consider creating your own structure, and indeed the process can be quite pleasurable and rewarding. Before you set out on such a task, however, I offer you one word of caution. When you are using direct access files on a disk that is also used for writing other data or **SAVEing** programs, the data in the direct access file is never completely safe. The validation of such a disk, even accidentally, will destroy the file completely in less than 30 seconds; a sobering thought when the data is particularly important. For the general storage of data, you should always consider the sequential file and the relative file in preference to this structure, for this reason. Only when you have considered all other possibilities and decided that a particular situation demands the special capabilities of direct access files should they be employed. Bon voyage.

8

Changing Device Numbers

As we mentioned earlier, you can connect up to four 1541 disk units to your Commodore computer simultaneously. However, if you were to do so without any preparation it would create a conflict within the system whenever any access to a disk-drive was attempted. This situation would arise because all the units would be device number 8 so they would be unsure which unit was being addressed. To cover up their embarrassment, none of the disks would do anything at all, just sit there and look at you! Fortunately, although all 1541s are set when manufactured to device number 8 it is possible easily to alter the device address for any disk unit. In fact there are two methods of carrying out this operation. The software method will alter the device number whilst the disk drive is on although it will revert to number 8 when the unit is turned off. The hardware method will change the device address more or less permanently.

To change a device number on a disk drive by the software method you use a single line of simple code, really just the equivalent of **POKE**ing the new number into a location within the disk drive **RAM** memory. To do this we have to use the **MEMORY-WRITE** command:-

```
PRINT 15,"M-W"CHR$(119)CHR$(0)CHR$(2)CHR$(DEV  
+32)CHR$(DEV+64)
```

Here the first two numbers are the location of the register within the **DOS** containing the current device number in standard 6502 form of low byte/high byte;- $119 + 256 * 0 = 119$. The next number (**CHR\$(2)**) is the number of bytes that are to be sent, and the final two numbers are the device number added to either 32 or 64. For example, if you want to change your disk drive to device number 9, you should carry out the following:-

1. Turn on the disk drive that you want to change and no other.
2. Type in and then **RUN** this short program:

10 OPEN 15,8,15

20 PRINT 15,"M-W"CHR\$(119)CHR\$(0)CHR\$(2)CHR\$(41)CHR\$(73)

30 CLOSE 15

3. Having **RUN** the program, the disk drive will now become device 9.
4. Now turn on the next unit, which will be device 8.
5. The procedure can be repeated if you wish to change the device number of several units, specifying a new number for each.

This arrangement is fine for short-term changes, but if you plan to use multiple disk drives frequently you might want to contemplate making a permanent change to one or more units. This hardware method involves making a slight adjustment to the circuit board of the disk drive, but it is very simple and, if performed carefully, will not invalidate the manufacturer's warranty. Follow these steps:—

1. Disconnect all leads from the disk unit, including and especially the power lead.
2. Turn the drive over and remove the four screws holding the casing together.
3. Turn the drive upright again and remove the case top. The circuit board will be clearly visible.
4. You should now locate the two jumpers that control the device numbering of the unit.
5. There are currently two different circuit boards that may be used in the 1541. Early versions of this disk drive contain a board where the jumpers are to be found along the left edge, (looking from the front), about half way down the length of the board. In this case jumper 1 is the one to the rear of the machine, with jumper 2 in front of it. Newer disk drives may contain a redesigned circuit board with the jumpers relocated to the centre-front of the board. In this case, jumper 1 is still the rearmost of the two connections.

Whichever board your disk unit contains, you should cut the jumpers according to the table in order to change the device number to your own requirements.

DEVICE	JUMPER 1	JUMPER 2
8	JOINED	JOINED
9	CUT	JOINED
10	JOINED	CUT
11	CUT	CUT

6. When you have carefully made the necessary breaks you should replace the top casing, the four screws and finally the cables. Your disk drive unit will then be ready for multiple unit use.

If you should find that neither of these methods exactly matches your requirements, there is yet a third option. Instead of just breaking one or other of the jumpers, you can do this but then connect a small, mountable, two position switch to both sides of the break. Such switches are readily available from any electronics shop and can be mounted on the outer casing so as to be readily accessible. This course of action, whilst making for convenient operation of your multiple disk system would, of course, invalidate the guarantee.

9

Introducing Printers

Once you have got your disk drive up and running and you have begun to realise what sophisticated programming you can do with it, if you haven't already felt the need for a printer you are sure to before long. With a printer you can record all your **BASIC** thoughts in deathless prose, by way of proof that those lonely hours of programming have not been fruitless; you can also strew your computing area with reams of impressive-looking printout to add a distinctive professional aura to the proceedings. You can use it for word-processing, and convey a little authority in your letters to the local council. What's more, you can produce hard copy of your program coding which you are sure to find an invaluable aid to debugging and program development.

For the small business user a printer will be simply essential to the paper side of your business transactions: receipts and invoices to send to customers, letters and documents that you can keep on disk but still need to send to others on paper. If you are smart you will buy a printer that can also do service as a typewriter, saving yourself space and money at the same time.

Whatever your motives, once you have decided that you simply cannot do without a printer your next move should be to attempt to inform yourself well enough to make a good choice. A quick flick through the advertisements in any popular computer magazine is likely to leave you daunted by an apparently endless array of different types and makes, adorned with buzz terms like "multifont", "cps." and "bidirectional logic-seeking". You will probably also notice that they all look very similar and you might be forgiven for wondering whether a disparity of several hundred pounds between prices for remarkably similar cream boxes is

attributable more to the economic aspirations of the manufacturers' shareholders than to a genuine difference in quality. In fact, there are several distinct types of printer, each best suited to particular applications, and by and large prices do actually reflect the quality of the printed output. So take the trouble to do some research before you buy; you are sure to find it well worthwhile for an investment that is probably going to cost more than your computer.

CHOOSING A PRINTER

The main criteria that will influence your choice, beside cost, are : type quality, speed and flexibility.

Type quality can vary from the sublime to the ridiculous, and so can print speed. Some printers produce poor, faint printout, so patchy and widely spaced that it could never be used for professional purposes. However, although these are also likely to be cheap and slow, (around 50 characters per second or less), they may be all that you need to list out your programs for your own documentation purposes. Others produce output that looks as if it has just come straight from the printer of an executive secretary, but they are likely also to be expensive and agonisingly slow although high quality printers often have a faster, draft mode as well. There is an unfortunate tradeoff between speed and print quality; you cannot have both and on the very cheapest dot-matrix printers you are unlikely to get either! The choice is yours.

Flexibility is also something you have to pay for. When considering how much you need, particularly if you are a business user, it is a good idea to think ahead a little and buy not just for your present needs but also for those as you see them developing over the next year or two. Will you want a wide choice of typeface, the alternative to use single sheet or roll paper as well as fan-fold, sophisticated software options? These aspects all need to be considered.

As we explain the main types of printer in this chapter, discussing their strengths and weaknesses, be thinking about these points so that you will be able to put them in the right perspective for your own needs.

There are other factors also likely to influence your choice. Although they are probably less important to you than the ones we have mentioned so far, you should also take into account: noise, size and running costs.

Noise can be something of a problem. All printers are noisy things but some types are much worse than others. Dot matrix printers are usually a lot quieter than daisy wheels, which tend to sound like machine-gun fire. You can buy acoustic hoods to deaden the sound but they can be quite expensive. You should be asking for a demonstration of any printer before you buy it, so as well as looking at the type quality and hearing about the million and one things that the salesman tells you it can do, listen to the noise it makes. If quietness matters to you and you don't think that you can live with this one rattling away you can always search for a quieter model.

The size of the printer you want will be to a great extent determined by where and how you intend to use it. In a normal office environment you will have no problems with a standard printer. However, if you are likely to travel with it, to conferences, customers or branch offices, for example, you will find a small, compact, battery-driven portable printer of most use. In this case you need to take dimensions, weight and portability into account when making your choice.

Finally, we mentioned running costs, which are often easy to overlook. Before you buy you need to know such things as the price of new ribbons, how long they will last in terms of characters printed, and also whether they are easily available in the high street or have to be especially ordered from the manufacturer. You also need to consider the cost of paper, especially if you will be using thermal paper which is quite a bit dearer than normal and often harder to obtain. If you are buying a battery-operated printer you should also take into account the price of batteries and how often you will have to replace them. These things might seem trivial when you are considering spending quite a large sum on a printer, but once you have bought it you carry on spending on accessories such as these and the costs can mount up for a home or small business user on a limited budget.

DOT MATRIX PRINTERS

Dot matrix printers are very widely used for a variety of applications and are really the work-horses of the species. They work by means of a printing head consisting of a vertical line of metal points (usually nine). Each point is independently mobile and can shoot out to make an impact through carbon ribbon onto paper as a dot. As the head moves across the page it is controlled by the printer's inbuilt software so that the correct pattern of dots is laid down to create printed characters on the paper.

Usually dot matrix printers can be programmed to produce more than one type-style. The standard typeface is Pica, like that of most office typewriters, and it is often possible also to obtain enlarged or condensed styles, or sometimes quite fancy ones.

The main strengths of the dot matrix printers are that they are, relatively speaking, cheap, fast and quiet. Personal printer models are now available for around £100 and prices are becoming increasingly competitive. They can print very fast, usually around 50-150 characters per second (cps.), and sometimes up to 250 cps. Also, although like anything mechanical there is a certain amount of noise when they are in operation, in comparison with golfball typewriters and daisywheel printers they are fairly quiet.

The weaknesses of dot matrix printers lie mainly in the quality of the print. The standard nine pin matrix produces print that is clearly a series of dots, not a solid outline, and it can sometimes be difficult to read. Moreover, it is not usually possible to obtain true descenders with the cheaper versions of these machines, that is, the tails of the y, the g and other descending letters do not drop below the line but are printed level with it, often giving the impression of capitals mid-sentence. Standard dot matrix printout is also inclined to be rather faint, owing to the spacing between dots and the speed at which it was printed. As a result it does not photocopy well, which could be a disadvantage if you like to keep multiple copies of everything, particularly in a business environment.

There are some dot matrix printers on the market with a denser matrix producing bolder print, true descenders and a considerable improvement in print quality, but these are at the top end of the price range. There is, currently, a trend towards ever denser dot-matrix heads with up to 24 pins. These provide the best of all

worlds because they can produce excellent quality print-out speedily and quietly. At the same time, the inherent flexibility of the dot-matrix method of producing characters allows for an increasingly wide range of typefaces to be offered on the printer and selected by software control. Whilst this type of printer, for instance the EPSON LQ-1500 can cost up to £1000, or even more, prices for these must, almost inevitably, go the same way as for other computer equipment: downwards.

To sum up, on the whole dot matrix printout is not a professional standard or letter quality, but it is perfectly suitable for private use, rough drafts and informal work. Therefore these printers are best suited to people who do not want to spend a great deal, for whom quality is not overridingly important, but who may want large amounts of data printed off quickly and frequently for private use.

DAISYWHEEL PRINTERS

Daisywheel printers, like daisywheel typewriters, are more likely to be found in the office than in the home. Instead of being a matrix of dots, the printing head is disk shaped, with the letters arranged on the ends of spokes radiating out from the centre, like the petals of a daisy. To print each character the head spins round to the correct character position and a hammer strikes the radiating spoke with some force to produce a character through a carbon ribbon onto the paper beneath.

The principle advantage of this type of printer is the quality of its printout. Because the characters are cast in high quality plastic rather like a traditional typewriter or typesetting machine the outline they produce is bold, distinct and unbroken. The printout is of a very fine quality and eminently suitable for business correspondence, reports, publicity material and any occasion where you want to create a professional image.

Most printers can be used with a selection of daisywheels giving different typefaces, or fonts as they are sometimes called. Instead of a traditional Times or Roman face you may be able to choose something rather different like Sanserif. You always get true descenders with these printers and the output is clear enough to photocopy well – unless you have recycled the ribbon too many times!

There has to be drawbacks, doesn't there? In fact, there are, namely cost, noise, speed.

Daisywheel printers are much more expensive than dot matrix ones. It is a competitive area though and prices have actually dropped considerably from what they were when the first models were launched. If you shop around you may get a bargain but it may still cost you much more than buying a dot matrix printer.

They are also noisier than other printers. In use they produce a loud metallic clack that can sound like a machine gun to a nearby sufferer. Again, you can get an acoustic hood to muffle the noise, but these are not cheap and boost the cost of your purchase quite a lot.

Finally we mentioned speed. Because of its mechanics: the spinning daisywheel and the force needed to strike each character out, the speed is quite low on this type of printer, averaging around 10 c.p.s. for the cheapest machines up to 60 c.p.s. for the really expensive models. If you were printing out vast amounts of data regularly this could be quite a disadvantage, but for standard word-processing applications it is not likely to be much of a problem.

By way of a summary then, daisywheel printers produce top quality output for an upmarket price-tag at the cost of speed and quietness. They are therefore best suited to someone in business or concerned about a professional image to whom quality is most important and who is prepared to pay for it.

TYPEWRITER-PRINTERS AND PORTABLES

Typewriter-printers are an option often overlooked by the prospective buyer because there are a relatively small number of models on the market and they get less media exposure.

They can be either dot-matrix or daisywheel, they can be astonishingly cheap or very expensive; what they have in common to put them in this special category is an integral keyboard and the capacity to function independently as a typewriter.

Many of them also have a small memory, up to around 4K of RAM which will store about a page of text for printing out later. In this respect they are almost like mini-word processors. Although some models are full size typewriters, more are battery-operated portables aimed at the mobile user.

This type of printer is useful and versatile, and has quite a few advantages for the home or small business user. For a start, as we have said, they can be very cheap and some of the low-end dot matrix ones should be well within the purse of the hobbyist.

Much of their cheapness is due to their size and their main advantage – portability. The portable printer-typewriters available now are extremely compact, light and easy to carry. They fit well into a standard briefcase, although they often have a carrying handle of their own, and are ideally suited for people such as journalists, representatives, executives at a conference and anyone else who needs to write as they travel. In the home they take up little space and are easy to pack away when finished with.

The third great benefit of these printers is their versatility. Because they double as typewriters you can rattle off a letter to the council or a memo to your business partner without having to load a word processing package and get involved with formatting or special commands. You can save quite a bit on equipment costs by using a dual purpose machine like this provided that the printout is good enough. Across the range of models available the print quality varies between the excellent and the ghastly. Again, it's your money and your choice.

The versatility of such machines extends also to the variety of printing modes that you can use. Where there is a memory you can type in copy and choose to have it printed out always a line behind, giving you time for second thoughts and corrections. You can also store paragraphs and use the editing facilities as you would a simple word processor.

There has to be a hitch somewhere, so let's look at the weaknesses of portable printer/typewriters. Many people used to a heavy-duty standard typewriter would find these printers rather flimsy to work on. They slide about easily, the keys are close together, and you might have doubts about whether they are substantial enough for heavy, regular use. This does not apply to the printer/typewriters that are not portable, however, these are the same size as the customary office models.

Running costs can be quite high. As well as regularly replacing ribbons you will be buying new batteries, and if it's a thermal model – relatively expensive thermal paper in addition.

Excellent though they may be as typewriters, some models have limitations as printers. For example, they may have all the characters a typist would want, but not all those a programmer might need, such as chevrons, curled brackets or graphics characters. If it is not on the keyboard it is not available, so do check, especially if you want to connect it up to a Commodore computer.

In conclusion, typewriter-printers, particularly portable ones, are worth serious consideration by the home or small business user. They are compact, easy to use, dual purpose and often very cheap, although they are not heavy duty machines and running costs can be relatively high.

Other methods of printing are used in certain printers, including ink-jets being squirted (messy if it goes wrong!), lasers being used to scorch the paper and in the very cheap printers, sparks being used to darken specially treated paper – this last method being used in the ZX printer amongst others, where the printer costs less than £30 but the paper will set you back some £5 a roll!

INTERFACES

You can use almost any printer with a Commodore computer, but you will need to buy an *interface* to go with it. Basically, what an interface does is to translate the signals generated for the printer by the computer into a form that the printer you are using can understand. If you are using a Commodore printer, no translation has to be done, but most printers expect different signals to work on. The signals expected by a printer (or indeed any other external device) are known as a *protocol*. There are several standard protocols in use, the two most common being called RS 232 serial and Centronics. The signals from the computer, however, are in neither of these formats, so an interface must be used to translate between computer and printer.

There are many different interface packages on the market at the moment, ranging in price from about £15 up to £200. They all do the same basic job, but they work in different ways, at different speeds and in different protocols. Obviously, the more you pay the more you get; one called Interpod which sells for about £100 is switchable, so you can use it with practically any printer at any speed. This is fine, but if you only want to use it with one printer you could probably get a suitable interface for a lot less. The one used in the examples in this text is a centronics interface produced by Aztec, consisting of a simple cable which connects to the user-port of a VIC 20 or Commodore 64. A short machine code program provided with the lead intercepts signals on their way to the printer and sends them through the cable in the correct format, making the printer respond to the normal Commodore printer commands. We had no difficulties with this system, which only cost £29. This type of interface has the disadvantage that the software needed to drive it takes up memory that you might want to use for programming, but it is written so that it would be fairly simple to move to another area of memory should the need arise. Other interfaces plug into the computer's cartridge slot; these too occupy memory, while others connect to the serial port where you would connect a standard Commodore printer. When you go into a shop to buy an interface, make sure that you get the correct interface for the printer you are going to use – after all, what's the point of buying an expensive interface when one at half the price will do the job just as well?

10

The 1526 Commodore Printer

The range of printers available for use with home computers has never been greater than at the present time and there is little doubt that there is one to suit every need and desire. Combining a Commodore computer with a disk unit and a printer makes for a powerful and versatile collection of hardware, but possibly a somewhat expensive collection as well. For cheapness, the Commodore 1520 printer/plotter is excellent value for money at around £100 and it's four colour plotting potential is marred only by it's excruciatingly slow pace of little more than 10 characters per second (cps.). At the other end of the spectrum, there are printers costing upwards of £2000 that rush through paper so quickly that most of Sweden is working overtime to keep up with their insatiable desires!

In between these extremes there are many printers in the price range of £200-£500 that can equally well perform the usual tasks expected of them, including a range of machines from Commodore. In this chapter we shall be looking briefly at the merits or otherwise of the Commodore printers before going on, in chapter 11, to look in much more detail at one printer in particular, the CBM 1526.

When you are considering the purchase of a printer for use with a Commodore computer there are two overwhelming factors that you should bear in mind. The first is that the character set on the Commodore computer is non-standard and cannot be printed out in its entirety on non-Commodore printers. This is generally not a problem for uses such as word-processing where the printer only has to do the words, and not the graphics characters. Program listings, on the other hand, can become almost unreadable because of the lack of appropriate graphics and cursor control characters on most printers. The second problem lies in

connecting the printer to the computer. Commodore home computers use a non-standard method of interfacing themselves to their periphery, a serial form of the IEEE-488 parallel port. This cut-down (ie. cheaper) port can make life particularly difficult for the printer hunter because very few manufacturers, apart from Commodore of course, produce printers that can be directly connected to a Commodore computer. Whilst it is possible and not particularly difficult to use an interface connection between the two this does add to the cost, inconvenience and mass of wires trailing spaghetti-like around the back of your machine. In other words, for convenience, compatibility and ease of use you would be well advised to take a good look at a Commodore printer before considering any other make.

The MPS-801 is the current replacement for the robust CBM 1525 and both are correctly described as graphics printers and can be considered good value for money as a cheaper, general purpose printer. At the other end of the dot-matrix scale, the CBM 1526 is described (by Commodore at least) as a business printer with 'near letter quality' printing (NLQ). Whilst the quality of the print is very good (all of the listings in this book were produced on a CBM 1526) it is stretching credibility to think that the quality is anywhere near that of a daisywheel typewriter.

Both of these printers will work quite happily with either the VIC-20, CBM-64 or the SX-64 and the choice between them must, ultimately, be a personal one based upon financial as well as mechanical considerations. Whilst the MPS-801 suffers particularly in not having any true descenders (tails) on its characters, unidirectional printing and slow printing speed, its saving grace for many people lies in its graphics capabilities. This machine, like the 1525 and 1515 before it, has the ability to print by what is known as programmable dot-addressing. This means that you, the programmer, can print dots on the page at any position you want with a few not particularly easy commands. This means that you can print out charts and graphs, screen dumps and pictures with the appropriate software.

On the other hand, the CBM 1526 is presented to the world as a business printer and it is altogether more sophisticated than its cheaper siblings. This machine prints bi-directionally at approximately 80 cps and it does have true descenders below the line to give a much more readable print-out. There are no graphics functions built-in, other than a single programmable character, but

the 1526 has some very powerful text-formatting commands, as we shall see in the next chapter.

Before moving into greater depths, however, let's look at the common ground of printer usage on the Commodore range of home computers; the similarities rather than the differences.

OPEN

The **OPEN** command is used to **OPEN** a communications channel between the computer and the chosen peripheral device. Commodore have deemed that device numbers 4 and 5 should be dedicated to printers, device 6 for the printer/plotter and numbers 8 to 11 for the disk-drives. The format of the command is very similar with a printer as with the disk-drive. The Commodore range of printers default to the upper case/graphics mode when first turned on; the same character set as the computer when it is powered up; so there is no need to send a secondary address if this is the required mode. To change to the lower case/upper case set the channel must be opened with a secondary address of 7. For example:—

OPEN 4,4

will set or keep the printer in the standard character set whilst:—

OPEN 3,4,7

will open a channel for use with the lower/upper character set.

PRINT

Having **OPENed** a channel, you actually send text to the printer in one of two ways. You can either use the **PRINT #** statement, generally the preferred method, or you can use the **CMD** command to redirect the primary output directly to the printer. When using **PRINT #** all the rules that we learned in the disk drive section of this book apply. That is to say that any of the following formats for the command can be used:—

```
PRINT#4,"HELLO"  
A$="HELLO":PRINT#4,A$  
PRINT#4,1234  
A=1234:PRINT#4,A  
PRINT#4,"HELLO";"HELLO"  
PRINT#4,"HELLO","HELLO"
```

and so on.

In normal usage there is nothing clever or different about this command compared with its use with sequential files on the disk drive. The print modifying characters, the comma and the semi-colon, both have a similar effect upon output to the printer; the semi-colon causes printing to continue from the current position of the print head upon the line and the comma causes the print position to be forwarded by 10 characters. Notice that the comma has a slightly different effect upon the printer output to that which it has upon the video screen, where the comma causes the print position to be advanced, on the screen, to the beginning of the next imaginary column of 10 characters rather than always by 10 characters.

CMD

An alternative to the **PRINT #** command is to use **CMD** to redirect the primary recipient of output from the video screen to the printer. The limitations to this command have already been discussed earlier and they also apply when using the printer. There is one major and necessary use for **CMD** for which there is no alternative; producing program listings. On other occasions **CMD** can be used instead of the more normal **PRINT#** but you should not really become over-enamoured with the command because of the inherent possibility of your program hanging-up with the printer still listening for its next batch of data. Remember that the **CMD** command does not 'unlisten' the peripheral device; for that you must issue a **PRINT#** before **CLOSE**ing the channel of communication.

LIST

By now you must be familiar with this command as it was probably one of the first that you learnt, back in the dark days when you first bought your computer and you were shrouded in the mystery of high-technology and 'programming languages', line number and keywords.

LIST does, of course, command the computer to list out the current program onto the video screen. When used in combination with **CMD** the program will be **LISTed** to the specified device, in this case a printer. Exactly the same syntax can be used with a printer; parts of programs can be **LISTed** or even single lines. When the required part of the program has been **LISTed** to the printer then you must always remember to issue a **PRINT#** statement, otherwise the printer will still be listening and the whole system will behave unpredictably.

CLOSE

This command does just what it claims to; it **CLOSEs** the communication channel between the computer and the peripheral which was specified in the associated **OPEN** statement. The printer can only be used to output data so **CLOSEing** a channel to it is obviously not as critical as ensuring that a disk file is properly **CLOSEd**. However, the computer will only allow you to have a maximum of 10 files **OPEN** simultaneously so it is always prudent to **CLOSE** all files not actually being used.

PRINTING IN THE DIRECT MODE

All home computers, including the Commodore range, allow two distinctly separate modes of communication between themselves and the outside world, the operator or programmer. These are described by convention to be the direct mode and the deferred or program mode and the distinction is a simple and easily understood one. When using the computer in the direct mode, it

will carry out it's instructions immediately. For instance, if you type in from the keyboard:—

PRINT "HELLO" <RETURN>

when you press the return key, the computer will immediately print **"HELLO"** onto the screen. In contrast, the deferred or program mode is where the instruction is within a program line:—

10 PRINT "HELLO"

In this case nothing apparently happens when you press the return key. What does actually happen is that the program line is stored into the computer's memory and the action is deferred until the program is **RUN**. The fundamental difference between the two modes is that instructions in the direct mode are performed immediately whilst in the deferred mode they are stored in memory for later use.

It is quite possible to perform any of the functions of the printer in the direct mode, typed in straight from the keyboard. You can **OPEN** a file, **PRINT#** data to the printer and **CLOSE** the file again, all direct from the keyboard. Try the following:—

OPEN 4,4 **<RETURN>**

PRINT#4,"DIRECT MODE" <RETURN>

CLOSE 4 **<RETURN>**

and the printer will have printed **"DIRECT MODE"** onto it's paper. You can, of course, **PRINT#** whatever you choose and you are able to perform as many printing commands as you care to type in before closing the file. I don't think that you will want to type in too many instructions, because the main disadvantage of direct mode usage is that you do have to keep typing the instructions in; they are never saved in memory for reuse. To be perfectly honest, the uses of direct instructioning with a printer are rather limited and tedious. The single most important use is when **LISTING** a program to the printer to obtain hard copy print-out. To achieve this use the following process. First you have to **OPEN** a channel to the printer by:—

OPEN 4,4 **<RETURN>**

Now you must redirect all output to the printer by using:—

CMD 4 **<RETURN>**

Notice at this point that the printer prints '**READY**' before the cursor returns to the screen. Strange? Well not really because you have just told the computer to redirect ALL of its output to the printer and what does the computer tell you when it has finished performing its instructions and is ready to receive more? Ready. So, the final instruction to give the computer is to tell it to **LIST** your program and instead of appearing on the screen the lines of code will appear as hard-copy. When the listing has finished, the printer is still the primary output device and it is still listening for more output. To terminate this relationship and 'unlisten' the printer you must issue a **PRINT#** command such as:-

PRINT#4

<RETURN>

If you don't do this then strange things can happen. Why not try it for yourself? You can't do any damage to either the computer or the printer.

PRINTING UNDER PROGRAM CONTROL

It may appear that apart from the specific instance of **LISTing** a program, the use of a printer in the direct mode is somewhat limited, to say the least. This is not so as the real use for computers lies in their ability to store and run programs and the many advantages of using a printer with your computer can only be realised from within a program. In Chapter 11 we shall be examining all of the various sophisticated commands and functions that are available on the CBM 1526 printer and, finally, in Chapter 12 we shall be casting our by now jaundiced, blood-shot and heavy lidded eye over a representative model from the vast selection of non-Commodore printers that can be coaxed into an harmonious co-existence with your computer.

The commands that are available to allow you to make programmed use of a printer are not essentially difficult to understand and indeed they have all been covered in this chapter already. By this stage in the book, you should be able to write programs for yourself and an extended discourse upon the art of programming would be blatantly boring both for us to write and for you to read. There is only one instruction to allow you to send data to the printer, **PRINT#** and we leave it to you to write your own

programs that demonstrate it's use. After all, if you can't do this by this stage in the book then we have failed in our primary task of dispersing our presumed knowledge. Additional commands or pseudo-commands to allow access to the in-built software features of any particular printer are of neccessity machine specific and should be adequately documented in the manual supplied with the printer.

USING YOUR PRINTER TO THE FULL

When you are using a printer with your computer, you can often fall into the trap of treating it simply as a means of obtaining hard copy, without exploiting its full potential to produce easily nicely laid-out and professional looking output. The CBM 1526 contains many advanced features and is, indeed, an intelligent device; it contains its own 6502 processor and a small amount of **RAM** to allow you to define your own character shapes. There are many *formatting* commands built into the machine that, as we shall see in this chapter, give you a great degree of flexibility in the presentation and layout of the finished copy.

The system of commands that Commodore have implemented are passed to the printer using the secondary address that is associated with the **OPEN** command. There are a total of 11 secondary addresses that can be used with the 1526, numbered 0 to 10. The function of the secondary address is to act as a means of instructing the printer's internal microprocessor to perform specific tasks; a rather crude form of computer language where keynumbers are used instead of the more familiar **BASIC** keywords with which you instruct your computer. To use a formatting command you must **OPEN** a file with the associated secondary address and then, using **PRINT#** statements to that file, issue the precise form of the instruction. Let's look at these commands and controls.

SA=0 - NORMAL PRINTING

A secondary address (**SA**) of zero is the normal, default, setting. When you first turn on the printer it is automatically set to this value so you do not have to specify **SA=0** for it to become operational. In this mode, all data is printed out in exactly the same linear format as it is received from the computer. Up to 80 characters will be printed

on each line and a new line will be started whenever a carriage return (**CHR\$(13)**) is received. If more than 80 characters are received from the computer without a **CHR\$(13)** the printer will perform a carriage return and a line feed automatically and continue printing at the beginning of the next line. To illustrate this point, type in the following:-

```

10 REM *****
12 REM *                               *
14 REM *      PRINTER DEMO          *
16 REM *                               *
18 REM *****
20 :
30 OPEN#4,4
40 GET KEY$:IF KEY$=" " THEN 40
50 PRINTKEY$;:PRINT#4,KEY$;
60 GOTO 40

```

When you **RUN** this little program you will have the facility to type in anything that you like from the keyboard. When you press the **RETURN** key, or if you type more than 80 characters without doing so, then the printer will output your jottings onto paper. The main point of this demonstration is that the printer will print your input in exactly the same format as it was received from the keyboard, although not necessarily in the same typestyle. You can illustrate this point, and remember it for the future, by changing from upper/graphics to upper/lower character sets in the middle of running the program (by pressing the **SHIFT** and **CMB** logo keys simultaneously). Notice that although the computer screen will change to upper/lower case characters the printer will still print onto paper using its original upper/graphics set! You can, however, change to reversed characters whenever you choose to in this mode.

SA=1 PRINT FORMATTED DATA

A secondary address of 1 is used to send data to the printer so that it can be formatted before being printed. The information to be printed out is arranged according to the various instructions that will have been sent via secondary address 2 (see in this chapter for details). If no formatting instructions have been issued, or if they have been cancelled, all the data will be printed out unchanged, just as it would with secondary addresses.

In order to format string data properly you must send it from the computer in an already partially-formatted form. In particular you should have to delimit each field within the data so that the printer will be able to tell where one item finishes and another begins. Character number 29 (**CHR\$(29)**) acts as this delimiter and this should be used to separate the various fields within your data. Character 29 is the cursor right character on the Commodore computer so you can, if you wish, add a 'cursor right' to the end of each field instead of the more cumbersome and lengthy **CHR\$(29)**. When string data is to be formatted, all leading spaces are stripped from the data before formatting starts. So, if you particularly want leading spaces to remain in the finished print-out, you should pad out the string data before sending it to the printer with shift spaces, (**CHR\$(160)**), because these will not be removed.

Although we have not yet covered the formatting instructions themselves, type in the following and notice the difference that is made by the inclusion or exclusion of the characters **32** (space), **29** (cursor right or skip) and **160** (shifted space).

```

10 REM *****
12 REM *
14 REM *      PRINTER DEMO      *
16 REM *  SECONDARY ADDRESS=1  *
18 REM *
20 REM *****
30 OPEN 1,4,1
40 OPEN 2,4,2
50 PRINT#2,"AAAAA      AAAAA"
60 PRINT#1,"ABCDE FGHIJ"
70 PRINT#1,"ABCDEFGHIJ"
80 PRINT#1,"  ABCDEFGHIJ"
85 REM SHIFTED SPACES IN NEXT LINE
90 PRINT#1,"  ABCDEFGHIJ"
100 CLOSE 2
110 CLOSE 1

```

```

ABCDE
ABCDE      FGHIJ
ABCDE      FGHIJ
  ABC      FGH

```

When you run this little program with a 1526 connected you should get four lines of print, as shown immediately below the listing. The first line is generated by line 60 in the program. Everything within the quotation marks is treated as a single field and is therefore formatted into the first column. In line 50 we have specified that the format should be two columns, each one, five characters wide and separated from the other by four spaces. In line 60 the field to be printed is 11 characters wide (ABCDE FGHIJ) so only the first 5 characters (ABCDE) can be printed within the first column and the rest is truncated and ignored. Line 70 shows the effect of the 'skip' character. Because this character has been used the data is now split into two fields, ABCDE and FGHIJ and the formatting acts accordingly to print out two columns. This is shown in the second line of printout. Line 80 performs the same function as line 70, but this time there are spaces added in front of each field. Notice that these spaces are stripped from the front of each field before printing takes place, as illustrated in the third line of print. Finally, line 90 demonstrates that shifted spaces are not removed but are left as part of the field to be formatted. This is why the final line of print looks as it does, that is to say 'ABC FGH'.

SA=2 - STORING THE FORMATTING DATA

A secondary address of 2 is used to send formatting data to the printer and is both the most complex and the most versatile of all the secondary addresses that can be implemented on the CBM 1526. By using the formatting capabilities available with this, you can justify columns of figures either to the right or to the left, and you can align columns by the decimal point. By using several forms of justification on a single line you can freely mix numeric and string data freely to create properly laid out documents that are easy to read and clear to understand, a great boon when studying columns of figures for the VAT account, for instance.

The characters that indicate the required format for printing are sent to the printer using **SA=2**, and there is a direct one-for-one relationship between the position on the line of the formatting character and the position to which it refers on the printed page. Because of this, when you are designing your own layout it is often good policy actually to print out the formatting string, now using **SA=0**, so that you can see the exact position on the page of all the various items. Before moving on to the formatting characters

themselves let's just dwell on this point for a few moments, with an example to make everything clearer. We have already used the formatting character **A** in the last example, so we carry on with that. This character simply specified the position of an alpha numeric character on the page. For example, suppose we want to print a column from the left-hand edge of the paper. The following will set up the printer ready for formatting any future data:-

```
30 OPEN 2, 4, 2
40 PRINT#2, "  AAAAAAAAAA"
50 CLOSE 2
```

When you have typed these lines in, (not forgetting to **NEW** the computer first of course), you can do a **RUN** and the printer will be ready. You should notice that the printer will appear not to have done anything and you have no way of checking whether your column positioning is correct. If you do the following, however, you will then be in a better position for checking:-

```
30 OPEN 2, 4, 0
40 PRINT#2, "  AAAAAAAAAA"
50 CLOSE 2
```

Now, when you **RUN** the program the printer will actually print out the formatting data in the proper place on the page. Notice that all we have changed is to **OPEN** file #2 with a secondary address of **0** instead of **2**. When you are quite happy with the format of the line, you can simply change line 30 (in this case) back to the original, with a secondary address of **2**, and then the next time the program is **RUN** the printer will be set up correctly. In the following examples we shall be using a rather similar approach to demonstrate the effects of the various formatting characters.

THE FORMATTING CHARACTERS

The formats available and the data to be formatted can be grouped broadly into two categories: numeric and string. Numeric data can be formatted a number of ways, whilst string or alphabetic data can generally only be placed within particular columns.

Format A

As we have just seen, the character **A** allows string fields to be formatted into columns of specific widths at particular positions on the page. Within each field leading spaces are removed, so that the column becomes justified to the left and the end of the field is padded, if necessary, with blanks. If you do not want this left-hand justification at the beginning of each field, you should use shifted spaces (**CHR\$(160)**), since these are not removed. If a field is longer than the specified number of characters for the column, all excess characters will be truncated and lost. This feature can often be used to good advantage when you want to limit the amount of data entered into a form, an invoice for example. As has been mentioned before, each field must be separated by a 'cursor right' character, a **CHR\$(29)**, in order for the formatting to be carried out correctly. Similarly if you want a column to be left empty you must send a field that contains a shifted space only. The example described in **SA=1** displays these features.

Formatting numeric data

There are six ways in which numeric data can be formatted on the CBM 1526 printer, so let's look at each of them in turn.

Format 9

The character **9** is used to specify a digit position within a numeric field. If there is no digit, a blank will be printed instead. Type in this example to see the effect of character **9** upon numeric data. It is a good idea to turn the printer off and then on again first, to remove any previous formatting instructions that may be lurking deep in its dark recesses!

```

10 REM *****
12 REM *                                     *
14 REM *      PRINTER DEMO                 *
16 REM *      NUMERIC FORMAT 9             *
18 REM *                                     *
20 REM *****
30 OPEN4,4
40 OPEN1,4,1
50 PRINT#4,"          99999999"
60 OPEN 2,4,2
70 PRINT#2,"          99999999"
80 FOR T=1 TO 10

```

```

90 PRINT#1,LEFT$( "1234567890",T)
100 NEXT T
110 CLOSE 1
120 CLOSE 2
130 CLOSE 4

```

```

99999999
 1
 12
 123
 1234
 12345
 123456
 1234567
 12345678
*****
*****

```

There are several points illustrated by this program and its resulting printout. The first thing to observe is that the column of figures is justified to the right, just as numeric data should be. However, if you were to try the same example using a secondary address of zero, (that is to say, no formatting), you would see that the printer normally justifies everything to the left. A second interesting point to notice is that if the number to be printed is too large to fit within the designated space, the printer prints a row of asterisks. This feature, although it may seem unimportant, is actually very useful for highlighting possible errors within printouts. Try telling your VAT inspector that your returns are out by a factor of *100* because your printer casually knocked off the last two digits of the total and you will soon realise the value of this built-in check!

While the character **9** is used to format numeric data, you should be aware that this data can actually be held within a string variable as well as within a numeric variable. For example, if you were to alter the numbers in line 70 to "**1234ABCDEF**" the printer would pick out the preceding numbers and format them whilst ignoring the following letters completely. In fact, the printer acts just as if a **VAL** command has been performed upon the string of characters.

One final point to note and remember is the effect of sending alphabetic data when numeric data is expected. Change the characters in line 70 to "**ABCDEFGHIJ**" and **RUN** the program

again and you will see that this data is accepted, but treated in a strange and unlikely way. It is all printed out and justified to the left from the beginning of the specified column. However, it is not true justification, since any length of data will still be printed in full. In this case the formatting character acts like a **TAB** and no more, which could have disastrous effects upon the layout of your document!

Format Z

The formatting character **Z** specifies a digit position within a numeric field. In operation it acts in a way very similar to the character **9** just covered. The difference between the two is that if there is no numeric digit to be printed at the position, the **Z** forces a **zero**, whereas the **9** forces a space to be printed. In all other respects the two characters follow identical rules of formatting. This next demonstration is an expansion of the previous program to include both the characters **9** and **Z**:-

```

10 REM *****
12 REM *                                     *
14 REM *          PRINTER DEMO             *
16 REM *      NUMERIC FORMAT Z             *
18 REM *                                     *
20 REM *****
30 OPEN4,4
40 OPEN1,4,1
50 PRINT#4,"          99999999  ZZZZZZZZ"
60 OPEN 2,4,2
70 PRINT#2,"          99999999  ZZZZZZZZ"
80 FOR T=1 TO 10
90 PRINT#1,LEFT$("1234567890",T),
100 PRINT#1,LEFT$("1234567890",T)
110 NEXT T
120 CLOSE 1
130 CLOSE 2
140 CLOSE 4

```

```
99999999  ZZZZZZZZ
      1  00000001
     12  00000012
    123  00000123
   1234  00001234
  12345  00012345
 123456  00123456
1234567  01234567
12345678 12345678
*****
*****
```

When you **RUN** this program you should see a printout the same as the one shown here. Just as with the last example, try changing lines **90** and **100** to “**1234ABCDEF**”. When you do, you will discover something which at first appears strange and unpredictable, like the printout shown here:-

```
99999999  ZZZZZZZZ
      1  00000001
     12  00000012
    123  00000123
   1234  00001234
  1234  A          1234A
  1234  AB         1234AB
  1234  ABC        1234ABC
  1234  ABCD       1234ABCD
  1234  ABCDE      1234ABCDE
  1234  ABCDEF     1234ABCDEF
```

So, what is happening to our nicely formatted columns of figures? The cause of the apparent disorganisation is, of course, the mixing of numeric and alphabetic data. The first column is in all cases formatted correctly, but the printer is treating the alphabetic characters within the first field (created by line **90**) as the second field for formatting. Since they are alphabetic data the formatting fails and they are simply justified to the left within the second column. Where does the third column come from? At the end of line **90** there is a comma, necessary to separate the two numeric fields, and this is responsible for creating the third (slanting) column of data. Both formatting columns have been used by the **PRINT**

instruction of line 90 (on the fifth and subsequent executions of the loop) so the comma is performed, to move the print head along the carriage by 10 character positions. Finally, the **PRINT** instruction of line 100 is performed in full because there are no more formatting instructions to follow. The end result: chaos! The solution lies in your own hands; you must always ensure that any data transmitted to the printer is of the type that it has been instructed to expect. Later in this chapter we shall be looking at the abilities of the printer to analyse and supply diagnostic messages for situations such as this where the wrong type of data is transmitted.

Format \$ (dollar sign)

When used as a formatting character the \$ has two distinct but similar functions. In both cases a dollar sign is assigned to the numeric data being formatted, thus immediately transforming a column of numeric data into a column of financial data.

A single dollar sign, when placed at the start of a field formatting string of characters, will force a dollar sign to be printed in the first column of that field. As you can see from the next example, placing the \$ in front of a set of 9's causes the printer to justify the dollar sign to the left whilst still printing the numeric data ranged to the right, with blank spaces filling any gaps that do not contain digits. Similarly, when it is used with the **Z** symbol, the \$ is justified to the left, although in this case zeros fill the otherwise unoccupied column positions.

```

10 REM *****
12 REM *
14 REM *      PRINTER DEMO      *
16 REM *      NUMERIC FORMAT $  *
18 REM *
20 REM *****
30 OPEN#4,4
40 OPEN#1,4,1
50 PRINT#4,"          $99999999    $ZZZZZZZZ "
60 OPEN 2,4,2
70 PRINT#2,"          $99999999    $ZZZZZZZZ "
80 FOR T=1 TO 10
90 PRINT#1,LEFT$( "1234567890",T),
100 PRINT#1,LEFT$( "1234567890",T)
110 NEXT T
120 CLOSE 1
130 CLOSE 2
140 CLOSE 4

```

\$99999999	\$ZZZZZZZZ
\$ 1	\$00000001
\$ 12	\$00000012
\$ 123	\$00000123
\$ 1234	\$00001234
\$ 12345	\$00012345
\$ 123456	\$00123456
\$ 1234567	\$01234567
\$12345678	\$12345678
\$*****	\$*****
\$*****	\$*****

Notice especially that the helpful asterisks are still printed if the column size is exceeded by the numeric data.

The dollar sign can also be used as a replacement for the **9** symbol. When this format is in use, the number is still justified to the right but now the dollar sign immediately precedes the number, rather than being in a specified column. You can demonstrate this for yourself by altering lines 50 and 70 so that they both contain the string “**\$\$\$\$\$\$\$\$ \$ZZZZZZZ**” or even “**\$\$\$\$\$\$\$\$ \$\$\$\$\$\$\$\$**”. When you **RUN** this modified version you will see the newly justified result of the changed formatting instructions.

If you want to use a pound sign instead of the dollar, see the section in this chapter on printing literals within data fields.

Format **S**

The function of the formatting character **S** is to ensure that the arithmetic sign of a number is also printed in the specified column position.

A demonstration of this simple formatting character is all that is necessary to make its use clear. Type in this modification of the previous program to see how the printout can be altered by using the **S** character.

```

10 REM *****
12 REM *
14 REM *      PRINTER DEMO      *
16 REM *      NUMERIC FORMAT S  *
18 REM *
20 REM *****
30 OPEN4,4
40 OPEN1,4,1
50 PRINT#4,"          S99999999  SZZZZZZZZ "
60 OPEN 2,4,2
70 PRINT#2,"          S99999999  SZZZZZZZZ "
80 FOR T=1 TO 10
90 PRINT#1,LEFT$("1234567890",T),
100 PRINT#1,LEFT$("1234567890",T)
110 NEXT T
120 CLOSE 1
130 CLOSE 2
140 CLOSE 4

```

```

S99999999  SZZZZZZZZ
+      1    +00000001
+     12    +00000012
+    123    +00000123
+   1234    +00001234
+  12345    +00012345
+ 123456    +00123456
+1234567    +01234567
+12345678    +12345678
+*****    +*****
+*****    +*****

```

Unlike the dollar character, the **S** will only allow the arithmetic symbol to be printed in the position specified, not immediately preceding the number itself unless the size of the number forces it so to do.

Format . (decimal point)

By using a full stop, (or decimal point), within a formatting string you can determine the position of the decimal point within the numeric data that is to be formatted. When this is the case, data will be formatted around the fixed position of the decimal point. Using a variant of the, by now familiar, demonstration program we can

see how this formatting character operates:-

```

10 REM *****
12 REM *
14 REM * PRINTER DEMO *
16 REM * NUMERIC FORMAT . *
18 REM *
20 REM *****
30 OPEN4,4
40 OPEN1,4,1
50 PRINT#4," 999999.999 ZZZZZZ.ZZZ "
60 OPEN 2,4,2
70 PRINT#2," 999999.999 ZZZZZZ.ZZZ "
80 FOR T=1 TO 10
90 PRINT#1,VAL(LEFT$("1234567890",T))/23,
100 PRINT#1,VAL(LEFT$("1234567890",T))/23
110 NEXT T
120 CLOSE 1
130 CLOSE 2
140 CLOSE 4

```

```

999999.999 ZZZZZZ.ZZZ
      .043 000000.
      .521 000000.
      5.347 000005.
      53.652 000053.
      536.739 000536.
      5367.652 005367.
      53676.826 053676.
      536768.609 536768.
      *****
      *****

```

As you can see, the most significant difference between the use of the decimal point with the 9 and the 7 is that the **Z** will not print anything after the decimal point, which is not very helpful where accuracy is the prime consideration. However, this difference can be used to good effect when you require figures to be rounded down to whole numbers; it achieves the equivalent of performing an **INT**egerization upon the data. If total accuracy is needed you

should use the **9** character to determine the size of the formatting field and then specify as many significant decimal digits as necessary.

Format - (minus sign)

The minus sign is used within a formatting string to specify that a trailing sign should be added to the numeric field. In such a case, if the number is positive a blank will be printed and if the number is negative a minus sign will be included. It is not possible for a numeric field to have both a leading and a trailing sign, so if both are specified only the leading sign will be honoured and the trailing sign ignored.

USING LITERALS WITHIN FORMAT STRINGS

In this context literals are characters that are to be printed exactly as they look in the formatting string, rather than acting as formatting characters for other data. That is to say, you can add characters to the format string that will be printed on the page in the exact position as was specified within the format string. There are very few restrictions applying to the use of literals, mainly that you cannot print reversed characters and you can only print characters that are within the currently selected character set.

Although you can use practically any other character you can only make use of literals in association with numeric data, not with string data.

Literals are added to the formatting string by preceding them with a 'reverse on' symbol (**CHR\$(146)**). One particularly fortunate use of literals for those of us in Great Britain is the printing of the pound sign prior to financial data. This can be achieved quite easily, as this example demonstrates:-

```

10 REM *****
12 REM *                               *
14 REM *      PRINTER DEMO           *
16 REM *      LITERALS                *
18 REM *                               *
20 REM *****

```

```

30 OPEN1,4,1
40 OPEN 2,4,2
50 OPEN 4,4
60 PRINT#4," 99999999  ZZZZZZZZZ "
70 PRINT#2," 99999999  ZZZZZZZZZ "
80 FOR T=1 TO 10
90 PRINT#1,LEFT$( "1234567890",T)
100 NEXT T
110 CLOSE 1
120 CLOSE 2
130 CLOSE 4

```

The other numeric formatting characters still perform their designated actions: the **Z** forcing leading zeros and the **9** forcing the inclusion of leading blanks. This means that the literal (in this case the pound sign) is justified to the left whilst the numeric data is lined up to the right. This differs from the use of the dollar sign where, when more than one occurs within a formatting string, the value sign can be justified to the right, immediately next to the data.

SA=3 - NUMBER OF LINES PER PAGE

The secondary address of **3** is used to set the number of lines that will be printed on each sheet of paper. A feature of the CBM 1526 is the option to skip over the perforations between sheets, leaving three lines blank at the bottom of each page and a further three lines at the top of the next. When this option is used the printer runs off 60 lines on each page plus a total of six blank lines equalling 66 lines per page. By using the facility of secondary address 3 you can alter the extent of your text to accommodate varying sizes of paper, within a range of 13 to 128 lines per page. The number of lines can be set as follows - where 'line no' is the total number of lines per page.

```

10 OPEN 3, 4, 3
20 PRINT#3, CHR$(line no)
30 CLOSE 3

```

When performing a skip over the perforation remember that there will always be 6 lines unused at the top and bottom of each sheet.

SA=4 - ENABLE THE PRINTING OF DIAGNOSTIC MESSAGES

SA=9 - DISABLE THE PRINTING OF DIAGNOSTIC MESSAGES

Although the numbers of these two secondary addresses are not strictly consecutive, they complement each other in function so we shall deal with them together.

A secondary address of 4 is used to turn on, or enable, the diagnostic messages that are generated by the 1526 printer. These highlight an error condition. The 1526 can generate six messages in all and each one is printed out immediately after a particular error condition has occurred. This demonstration program deliberately creates the necessary conditions for all these error messages to be generated. Type in the program and **RUN** it and then we shall look at each error individually:-

```

10 REM *****
12 REM *
14 REM * PRINTER DEMO *
16 REM * ERROR MESSAGES *
18 REM *
20 REM *****
22 :
30 OPEN 14,4,4:PRINT#14:CLOSE 14
40 OPEN 3,4,3:PRINT#3,CHR$(128):CLOSE 3
50 OPEN12,4,12:PRINT#12:CLOSE 12
60 OPEN2,4,2: OPEN1,4,1
70 PRINT#2,"9999999"
80 PRINT#1,"ABCDEFG"
90 PRINT#1,"123E3"
100 PRINT#2," AWAHAHA"
110 OPEN4,4
120 PRINT#4,"HELLO";:PRINT#2,"AAA"
130 CLOSE 1
140 CLOSE 2
150 CLOSE 4
READY.

```

★PE:L★- This message is generated whenever an attempt is made to set the number of lines per page to a value that is outside the accepted range of 13 and 128 lines. The attempted change is ignored and the previous line setting remains operative.

★PE:C★- Bad command. This message is generated whenever an attempt is made to give the printer an invalid secondary address number. Although the error message is printed the command is otherwise ignored.

★PE:M★- Data-format mismatch. An attempt has been made to print non-numeric data in a numeric field. The first character printed after the error message is the culprit.

★PE:E★- Exponent error. Numeric data can be sent to the printer in a format known as exponential, or *scientific notation*. This system is usually reserved for condensing very high or very low numbers that would otherwise contain a large number of zeros. Essentially, this type of numeric value is one where the number is broken down into two parts: a mantissa and an exponent. The first part, the mantissa, must have a value less than 10 and can be either an integer or a decimal fraction. The exponent is an integer value to the power of ten, which when multiplied by the mantissa equals the original real number. For example 1,500,0001.5★10^6 which in form is 1.5E+06. The 1526 printer expects exponential data in this exact format and will produce an error message if the exponent is not a sign followed by two digits.

★PE:F★- Bad format. The formatting data that has been sent to SA=2 is incorrect. Either there are invalid or illegal characters within the formatting string, or the whole instruction has an unrecognizable syntax. The entire formatting string is printed out after this error message and a pointer locates the first offending character.

★PE:T★- Terminator error. An attempt has been made to use another secondary address before a line terminator has been detected. A terminator is either a carriage return (CHR\$(13)), a line feed (CHR\$(10)) or a combination of the two. You cannot 'talk' to two secondary addresses simultaneously. Communications with the first must be terminated before communicating with the second.

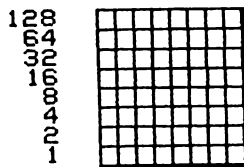
You can see from the relevant blocks of code in the program above just how these errors can be created. These diagnostic messages

can prove extremely helpful when planning, testing and debugging your own programs. When all of the bugs have been ironed out, a **PRINT to SA=9** will disable the printing of the messages and allow good clean copy.

SA=5 - PROGRAMMABLE CHARACTER

A feature which expands the versatility of the 1526 printer even further is the secondary address 5, which allows you to create a user-defined character. Although you only have the facility to use a single character, the definition of it can be altered at any time under software control (by your program). The only limitation is that you cannot reprogram until a terminator (**CHR\$(13)** or **CHR\$(10)**) has been detected by the printer. In other words, you cannot have more than one programmed character on the same line except by the overprinting of that line several times.

The 1526 uses a matrix of 8 dots wide by 8 dots high to form all the characters that can be printed; exactly the same size as the computer itself uses. The only difference between the two methods is that the printer defines its characters in columns starting from the left side of the grid, whereas the computer starts at the top of the grid and defines its characters in terms of the bit positions along each row. So, to design your own character for use with the printer you must take a piece of paper and a pencil and draw a grid similar to the one below:-



Fill in the squares to make up your desired shape. When you are happy with your character, add up the columns to form the data. In each column, where a square is filled in you should add the corresponding number to the total for that column, otherwise the value of the square will be **zero**. You should end up with eight items of data which you can then use to define a character within your own programs.

To pass this data to the 1526 printer you must channel it through a file that should have been **OPENed** to secondary address 5, in the form of a character string, not numeric data. For example, if you had created a tank shape the following routine would do this and also print the shape through the printer:-

```

10 REM *****
12 REM *
14 REM *      PROGRAMMABLE      *
16 REM *      CHARACTER        *
18 REM *      DEMO PROGRAM      *
20 REM *
22 REM *****
24 :
30 OPEN5,4,5
40 FOR COUNT=1 TO 8
50 READ X
60 A$=A$+CHR$(X)
70 NEXT COUNT
80 PRINT#5,A$
90 OPEN 4,4
100 PRINT#4,"THE PROGRAMMED CHARACTER IS "
110 FOR T=1 TO 5
120 PRINT#4,CHR$(254);CHR$(32);
130 NEXT T
140 PRINT#4
150 PRINT#4,"DATA=";
160 RESTORE
170 FOR T=1 TO 8
180 READ X
190 PRINT#4,X;" ";
200 NEXT T
210 CLOSE 5
220 CLOSE 4
1000 DATA 0,62,28,252,28,62,0,0
READY.

```

The programmable character is character 254 (**CHR\$(254)**) and it can be printed whenever you choose simply by specifying **PRINT#4,CHR\$(254)**, as in line 120 in the demonstration program.

It is only possible to use a single programmed character within each line of printout. The printer must perform an end-of-line

operation before the user-definable character can be redefined. However, as we shall see later, there is a method of printing a new line without activating the line feed process. This can be achieved by **PRINT#,CHR\$(141)**, which will then allow you to overprint characters on the same line of printout. This procedure will allow you, with an awesome amount of programming, to simulate the dot-addressable programming capabilities of more expensive printers, but the process will be very, very slow!

Whilst on the subject of slow processing, creating user-defined characters on pieces of paper can only be described as a tedious task, especially as there is a computer sitting there doing nothing at the same time. The next program is to allow you to create characters for your 1526 printer. The running process should be familiar by now; the end result in this case is a printout of the full-sized character on paper plus a list of the data needed to recreate the shape. Type the following program in, and **SAVE** it onto disk and without doubt you will save yourself both pencil and paper. All the controls are the same as in the earlier program and it performs the same function of recreating characters (to disk), except the details will not be saved onto a disk:—

```

10 REM *****
11 REM *
12 REM *   CHARACTER DEFINER   *
14 REM *       1526 PRINTER   *
16 REM *
18 REM *****
19 :
20 AT$=CHR$(17):FORT=1T05:AT$=AT$+AT$:NE
XTT:AT$=CHR$(19)+AT$
25 DIMCHAR(8,8),X$(25)
100 GOSUB 1000
110 GOSUB 1100
120 :
1000 REM *****
1002 REM *
1004 REM *   DRAW THE GRID   *
1006 REM *
1008 REM *****

```

```

1009 :
1010 PRINTCHR$(147)"    C H A R A C T E R
      D E F I N E R"
1015 PRINT:PRINT"          ";CHR$(18);" COM
MODORE 1526 PRINTER "
1020 PL$="TTTTTTTT"
1030 FORT=0T07:PRINTLEFT$(AT$,8+T)SPC(15
-LEN(STR$(2↑(7-T))))2↑(7-T)
1035 PRINTLEFT$(AT$,8+T)SPC(16)PL$:NEXTT
1040 PRINTLEFT$(AT$,16)SPC(16)"-----"
1050 PRINTLEFT$(AT$,17)SPC(16)"76543210"
1052 PRINTLEFT$(AT$,7)SPC(25)"    CONTRO
LS"
1055 PRINTLEFT$(AT$,8)SPC(25)"F1=DRAW/UN
DRAW"
1060 PRINTLEFT$(AT$,9)SPC(25)"F3=UPDATE
DATA"
1065 PRINTLEFT$(AT$,10)SPC(25)"F5=CREATE
"
1070 PRINTLEFT$(AT$,11)SPC(25)"    CHARA
CTER"
1075 PRINTLEFT$(AT$,12)SPC(25)"F7=QUIT "
1080 PRINTLEFT$(AT$,14)SPC(25)"CURSOR KE
YS TO"
1085 PRINTLEFT$(AT$,15)SPC(25)" MOVE "
1095 GOSUB1200
1097 RETURN
1099 :
1100 REM *****
1102 REM *                      *
1104 REM *   MOVE THE CURSOR   *
1106 REM *                      *
1108 REM *****
1109 :
1110 Y=8:X=16:CH$=CHR$(111)
1150 PRINTLEFT$(AT$,Y)SPC(X)"* "CHR$(157)
;:POKE204,0
1155 BIT((Y-7),(23-X))=FL
1160 GETA$: IFA$="" THEN1160
1165 POKE204,1
1170 IFA$=CHR$(17) THENPRINTLEFT$(AT$,Y)
SPC(X)CH$;:Y=Y-(Y<15)
1175 IFA$=CHR$(145) THENPRINTLEFT$(AT$,Y
)SPC(X)CH$;:Y=Y+(Y>8)
1180 IFA$=CHR$(29) THENPRINTLEFT$(AT$,Y)
SPC(X)CH$;:X=X-(X<23)

```

```

1185 IFA$=CHR$(157) THENPRINTLEFT$(AT$,Y
)SPC(X)CH$;X=X+(X>16)
1190 IF FL=0THEN IFA$=CHR$(133)THENCH$=C
HR$(18)+CHR$(32)+CHR$(146):FL=1:GOTO1197
1192 IF FL=1THEN IFA$=CHR$(133)THENCH$=C
HR$(111):FL=0:GOTO1197
1194 IFA$=CHR$(134)THENGOSUB1200
1195 IFA$=CHR$(135)THENGOSUB1300
1196 IFA$=CHR$(136)THENPRINTCHR$(19);:EN
D
1197 IFA$=CHR$(140) THEN RUN
1198 GOTO1150
1199 :
1200 REM *****
1202 REM *
1204 REM *      UPDATE DATA      *
1206 REM *
1208 REM *****
1209 :
1210 PRINTLEFT$(AT$,7)SPC(8)"DATA"
1220 FORT=0T07
1230 DTA(T)=0
1240 FOR BIT=1T08
1250 DTA(T)=DTA(T)+(2↑(8-BIT))*BIT(BIT,T
)
1260 NEXT BIT
1270 PRINTLEFT$(AT$,15-T)SPC(8)DTA(T)
1280 NEXTT
1290 RETURN
1299 :
1300 REM *****
1301 REM *
1302 REM *      CREATE A NEW      *
1304 REM *      CHARACTER        *
1306 REM *
1307 REM *****
1309 :
1310 GOSUB1200:B$=""
1320 OPEN5,4,5
1330 FORT=0T07
1340 B$=B$+CHR$(DTA(7-T))
1350 NEXT T
1360 PRINT#5,B$:CLOSE5
1370 OPEN4,4
1380 PRINT#4,"THE PROGRAMMED CHARACTER I
S ";

```

```

1385 PRINT#4,CHR$(254);CHR$(32);:FORT=1T
010:PRINT#4,CHR$(254);:NEXTT:PRINT#4
1390 PRINT#4,"DATA:-";:FORT=0T07:PRINT#4
,DTA(7-T);
1395 IF T<7 THEN PRINT#4,",";
1397 NEXTT:PRINT#4:CLOSE4
1399 RETURN

```

SA=6 - SPACING BETWEEN LINES

The secondary address of **6** is used to alter the line feed spacing between each line of text on the printed page. The default value is **36** which produces 6 lines per inch (although the official manual claims that the value is 24!). The spacing between lines can easily be changed by sending a new value to the secondary address in the form of a character. For example, to change the spacing so that there is no space between lines, try the following:-

```

10 REM *****
12 REM *
14 REM *          PRINTER DEMO          *
16 REM *              SA=6              *
18 REM *
20 REM *****
22 :
30 OPEN 4,4
40 OPEN 6,4,6
50 PRINT#6,CHR$(20)
60 FOR T=1 TO 5
70 PRINT#4,"NNNNNNNNNNNNNNNNNNNNNNNN"
80 NEXT T
90 CLOSE 6
100 CLOSE 4
READY.

```

The value that is transmitted to secondary address **6** must be within the range of 1 to 127 for there to be any linefeed. Outside this range no linefeed is generated, a fact that can be used to create normal-sized emphasised print for highlighting particular lines within your document by double printing to produce a darker copy. For example:-

```

10 REM *****
12 REM *                                     *
14 REM *          PRINTER DEMO             *
16 REM *          SA=6                     *
18 REM *                                     *
20 REM *****
22 :
30 OPEN 4,4
40 OPEN 6,4,6
50 PRINT#6,CHR$(36)
60 PRINT#4,"THIS IS AN EXAMPLE"
70 PRINT#6,CHR$(0)
80 FOR T=1 TO 2
90 PRINT#4,"OF DOUBLE PRINTING TO GIVE"
100 NEXT T
110 PRINT#6,CHR$(36):PRINT#4
120 PRINT#4,"EXTRA EMPHASIS TO THE TEXT"
130 CLOSE 6
140 CLOSE 4
READY.

```

To alter the degree of emphasis change the number of overprintings that are performed, by altering the loop value in line 70 to suit your own requirements.

This facility can be particularly useful in circumstances where you wish to produce one or more continuous vertical lines down the page, to create an invoice for instance. Whilst the use of formatting strings and literals within them can produce the necessary dividers between columns, the vertical lines will not join up with each other unless the spacing between each line is reduced, using **SA=6** and a setting of **18**. Just doing this, however, will make any printing overlap so you must also insert an otherwise blank line that

contains only the dividing characters between each line of text. As an example of this, try the next small program:-

```

10 REM *****
12 REM *                               *
14 REM *      PRINTER DEMO          *
16 REM *      LITERALS              *
18 REM *                               *
20 REM *****
22 :
30 OPEN 1,4,1:OPEN 4,4
40 OPEN 6,4,6
50 OPEN 2,4,2
60 PRINT#2,"| AAAAA | AAA | AAA | AA
AAAAAA | AAA | AAAAA | AAAAAA"
70 PRINT#6,CHR$(18)
75 FOR T=1 TO 5
80 PRINT#1,"THIS IS AN EXAMPLE";
90 PRINT#1,"OF FORM SETTING"
100 PRINT#4,"| | | |
| | | |"
110 NEXT T

```

SA=7 - SELECT UPPER/LOWER CASE SA=8 - SELECT ASCII/GRAPHICS

On Commodore printers there are two complete character sets, just as there are on the Commodore range of computers. On the computer, each set can be selected by pressing the **CBM** logo key and the **SHIFT** key simultaneously and the screen display will switch, or toggle, between the two. If you haven't already discovered this feature of your computer why not try it now?

The same effect can be achieved with the printer by sending a **PRINT#** to either the secondary address **7** to select the upper/lower case character set or secondary address **8** to select the normal, default set. When using this method of switching, you must complete an entire line of print before the set can be changed again, although as we shall see in the next section it is possible to mix character sets on the same line.

```

10 REM *****
12 REM *
14 REM * PRINTER DEMO *
16 REM * UPPER/LOWER CASE *
18 REM *
20 REM *****
30 OPEN4,4
40 OPEN3,4,7
50 PRINT#4,"THIS IS UPPER CASE"
60 PRINT#3,"THIS IS LOWER CASE"
70 CLOSE 3
80 CLOSE 4

```

SA=9 - SUPPRESS DIAGNOSTIC MESSAGES

Sending a **PRINT#** to secondary address **9** will turn off the printer's ability to print diagnostic and error messages. Refer to the appropriate section concerning **SA=4**, where there is a full explanation of the messages that are generated by the CMB 1526 printer.

SA=10 - RESET THE PRINTER

The secondary address of **10** is used to reset the printer to its original power-on condition. It is always advisable to perform this function between different, complex formatting exercises so as to ensure that previous instructions to the printer do not carry over and interfere with the new formatting layout. Failure to do so may well cause unexpected and unwanted results to appear on your nicely laid-out document!

CONTROL CHARACTER FUNCTIONS

Perhaps the most flexible formatting capability of the CBM 1526 printer is the control that is possible for you to exercise using the particular control codes that can be actually buried within the text to be printed out, rather than sent to a particular secondary address as a separate command. In this section we shall be looking at all these control characters. We have checked and tested them thoroughly and found them to be as stated here.

ENHANCED CHARACTERS

The standard-sized characters on the 1526 are composed around a matrix of dots 8 wide by 8 high. You can also use four other sizes; in each case the height is the same but the width increases to 16, 24, 32 and 40 dots. To set each level of emphasis you must **PRINT#** a **CHR\$(14)** (not **CHR\$(1)** as stated in the manual), so to set characters 24 wide you need to send two **CHR\$(14)**.

Character 14 can be placed within quotation marks by pressing the **CTRL** key and the **N** key simultaneously, giving a reversed video **N** on the screen. The emphasised mode is cancelled whenever a carriage return is received from the computer or when the printer must perform it's own line feed because of a full buffer.

You can also cancel the emphasised mode within a single line by sending **CHR\$(15)**; alternatively this can be embedded within quotation marks by using **CTRL** and **O** simultaneously. This next program demonstrates all the modes of emphasis and finally how to mix different levels of emphasis within the same line of print.

```

10 REM *****
12 REM *   EMPHASISED MODE   *
14 REM *****
16 :
20 OPEN#4,4
30 PRINT#4,"HELLO"
40 PRINT#4,"HELLO"
50 PRINT#4,"HELLO"
60 PRINT#4,"HELLO"
70 PRINT#4,"HELLO"
80 PRINT#4,"HELLO"
90 CLOSE 4
READY.
```

```

HELLO
HELLO
HELLO
HELLO
HELLO
HELLO
```

INDIVIDUAL UPPER OR LOWER CASE LETTERS

Whilst it is possible to switch between the two built-in character sets by printing to either secondary address **7** or **8**, this method can only be performed at the beginning of a printing line, never in the middle.

If you want to mix character sets on a single line you can embed a control character within the text itself. Switching to the upper/graphics set is achieved by using **CHR\$(145)**, the cursor/up symbol and lower/upper case is obtained by using **CHR\$(17)**, the cursor/down symbol.

```

10 REM *****
12 REM * UPPER/LOWER CASE *
14 REM *****
16 :
20 OPEN4,4
25 FOR T=1 TO 4
30 PRINT#4," HELLO HELLO HELLO"
35 NEXT T
40 CLOSE 4

```

SKIP OVER PAGE PERFORATIONS

We have already seen that the secondary address **3** can be used to set the number of lines per page. Unless paging is turned on, it will have no effect, however, because nothing will happen to distinguish between the pages! You can turn paging on by printing **CHR\$(147)**, the clear screen symbol. The most frequent use of paging is to skip over the perforations of the continuous stationery used by printers. You must be careful when using paging, though, because the page breaks are determined by the number of lines on the page, not by the size of the page. Thus, if at any time you alter the spacing between lines (using **SA=6**), you will also alter the page length, usually detrimentally. To turn paging off, use **CHR\$(19)**, the cursor/home symbol.

REVERSED CHARACTERS

When constructing screen displays on your Commodore computer you have probably discovered that you can print characters in reverse or normal video quite simply, by adding the 'reverse/on' or 'reverse/off' characters within the body of the text to be printed to the screen. The process is easy and we are sure that it need not be elaborated upon at this juncture. Exactly the same principle should be followed if you want to print characters on paper in reverse or normal print. To make the 1526 print in reverse text just precede the text to be printed with the 'reverse/on' code, (**CHR\$(18)**), and to cancel it use the 'reverse/off' symbol, (**CHR\$(146)**). A carriage return will automatically cancel reverse mode, just as it does when being used to the screen. This is shown in the following program:-

```

10 REM *****
12 REM *                                     *
14 REM *      PRINTER DEMO                *
16 REM *      REVERSED CHARS              *
18 REM *                                     *
20 REM *****
30 OPEN4,4
40 PRINT#4,"NORMAL CHARACTERS"
50 PRINT#4,"REVERSED CHARACTERS"
60 PRINT#4,"MORE REVERSED CHARACTERS"
70 CLOSE 4

```

SHIFTED CARRIAGE RETURN

Under normal circumstances, a carriage return sent to the printer (at the end of a line of text) will also generate a line feed so that the printer can then continue on a fresh line. A **SHIFT**ed carriage return, however, will not generate a line feed but will in all other respects act as a normal carriage return. That is to say, it will cancel all control characters that are currently in operation. The **SHIFT**ed carriage return can, therefore, be used in those special circumstances where you wish to overprint upon the same line of text, for instance particular parts of the line by overprinting, or to change secondary addresses within the same printed line. Alternatively, you might

want to alter the spacing between the lines, but for only part of the current line. An alternative to this code is to set the line spacing to 0 or >127, which has mostly the same effect except that, in this case, the paging will remain valid and correct.

On the CBM 1526 you have two levels of control over the format of the printout; secondary addressing which can be thought of as providing overall command of the printer and control characters which only have effect within a single line of text. If you choose the required combination of these facilities carefully you can create any form of printout that you wish. You can even make the printer produce detailed graphs or screen dumps by judicious and expert use of the programmable character facility, but the programming necessary to do so will keep you occupied for many a long, dark night (although the results can be very rewarding and satisfying!). The summary table below, along with the explanations in this chapter should allow you to get the very best from your investment and work the printer to its fullest capabilities.

Special Control Character

Printer function	Code	Keyboard
Enhance	CHR\$(14)	CTRL + N
Unenhanced	CHR\$(15)	CTRL + O
Paging on/reset	CHR\$(147)	SHIFT + CLR HOME
Paging off	CHR\$(19)	CLR HOME
RVSON	CHR\$(18)	CTRL + RVS ON
RVS OFF	CHR\$(146)	CTRL + RVS OFF
Carriage return	CHR\$(13)	RETURN
Carriage return with no line feed	CHR\$(141)	NA
Line feed	CHR\$(10)	+ J
Uppercase	CHR\$(145)	SHIFT + CURSR Up
Lowercase	CHR\$(17)	CURSR Down
Skip space	CHR\$(29)	CURSR Right

12

Using the Epson RX-80 Printer

Normally, when using a Commodore computer, one would choose to use a Commodore printer with it because they are designed to be compatible with each other. However, in some circumstances, you may decide to use another type of printer with your computer. This may be because you already have another printer and don't want to spend the extra money on buying a new one, you may need letter quality printing if you are using it for word-processing (the Commodore printers are dot-matrix) or speed may be an important factor in your particular application.

The type of printer that you decide to buy will depend to a great extent on what you want to use it for. There are literally hundreds to choose from, varying in price from about £40 to well over £2000. Again, it depends on what you want to do with it and to a great extent, on the depth of your pocket. We have discussed the various types in Chapter 9, and in this chapter we shall look at the popular and widely-used family of Epson printers, using the RX 80 as an example to show the broad differences between Commodore and other printers.

The Epson range of printers supports a number of different features, including a choice of several sizes for characters which can be altered by sending appropriate control characters from the computer. The two main ones are elite sized (12 characters per inch) and pica sized (10 characters per inch), but these may be enlarged or condensed by other control codes, giving a wide choice of different settings.

One of the most popular in this range is the Epson RX-80, which is a dot matrix printer offering some very advanced features such as condensed print, alternative character sets and even the ability to print in high resolution (bit-mapped mode). The examples in this

chapter were all tested on an Epson RX-80, but the same general principles will apply to other printers, provided that the same features are available on them.

THE EPSON RX-80

The printer comes with a Centronics interface built in, so first you must find a Commodore to Centronics interface before you can start using it. The Aztec interface and software mentioned previously is reasonably priced and seems to be freely available in most computer shops. See also Chapter 9 for more information about interfaces.

The printer manual begins by showing you how to connect it to your computer – all you actually have to do is to connect the output of your interface to the socket on the left hand side of the printer at the back and connect the interface to the computer according to the instructions supplied with it.

Once the printer is connected to the computer, they can then both be turned on and we are ready to go.

LOGICAL FILES – Talking to the printer

Commodore computers use a rather peculiar system for communicating with external devices, where data, including that for the printer, is sent out on the data lines as if it were going to a disk or tape file. Each device is allocated a device number, the keyboard being 0, the screen 3, the tape deck 1 and a disk drive normally 8. In this system, a printer is usually device number 4, so information sent to device 4 will appear in some form on the printer.

In order to set the communication system up we must open a logical file to the printer, using the **BASIC OPEN** command, like this:–

OPEN "LFN,DN"

where **LFN** is the Logical File Number and **DN** is the Device Number. With Commodore printers another parameter can be used, the Secondary Address, to set up some special functions but with a non-Commodore printer this is not employed.

A typical **OPEN** statement for the printer could be **OPEN 3,4**. When we want to send data to the printer, we must send it to logical file number 3. This system may seem a little odd, but it gives the computer the ability to have more than one communication channel open to a device at the same time, which is very useful when you are dealing with disk files and want to define characters, enable the error message function and so on.

Now we have a communication channel **OPEN** to our printer, how do we send data to it? Well, we use the by now familiar **PRINT#** command, in the following syntax:

PRINT# "LFN,DATA"

where "**DATA**" is the information that we want to print. Note that the abbreviation for '**PRINT#**' is '**pr**', not '**?#**' as you might expect. Try typing in this example program with your computer and printer connected up:

```
10 OPEN 3,4 : REM OPEN FILE NUMBER 3 TO
   DEVICE 4 (THE PRINTER)
20 PRINT#3,"HELLO WORLD!!"
30 CLOSE 3:END
```

Line 10 **OPENS** a logical file to the printer and line 20 sends the text that is to be printed. The **CLOSE** in line 30 tells the computer and printer that that channel is not wanted any more, so the printer prints out anything that it has been sent but hasn't yet printed, and the communication channel is then **CLOSED**.

When you **RUN** the program, the printer should print 'Hello World!!' If it doesn't, you should check the connections and that the printer '**ON LINE**' light is on: some interfaces will hold the computer up if the data is not accepted by the printer, while other systems may just ignore it. Some printers acknowledge that they have received data by a method called *handshaking*, where the printer sets one of the wires in the cable to a known state, for the

interface to detect. If this is not done, the computer may be held up so that data is not lost. This feature is important if you are using a slow printer, where the computer may end up sending data faster than it can be printed – the printer effectively says ‘Oy! Hang on a minute while I catch up!’.

Apart from the **PRINT#** command for sending information to the printer, characters that would normally be printed on the screen by **PRINT** statements can be diverted to the printer by using the **CMD** command from **BASIC**. (See also Chapter 4) A Channel must have been previously **OPENed** to the printer, as in this example:–

```
10 OPEN 3,4
20 CMD 3
30 PRINT "HELLO WORLD!!"
40 PRINT#3
50 CLOSE 3
```

Notice that the string which would normally have been **PRINTed** on the screen is sent to the printer instead. Line 40 has a straight **PRINT#3** which is executed before the file is **CLOSEd**. This forces any characters in the printer’s buffer to be printed and tells the computer that any further **PRINT** statements are to be sent to the default device, the screen. This process is known as *unlistening*, because it tells the printer to stop listening to the computer and printing out what it ‘hears’ through the interface and cable.

Data and commands can be sent to the printer by either of these two methods, depending on what you are doing. For example, when you **LIST** a program it is normally displayed on the screen, by using **CMD** you can send the output to your printer. This is done by issuing the following commands in direct mode, that is without line numbers:–

OPEN 3,4:CMD3	[Set up the printer as the default device]
LIST	[LIST the program in memory]
PRINT#3:CLOSE 3	[Unlisten the printer and CLOSE the file]

The printer should spring into action, **LIST** the program out and then stop. Note that on non-Commodore printers, the graphics characters will not be printed out correctly. For this reason, you may want to use a program like a brackets utility, which checks for these characters and prints them out as a word enclosed in square brackets, for example the 'cursor down' character, normally displayed on the screen as a reverse field 'Q', is converted to **[DOWN]**.

CONTROL CODES

On the Epson RX-80, as with the Commodore 1526, certain characters act as formatting commands rather than as data to be printed. These can be used to control some of the special functions of the printer, such as setting the top of form position, operating the buzzer and selecting the different print modes.

On the Epson machines they are known as *control codes* and in this section we shall be going through those that operate with the RX-80, with examples of how to use them. Some may not apply to you if you are not using an Epson printer, being functions of the printer rather than the computer. You will find instructions in your own printer manual regarding what features are available on it, but even if the codes you use are different the method and general principles will be the same. Note in particular that some manuals contain examples of how to use specific functions written in Microsoft **BASIC**; this uses the command **LPRINT** to send data to a printer. On the Commodore computers, the corresponding instruction is **PRINT#**, as we have already seen, but the commands and data actually sent should be the same.

There are two distinct types of control available on the Epson RX-80. Firstly, some control codes are sent as a single character and perform functions like forcing the printer to feed the paper to the top of the next page (Form Feed), while the other group consists of several items of data that are always preceded by the ESC code, **CHR\$(27)**.

SINGLE BYTE CONTROL CODES

Carriage Return – CR [CHR\$(13)]

This is the most common of the control codes and is used to force the printer to print the data it has been sent. The RX-80 has what is known as a *print buffer*, to which data is sent before printing. Normally, nothing is printed out until the buffer is full, but this condition can be simulated if you want to for some reason by sending a carriage return character. When it receives a carriage return character the following operations will be executed: the printer will immediately print out any data still in its buffer, the paper will be fed by one line and the print head will move to the left-hand side of the page, ready for the next line.

Line Feed – LF [CHR\$(10)]

This control code is similar to the carriage return code in that it too causes any data stored in the print buffer to be printed and the paper to be fed through by one line. In this case, however, the print head is not moved to the left-hand edge of the printer but stays where it was after printing any data. Both **CR** and **LF** cancel the enlarged print mode (see the **SO** code) if it was set. Also, if these commands result in the paper being fed into the area at the bottom of the page normally not printed on (called the perforation skip-over area), the paper will be fed through to the top of the next page.

Form Feed – FF [CHR\$(12)]

When this character is encountered by the printer, any data left in the buffer will be printed and the paper fed through until the next print position is reached at the top of the next sheet. These features are found on most printers and can be used to control the position on the paper where the next line is to be printed.

Shift Out – SO [CHR\$(14)]

This control character is used to print at double width. After an '**SO**' character is received by the printer, subsequent data is printed out in enlarged mode. This mode is automatically turned off when a **CR** or **LF** character is received at the end of a line, or by the **DC4** or **ESC W** codes at any time. See this

demonstration example:

```
10 OPEN 3,4
20 PRINT#3,CHR$(14);
30 PRINT#3,"ENLARGED CHARACTERS"
40 PRINT#3,"NORMAL CHARACTERS"
50 CLOSE3
```

Enlarged Mode Cancel – DC4 [CHR\$(20)]

This code cancels the effect of the **SO** code, resetting the character size to normal. This enables us to mix printing of both normal and enlarged sizes on the same line when using Epson printers. Other printers may or may not support these features, so you should read the manual supplied with your printer carefully. Here is an example demonstrating this enlarged mode reset control.

```
10 OPEN 3,4
20 PRINT#3, CHR$(14); "ENLARGED";
30 PRINT#3, CHR$(20); "  NORMAL"
40 CLOSE3
```

Shift In – SI [CHR\$(15)]

This code enables printing in condensed mode, with a spacing of 17 characters per inch. Some of the print modes can be mixed. For example, it is possible to print in both condensed and enlarged modes simultaneously when pica-sized characters are selected, (as in the example below), but care should be taken when using elite-sized characters because although the condensed code is accepted the characters are not actually condensed until the elite mode is turned off.

```
10 OPEN 3,4
20 PRINT#3,CHR$(15); "CONDENSED MODE"
30 PRINT#3,"STILL IN CONDENSED MODE"
40 PRINT#3,CHR$(14); "NOW WE'RE IN CONDENSED AND ENLARGED MODE"
50 CLOSE3
```

Condensed Mode Cancel – DC2 [CHR\$(18)]

This code cancels the condensed mode set up by 'SI' code. When you are using elite-sized characters, the actual characters are not altered at all; an internal flag is set inside the printer and the effect only becomes apparent after the elite mode is cancelled using the **ESC P** code (see later).

```
10 OPEN 3,4
20 PRINT#3,CHR$(15);"CONDENSED MODE PRIN
TING"
30 PRINT#3,CHR$(18);"NOW WE'RE BACK TO N
ORMAL PICA-SIZED PRINTING"
40 CLOSE3
```

Bell [CHR\$(7)]

This code does not affect anything printed on the paper, but sounds a buzzer in the printer for about a fifth of a second. It is useful to put in programs at points when the operator's attention needs to be attracted, for example to change from one type of paper to another. The buzzer can also indicate that an error has occurred internally, such as the paper running out, but this feature can be disabled if you want to by setting a switch inside the printer case.

Backspace – BS [CHR\$(8)]

When this code is encountered in a string of data, anything in the printer buffer is printed out and the print head moved back by one space. This is useful for creating characters that are normally unavailable by overprinting two or more different characters. When in enlarged mode, the print head is moved back by two positions to compensate for the double-width characters.

With normal sized characters.

```
10 OPEN3,4
20 PRINT#3,"XXXX";CHR$(8);CHR$(8);
30 PRINT#3,"===="
40 CLOSE3
```

With enlarged characters.

```

10 OPEN3,4
20 PRINT#3,CHR$(14); "<<<<";CHR$(8);CHR$(
8);
30 PRINT#3,CHR$(20);:REM CANCEL ENLARGED
MODE
40 PRINT#3,"-----"
50 CLOSE3

```

Horizontal Tab – HT [CHR\$(9)]

When this control code is sent to the printer, the print head is positioned at the next tab position as set by the '**ESC e**' sequence. When the printer is first switched on the tab positions are automatically set to every eight characters. Multiple **HT** codes set the print head to a position corresponding to the appropriate tab mark, so if you were to send three **HT** codes on power up the printer would print the next string starting at column 24, as in the example.

```

10 OPEN3,4
20 PRINT#3,"0123456789012345678901234567
890"
30 PRINT#3,CHR$(9);CHR$(9);CHR$(9); "THIS
STARTS AT COLUMN NUMBER 24!"
40 CLOSE3

```

Vertical Tab – VT [CHR\$(11)]

This code is used to set the vertical positioning of characters, so that the next item printed appears on a given line number. Vertical tabs are set using the '**ESC e**' sequence, but if they are not set this control character behaves in the same way as linefeed, advancing the paper by one line. This code cancels the enlarged mode as set by Shift Out.

```

10 OPEN3,4
20 PRINT#3,CHR$(27);"E";CHR$(1);CHR$(2);
:REM SEE 'ESC E'.
30 PRINT#3,"VERTICAL TAB"
40 PRINT#3,CHR$(11);"THIS IS TABBED BY ONE"
50 PRINT#3,CHR$(11);CHR$(11);"AND THIS LINE IS 2 TABS DOWN!"
60 CLOSE3

```

Delete – DEL [CHR\$(127)]

This code deletes the last character in the printer buffer. If several characters are to be deleted this can be done simply by sending several **DEL** codes to the printer. If there are no printable characters in the print buffer, for example if a **CR** has just been executed, the code will be ignored – after all, it cannot delete something that isn't there! Delete is similar to the Backspace code, except that rather than a character being printed and the head moved back, it simply doesn't print.

```

10 OPEN3,4
20 PRINT#3,"DELETE";
30 PRINT#3,CHR$(127);:REM THE DELETE CODE.
40 PRINT#3,"ING CHARACTERS..."
50 CLOSE3

```

```

10 OPEN3,4
20 PRINT#3,"SINGLE";
30 FOR N=1 TO 6:PRINT#3,CHR$(127);:NEXT N
40 PRINT#3,"MULTIPLE DELETES CAN BE USED!"
50 CLOSE3

```

Multiple Byte Control Codes – Using ESC

When the Epson RX-80 printer receives a **CHR\$(27)** in a print statement it takes it as meaning “the next character is a command, not something to print”. This is called an escape code (**ESC**). In this section we will go through the escape code controls, which can be used to enable some of the more specialised features of this printer.

ESC J – Feed the paper by n/216 inches.

[CHR\$(27);“J”;CHR\$(n)]

This sequence causes any data in the buffer to be printed out, then the paper to be fed n/216 inches. Note that it does not set the line spacing, but only effects the one line. To set the line spacing for more than one line, see the ‘**ESC 3**’ command.

This example demonstrates the use of **ESC J** to perform a 108/216 (0.5) inch line feed:–

```
10 OPEN3,4
20 PRINT#3,"THESE TWO LINES HAVE"
30 PRINT#3,CHR$(27);"J";CHR$(108);
40 PRINT#3,"HALF AN INCH BETWEEN THEM!"
50 CLOSE3
```

With very small values of ‘n’, (less than 3), Epson do not guarantee the accuracy of the paper feeding. With n=0 no paper feeding is executed, as you would expect.

ESC M – Set printer to Elite sized characters.

[CHR\$(27);“M”]

Data printed after this code is received by the printer will be in elite-sized characters, with 12 characters per inch. With the elite character set, emphasized and condensed modes of printing are unavailable, but the appropriate control codes can be used to set internal flags in the printer so that they will come into operation when the printer returns to pica characters. In elite mode, one full line can be up to 96 characters as opposed to the 80 character line in pica mode, where there are 10 characters per inch.

```

10 OPEN3,4
20 PRINT#3,"THIS IS NORMAL PICA SIZED PR
INTING"
30 PRINT#3,CHR$(27);"M";"AND THIS IS ELI
TE SIZED PRINTING"
40 CLOSE3

```

ESC P – Set the characters to Pica (10 Characters / inch) [CHR\$(27);“P”]

When elite characters have been used and are not required any more, the printer can be reset to pica using this escape command. Any of the features indicated by the internal flags, such as condensed and enlarged modes will take effect immediately the print size is reset.

```

10 OPEN3,4
20 PRINT#3,CHR$(27);"P";"THIS IS PICA PR
INTING"
30 PRINT#3,CHR$(27);"M";"NOW IT'S ELITE
SIZED"
40 PRINT#3,CHR$(27);"P";"AND PICA SIZE A
GAIN!"
50 CLOSE3

```

ESC W – Set or cancel Enlarged print mode. [CHR\$(27);“W”;CHR\$(n)]

This command has two effects, depending on the value of the parameter 'n'. If 'n' is specified as 1 or 49, everything printed subsequently will be in enlarged mode, with each character twice its normal width. This is similar to the Shift Out code in effect, except that unlike the latter it is not cancelled automatically when a linefeed is performed,. When 'n' is given a value of 0 or 48, **ESC W** turns the enlarged mode off, regardless of whether it was set by **SO** or **ESC W**.

```

10 OPEN3,4
20 PRINT#3,CHR$(27);"W";CHR$(1);
30 PRINT#3,"THIS IS ENLARGED MODE AS SET
BY ESC W 1 "

```

```

40 PRINT#3,"IT ISN'T RESET BY LINEFEED A
S SO IS,"
50 PRINT#3,CHR$(27);"W";CHR$(0);
60 PRINT#3,"BUT IT IS RESET BY ESC W 0 0
R 48."
70 CLOSE3

```

ESC SO and ESC SI – Set enlarged and condensed print modes. [CHR\$(27);CHR\$(14) and CHR\$(27);CHR\$(15)]

These two commands are identical to the single byte control codes, **SO** and **SI**, which set Enlarged mode and Condensed mode respectively. There is no real advantage in using the escape code before the actual controls, except to maintain consistency in your programming.

```

10 OPEN3,4
20 PRINT#3,CHR$(27);CHR$(14);"THIS IS EN
LARGED";
30 PRINT#3,CHR$(20);" THIS IS PICA SIZE
D"
40 PRINT#3,CHR$(27);CHR$(15);"THIS IS CO
NDENSED AND ";
50 PRINT#3,CHR$(18);"THIS IS BACK TO NOR
MAL"
60 CLOSE3

```

ESC – (minus) – Set or cancel underline mode [CHR\$(27);“-”;CHR\$(n)]

This sequence is used to set or cancel underline mode – when the value of ‘n’ is 1 or 49, subsequent data printed will be underlined, until the command is repeated with ‘n’ having a value of 0 or 48. (48 and 49 are the ASCII codes for “0” and “1” respectively, so the commands could be sent as **ESC “-1”** and **ESC “-0”** – the same applies to other escape commands with these parameters.)

```

10 OPEN3,4
20 PRINT#3,CHR$(27);"-1";"THIS STATEMENT
IS UNDERLINED"
30 PRINT#3,CHR$(27);"-0";"BUT THIS ONE I
S NOT!"

```

ESC E – Set Emphasised Mode Printing [CHR\$(27);“E”]

This command brings emphasised printing into operation, giving a better quality output than normal, when in pica mode only. If the command is sent while the printer is set to print in elite characters the internal flag in the printer is set, but it will only have an effect when pica is restored by means of the **ESC P** command. While the printer is in emphasised mode, the printer will only operate at a rate of 50 CPS, rather than the usual 100 CPS, in order to avoid overheating the print-head and causing damage.

```
10 OPEN3,4
20 PRINT#3,"ONLY ONE WORD IN THIS LINE I
S ";
30 PRINT#3,CHR$(27);"E";"EMPHASISED!"
40 CLOSE3
```

ESC F – Cancel Emphasised Mode. [CHR\$(27);“F”]

Once emphasised mode has been used, we will normally want to be able to return the printer to normal printing. This is done by means of the **ESC F** command, which cancels the emphasised mode set by **ESC E**. As with the other special printing modes, when using elite characters only the internal flags in the printer are set, so that when it is returned to normal printing with the **ESC P** command it is in the required mode.

```
10 OPEN3,4
20 PRINT#3,"ONLY ";CHR$(27);"E";"ONE ";
30 PRINT#3,CHR$(27);"F";"WORD IS EMPHASI
SED IN THIS LINE!"
40 CLOSE3
```

ESC G – Set Double Strike Mode. [CHR\$(27);“G”]

Sending this command sequence to the RX-80 will cause it to print anything that is stored in its buffer – further data is then printed in double strike mode. This means that each line of information sent to the printer will be printed twice, the paper being fed by about 1/216 inches between each pass. Because of this slight offset the vertical gaps between the dots are filled in, giving the resulting print a very high quality look.

```

10 OPEN3,4
20 PRINT#3,"THIS IS NORMAL PRINTING."
30 PRINT#3,CHR$(27);"G";"AND THIS IS DOU
BLE STRIKE PRINTING"
40 CLOSE3

```

ESC H – Cancel Double strike mode. [CHR\$(27);“H”]

Once double strike mode has been set, it is often desirable to set the printer back to normal, single strike mode. This is done by sending the **ESC H** command, as in this example:–

```

10 OPEN3,4
20 PRINT#3,CHR$(27);"G";"THIS IS DOUBLE
STRIKE PRINTING"
30 PRINT#3,CHR$(27);"H";"BUT NOW WE'RE B
ACK TO NORMAL!!!"
40 CLOSE3

```

ESC S – Set Superscript or Subscript Mode. [CHR\$(27);“S”;CHR\$(n)]

One of the more unusual features of the Epson RX-80 is the ability to print characters as superscripts or subscripts. When the value of 'n' is 0 or 48, superscript mode is selected and data will be printed half sized on the top half of the line. Setting the value of 'n' to 1 or 49 (the ASCII-code for "1") will give rise to half sized characters being printed on the bottom half of the line (subscript). The printer forms the characters by double strike printing, the paper being advanced by 1/216 inches before the second pass which completes the character. In both double strike and super/subscript modes, this extra degree of linefeed is automatically compensated for by the printer, but play in the paper feeding mechanism may occasionally cause deformed characters. Both of these conditions can be cancelled by the **ESC T** command.

```

10 OPEN3,4
20 PRINT#3,"THE GENERAL FORM OF A CUBIC
EQUATION IS:-"
30 PRINT#3,"Y=AX";CHR$(27);"S0";"3";CHR$(
27);"T";"+BX";
40 PRINT#3,CHR$(27);"S0";"2";CHR$(27);"T
";"+CX+D"
50 PRINT#3:PRINT#3:REM 2 BLANK LINES...
60 PRINT#3,"THE CHEMICAL EQUATION FOR SU
LPHURIC ACID IS:- ";
70 PRINT#3,"H";CHR$(27);"S1";"2";CHR$(27
);"T";"S0";
80 PRINT#3,CHR$(27);"S1";"4";CHR$(27)"T"
:PRINT#3
90 CLOSE3

```

ESC T – Cancel Superscript and Subscript Mode [CHR\$(27);“T”]

Well, if you read about superscripts and subscripts, you will already know that this code, when sent to the printer will cancel any superscript or subscript mode set up by **ESC S**. For an example of how it is used, see the example program for **ESC S**.

ESC 4 – Select the Alternate Character set [CHR\$(27);“4”]

This command allows you to print data out in the alternate character set supplied with the RX-80. It is, in fact an italic character set and can be used to draw attention to a specific word or phrase in the same way as italics are used. For an example of this command, see the sample program under **ESC 5**.

ESC 5 – Cancel the Alternate Character set [CHR\$(27);“5”]

As previously mentioned, sending this command to the RX-80 will reset it to the normal character set, cancelling the effect of the **ESC 4** sequence.

```

10 OPEN 3,4
20 PRINT#3,"THIS IS IN NORMAL PRINT"
30 PRINT#3,CHR$(27);"4";"THIS IS THE ALT
ERNATE CHARACTER SET,"
40 PRINT#3,CHR$(27);"5";"AND NOW WE'RE B
ACK TO NORMAL AGAIN!"
50 CLOSE3

```

ESC R – Select one of the International Character Sets [CHR\$(27);“R”;CHR\$(n)] – N must be between 0 and 10 inclusive

Sending this command to the printer will cause it to print further data in one of the 11 international character sets provided with it. The most usual use for this in England is to select the English character set which includes the pound sign – this corresponds to a value of ‘n’ of 3. The following program prints out each of the character sets available in turn; for further details one the form of each of the founts you should refer to the operating manual supplied with the RX-80, Appendices E, F and G.

```
10 OPEN3,4
20 FOR I=0 TO 10
30 PRINT#3,CHR$(27);"R";CHR$(I);
40 FOR N=33 TO 126:PRINT#3,CHR$(N);
50 NEXT N:PRINT#3
60 NEXT I
70 CLOSE3
```

ESC m – Select Special Characters [CHR\$(27);“m”;CHR\$(n)]

As well as containing the eleven international character sets, the Epson RX-80 can print out the graphics characters found on the Epson HX-20 portable computers. A value of 4 for ‘n’ will give access to these characters which are found between **CHR\$(128)** and **CHR\$(159)**, while giving ‘n’ a value of 0 will cause characters in that range to be treated as control characters.

```
10 OPEN 3,4
20 PRINT#3,CHR$(27);"M";CHR$(4);:REM NOT
E THAT IT'S A SMALL 'M'!!!
30 FOR N=128 TO 159
40 PRINT#3,CHR$(N);" ";
50 NEXT N
60 PRINT#3,CHR$(27);"M";CHR$(0);
70 CLOSE3
```

ESC 0, 1 & 2 – Set Line Spacing. [CHR\$(27);“digit”]

This command sequence can be used to set the spacing between successive lines. If the digit specified is “0”, lines will be spaced at 1/8 inch intervals. A digit of “1” gives 7/72 inch line spacing where no gap is left between lines, and **ESC 2** sets spacing to 1/6 inch, as shown by the example program below:–

```

10 OPEN 3,4
20 PRINT#3,CHR$(27);"0";"THESE LINES ARE
"
30 PRINT#3,"1/8 INCHES APART."
40 PRINT#3,CHR$(27);"1";"LINE SPACING IS
NOW"
50 PRINT#3,"SET TO 7/72, AND";CHR$(27);"
2"
60 PRINT#3,"NOW WE'RE ON 1/6 INCH LINE S
PACING!"
70 CLOSE 3

```

ESC 3 – Set the line spacing in terms of n/216 inches. [CHR\$(27);“3”;CHR\$(n)]

This code, when received by the printer, causes all subsequent lines to be n/216 inches apart. The value of ‘n’ may be anything between 0 and 255, but as with **ESC J** (which does the same, but for only one line), values of 1 and 2 for n are not guaranteed by Epson. The following example shows the use of **ESC 3** to set the line spacing to 1/2 inch, with a value of 108 for ‘n’.

```

10 OPEN 3,4
20 PRINT#3,CHR$(27);"3";CHR$(108);
30 PRINT#3,"LINES WILL NOW"
40 PRINT#3,"0.5 INCHES APART"
50 PRINT#3,"UNTIL THEY ARE RESET!"
60 CLOSE 3

```

**ESC A – Set the line spacing in terms of n/72 inches.
[CHR\$(27);“A”;CHR\$(n)]**

Similar in operation to the previous command, the **ESC A** sequence will set the line spacing to n/72 inches. Two adjacent pins in the print head are 1/72 inches apart, so this command can be thought of as setting the line spacing in terms of numbers of dots. For example, 1/2 inch line spacing can be done with the program above by substituting line 20 with this:–

```
20 PRINT#3,CHR$(27);"A";CHR$(36);
```

ESC e – Set TABs. [CHR\$(27);“e”;CHR\$(n);CHR\$(m)]

With the value of ‘n’ as 0 or 48, this command will set horizontal tabs every ‘m’ characters – these can then be used in formatting by the **HT** instruction [CHR\$(9)]. The minimum value of ‘m’ is 0, in which case the **HT** code is ignored, but the maximum value depends on the character mode at the time: 21 for pica, 25 for elite, 36, 36 for condensed and 21 for emphasised. Once set, the horizontal tab is stored within the printer as an absolute length, so changing the character sizes will not alter the positions that the print head moves to one the **HT** instruction.

If the value given to the first parameter ‘n’ is either 1 or 49, the printer takes the command as a vertical tab set and the parameter ‘m’ is taken as the length of paper to be fed for each **VT** instruction, at the current line spacing. As with the horizontal tab settings, this length is stored as an absolute distance, so altering the line spacing will not affect the distance tabbed. The minimum value for ‘m’ is 0, when **VT** will behave in the same way as a carriage return and the maximum value corresponds to the form length. Attempts to set vertical tabs with greater values than this will be ignored by the printer.

```

10 OPEN3,4
20 PRINT#3,"0123456789012345678901234567
890"
30 PRINT#3,CHR$(27);"E";CHR$(0);CHR$(10)
;
40 PRINT#3,CHR$(9);"1ST TAB"
50 PRINT#3,CHR$(9);CHR$(9);"2ND TAB"
60 PRINT#3,CHR$(15);CHR$(9);CHR$(9);"2ND
TAB, BUT CONDENSED!"
70 CLOSE3

```

```

10 OPEN3,4
20 PRINT#3,CHR$(27);"E";CHR$(1);CHR$(3);
30 PRINT#3,CHR$(11);"ONE VERTICAL TAB"
40 PRINT#3,CHR$(11);CHR$(11);"TWO VERTIC
AL TABS"
50 PRINT#3,CHR$(11);CHR$(11);CHR$(11);"T
HREE VERTICAL TABS!"
60 CLOSE3

```

ESC f – Set skip position. [CHR\$(27);CHR\$(n);CHR\$(m)]

When the **ESC f** command is encountered by the RX-80, it is taken as a **SKIP** setting instruction. This enables the printer to avoid printing over perforations, so that the output is legible from page to page. Values of 0 of 48 for 'n' cause the print head to skip 'm' characters, where 'm' must be between 0 and 127. If parameter 'n' is 1 or 49, it is taken as a vertical skip, so 'm' lines are skipped, as in this example:-

```

10 OPEN3,4
20 PRINT#3,"HORIZONTAL";CHR$(27);CHR$(0)
;CHR$(10);"SKIP OF 10 SPACES"
30 PRINT#3,"VERTICAL";CHR$(27);CHR$(1);C
HR$(10);"SKIP OF 10 LINES"
40 CLOSE3

```

ESC C – Set the form length. [CHR\$(27);“C”;CHR\$(n)]

This command has two formats and allows the user to specify the length of the paper being used, either in terms of the number of printable lines per page or the page length in inches.

The first example shows the page length being set to 50 (its legal values range between 1 and 127), while the second example shows **ESC C** being used to set the form length in terms of inches. In this latter case, the value of ‘n’ is set to 0 and the printer takes the next character as the length in inches. Note that the form length is stored within the printer as an absolute length. Altering the line spacing will not change the form length, even if it is specified in terms of the number of lines.

Setting Form length to 50 lines.

```
10 OPEN#3,4
20 PRINT#3,CHR$(27);"C";CHR$(50);
30 CLOSE#3
```

Setting Form length to 11”.

```
10 OPEN#3,4
20 PRINT#3,CHR$(27);"C";CHR$(0);CHR$(11)
;
30 CLOSE#3
```

ESC N – Set the number of lines to be skipped at the bottom of each page. [CHR\$(27);“N”;CHR\$(n)]

This sequence is used to set the number of lines to be skipped at the end of each page. This function stops printing over the perforation across the end of each page so that lines are not split across the page boundaries. For example, if the gap required at the bottom of each page is 10, the value of ‘n’ should be set to 10. When the printer is first turned on, the skipover is set to 1”, but resetting the form length using **ESC C**, it is cancelled, so **ESC N** must be used. Attempts to set the skipover length to a value greater than the form length will be ignored by the printer.

ESC O – Cancel the perforation skip feature. [CHR\$(27);“O”]

This sequence is used to alter the position of the end of each line. The value given to the parameter ‘n’ becomes the actual number of characters that can be printed on a line, starting at the left margin.

ESC Q – Set the right margin. [CHR\$(27);“Q”;CHR\$(n)]

This sequence is used to alter the position of the end of each line. The value given to the parameter ‘n’ becomes the actual number of characters that can be printed on a line, starting at the left margin.

```
10 OPEN3,4
20 PRINT#3,CHR$(27);"Q";CHR$(10);
30 PRINT#3,"0123456789012345678901234567
890"
40 CLOSE3
```

ESC 1 – Set the left hand margin. [CHR\$(27);“1”;CHR\$(n)]

This code is used to determine the position of the first character of each line, in terms of the number of character spaces from the edge of the printer. This feature can be used to force the printer to indent text, as in the example:—

```
10 OPEN3,4
20 PRINT#3,"THIS IS THE NORMAL LINE POSI
TION-"
30 PRINT#3,CHR$(27);"L";CHR$(10);
40 PRINT#3,"BUT NOW ALL TEXT"
50 PRINT#3,"WILL START AT COLUMN"
60 PRINT#3,"TEN!"
70 CLOSE3
```

ESC 8 and ESC 9 – Disable/Enable the paper end detector.
[CHR\$(27);“8” and CHR\$(27);“9”]

The RX-80 has a safety feature built in which stops printing if it detects that there is no paper mounted. In certain circumstances, such as when using the sheet in a batch of pre-printed forms, the printer will detect that the paper is running out before it has even got to the bottom of the form. In this case, the printer is put “**OFF LINE**” and printing halted. This feature can be disabled by sending the printer **ESC 8** – printing will then continue even if there is no paper in the carriage (this can damage the print head – you have been warned!)

To re-enable the paper end detector, the **ESC 9** code should be sent – when the paper runs out, the printer will stop, hold the computer up (by going ‘**OFF LINE**’) and give you time to change the paper. You can then continue by pressing the ‘**ON LINE**’ button on the printer.

ESC < – Set unidirectional printing for one line.
[CHR\$(27);“<”]

On this command the print head moves to the leftmost position and the next line of data is printed from left to right. Unidirectional printing should be used in applications where horizontal accuracy is important, because the normal bi-directional printing can sometimes ‘smudge’ characters if there should be any play in the printing mechanism.

ESC @ – Reset the printer. [CHR\$(27);“@”]

When the RX-80 receives this code, a power-up reset is simulated. This sets the top-of-form position to the current position and resets the printer to the condition it was in when it was first turned on. Note that the print buffer will also be cleared, losing any data that was in it waiting to be printed. This command should really be used whenever you print out a fresh document to ensure that the printer has no unexpected features set.

ESC U – Set Unidirectional printing mode.
[CHR\$(27);“U”;CHR\$(n)]

This command is similar to the **ESC <** code, used to set unidirectional printing when ‘n’ has a value of 1 or 49, but where

ESC < only sets it for one line, all data following the **ESC U** command will be printed in unidirectional mode. If the parameter 'n' is given a value of 0 or 48, the printer will be set to print in bidirectional mode.

ESC s – Set half-speed printing. [CHR\$(27);“s”;CHR\$(n)]

When 'n' has a value of 1 or 49 and this code is sent to the RX-80, data will be printed out at 50 characters per second, rather than 100 characters per second. This greatly reduces the amount of noise and also lessens the risk of overheating the print head, thereby avoiding damage. Sending this command with a value of 0 or 48 to the printer will return it to the normal printing speed – 100 characters per second.

4 BIT IMAGE MODES ON THE EPSON RX-80

These modes are entered by sending **ESC K**, **ESC L**, **ESC Y**, **ESC Z** or **ESC X** codes to the printer and can be used to print in high resolution. All the commands have the same format, which is:–

[CHR\$(27);“Letter”;CHR\$(n1);CHR\$(n2);CHR\$(d1);CHR\$(d2); . . .

'n1' and 'n2' are the high and low bytes of the number of pieces of data you want to send – the data is then sent to the printer as a string of characters.

The dots to be printed are determined by the value of the character data, each byte defining a vertical bar eight pixels deep with the most significant bit of the character corresponding to the top of the bar and the least significant bit defining the bottom.

The actual characters you send to the printer are derived in a similar way to the data you may use to define characters for display on the screen on either a VIC 20 or Commodore 64, except that each byte represents a column rather than a row in the finished character. For example, if we wanted to print the 'space invader' character in diagram 1, we would divide it into columns and add up the values for each dot to be printed, for each of the columns, as shown:–

DIAGRAM 1.

DATA TO PRINT
A SPACE INVADER
IN BIT MODE
ON THE EPSON
RX-80

Value of each pixel	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
128				●	●			
64		●	●	●	●	●	●	
32	●		●	●	●	●		●
16	●	●		●	●		●	●
8			●	●	●	●		
4			●			●		
2		●					●	
1		●					●	
Data to send →	48	83	108	248	248	108	83	48

As we have eight items of data to send, we tell the printer to expect eight characters, by setting the values $n1$ and $n2$. Two bytes are required to specify the number of data items because the printer can accept up to 1920 data bytes for each line with some of the modes. The number of data items is therefore specified in two bytes, derived like this:—

$$n1 = \text{items} - (256 * \text{INT}(\text{items}/256))$$

and

$$n2 = \text{INT}(\text{items}/256)$$

In the space invader example we have 8 data items, so $n1=8$ and $n2=0$. This example program will print our character in normal density bit mode (ESC K). For details of the **ESC** codes used to specify other densities and speeds for bit-image printing, you should refer to the Epson Operation Manual supplied with the RX-80.

One potential use of this feature is to make the RX-80 print the Commodore graphics, thus overcoming one of the major disadvantages of using a non-Commodore printer – in fact, you could make it print out user-defined characters just as easily, a feature which is difficult to implement even when using the Commodore printers!

```

10 OPEN3,4
20 PRINT#3,CHR$(27);"K";CHR$(8);CHR$(0);
30 FOR N=1 TO 8:READ BYTE
40 PRINT#3,CHR$(BYTE);
50 NEXT N:PRINT#3
60 CLOSE3:END
70 DATA 48, 83, 108, 248, 248, 108, 83,
48
```

APPENDIX I

Error Messages

BASIC I/O ERRORS

BAD DATA ERROR

The wrong type of data has been read from disk. This occurs with the **INPUT#** or **GET#** commands when you read string data instead of the required numeric data. You can get round this in two ways: either by keeping a note of the file structure that you are using so that you read in the right data at the right time, or by reading your data into string variables and then converting them into numeric ones by the use of the **BASIC** statement **VAL**.

?DEVICE NOT PRESENT ERROR

The computer is unable to communicate with the device that you have specified, (in this case the disk drive). Check the following:

1. Is the disk drive connected up to the computer?
2. Is the disk drive connected up to the mains supply?
3. Is the disk drive switched on?
4. Have you used the correct device number (usually 8 unless you have specifically changed it)?

?FILE NOT FOUND ERROR

You are trying to access a file but the **DOS** cannot find the name you have given in the directory. Possible reasons are:

1. You are using the wrong disk.
2. You have spelt the file name incorrectly.
3. The file is not still on disk, you erased it earlier.

?FILE NOT OPEN ERROR

You are trying to read from or write to a file that has not been **OPENed**.

1. Have you forgotten to **OPEN** it earlier?
2. Do you have an error condition which has effectively **CLOSEd** the file?
3. Does the program you are using **CLOSE** the file?

?FILE OPEN ERROR

You are trying to **OPEN** a file for use which has already been **OPENed** earlier and not **CLOSEd**.

or

you are trying to **OPEN** a file by using a logical file number which you have already used previously, so you need to change the number.

?ILLEGAL DIRECT ERROR

You are trying to enter data directly from the disk drive using either **INPUT#** or **GET#** as direct **BASIC** commands. This is not acceptable; **INPUT#** and **GET#** can only be used as statements to enter data indirectly from within a program.

? NOT INPUT FILE

You are trying to Read data from a file that has been **OPENed** as a Write file. You need to **CLOSE** the file and re**OPEN** it, correcting the mode type from Write to Read.

?NOT OUTPUT FILE

You are trying to write data to a file that has been **OPENed** as a **READ** file. You need to **CLOSE** the file and re**OPEN** it, correcting the mode type from **READ** to **WRITE**.

? VERIFY ERROR

The program that you have just **VERIFY**ed has not **SAVED** properly.

DOS ERROR MESSAGES**01,FILES SCRATCHED,XX**

After effecting the **SCRATCH** command which you have just input, the **DOS** has deleted XX number of files from the disk. This is not an error.

02-19

These error numbers are not used on the 1541, and if they should occur can be ignored as meaningless.

DATA TRANSFER ERRORS**20,READ ERROR,TT,SS** (block header not found)

The **DOS** cannot find the header of the block which it is looking to read from. Possible causes are:

1. The disk was not properly formatted.
2. The disk has become corrupted.

21,READ ERROR (No syno. character)

The **DOS** cannot find a synchronisation mark on the tracks that it is looking for. Possible causes:

1. The disk was not properly formatted.
2. The disk drive is badly set up and needs attention.
3. The disk is corrupted.

22,READ ERROR (Data block not present)

The **DOS** cannot find the block that it has been asked to read from. You may have forgotten to **CLOSE** the last time that you used it.

23,READ ERROR (Checksum error in data block)

The checksum, which is stored in each block as a check that the data in it has been correctly read back, is indicating an inaccuracy.

24,READ ERROR (Byte decoding error)

While checking the data, in between reading it from disk and transferring it into the computer memory, the **DOS** has found it to be corrupt and meaningless.

25,WRITE ERROR

Checking the data that has just been written to disk against the original version still in memory, the **DOS** has found an irregularity.

26,WRITE PROTECT ON

The **DOS** has been asked to write data to a disk which is protected against this by having its write-protect notch covered up. This may indicate that you are using a commercial disk by mistake.

27,READ ERROR (Checksum error in header)

The checksum, which is stored in the block header as a check that data has been read back correctly, is indicating an inaccuracy.

28,WRITE ERROR (Long data block)

After writing data to one block, the **DOS** has failed to find the next block and some data will have been erased. This usually means that the disk has become corrupted and must now be reformatted.

29,DISK ID MISMATCH

There is an inconsistency between the ID of the disk in the drive and the ID that the **DOS** thinks that it is working with.

Possible causes:

1. You have inadvertently changed disks.
2. Your disk is faulty.

DOS SYNTAX ERRORS**30,SYNTAX ERROR** (General syntax)

The **DOS** has received a command in a format that it does not fully recognise. Check carefully that you have used the correct syntax, including spaces and punctuation. (See also Appendix II: disk command summary)

31,SYNTAX ERROR (invalid command)

The **DOS** has received an instruction that it does not recognise at all. Check carefully that you are using the correct keyword and the right syntax.

32,SYNTAX ERROR (long line)

You have just used a command string that is too long, that is, more than 58 characters.

33,SYNTAC ERROR (Invalid filename)

You are using a filename of unacceptable format, possibly containing one of the wild card characters * or ?

34,SYNTAX ERROR (No filename)

The **DOS** does not recognise the filename that you have just specified, possibly because you have omitted the colon preceding it.

39,SYNTAX ERROR (Invalid command)

The **DOS** has received an instruction that it does not recognise through the command channel, 15. Check carefully that you are using the correct keyword and the right syntax.

RELATIVE ACCESS FILE ERRORS**50,RECORD NOT PRESENT**

The **DOS** has been instructed to read a record which it cannot find, possibly because you are trying to read beyond the end of the file.

51,OVERFLOW IN RECORD

While using a relative file you are trying to write an amount of data that is longer than the record length you originally specified.

52,FILE TOO LARGE

You are trying to write a record to a relative file which would make the file exceed the capacity of the disk.

GENERAL FILE ERRORS**60,FILE NOT OPEN**

You are trying to access a file before it has been **OPENed**.

62,FILE NOT FOUND

The **DOS** has been asked to find a file that is not on the disk. Possible causes are:

1. You are using the wrong disk.
2. You have spelt the filename wrongly.

63,FILE EXISTS

You are trying to create a file on disk using an already existing filename.

64,FILETYPE MISMATCH

There is an inconsistency between the filetype specified in your last instruction and the actual type of the file on disk.

65,NO BLOCK,TT,SS

The block you have specified in the **BLOCK-ALLOCATE** command has already been reserved previously. You will then be sent the address of the next free block, or zeros, indicating that no more blocks are free.

66,ILLEGAL TRACK AND SECTOR

The **DOS** has been asked to access a track and sector address that does not exist. Possible causes are:

1. A programming fault.
2. Corruption of the pointers that should indicate the next block in the file.

67,ILLEGAL SYSTEM T OR S

The **DOS** has been requested to access data from blocks on the disk that do not exist.

SYSTEM ERRORS**70,NO CHANNEL**

You are trying to **OPEN** more than the limit of three sequential files at a time, so no more channels are available.

71,DIR ERROR

There is an inconsistency between the **BAM** and the directory, that is between what should be and what actually is on the disk. You could **reINITIALIZE** the disk but you may already have lost some data.

72,DISK FULL

You are trying to add a file for which there is no more room, either in the directory or physically on the disk.

73,DOS MISMATCH

You are trying to write to a disk that has been formatted on another disk drive running a different and incompatible **DOS**.

74,DRIVE NOT READY

You have tried to access the disk drive before it is fully functioning. Possible causes:

1. There is no disk in the drive.
2. The disk was not properly formatted.
3. The drive door is not closed.

APPENDIX II

Disk Drive Commands

BLOCK-ALLOCATE

"B-A"DR;T;S

Allocates or reserves a specific block on a disk for your own use.

BLOCK-EXECUTE

"B-E"CH;DR;T;S

Reads the contents of a specific block into a channel buffer and then runs them as a machine code program.

BLOCK-FREE

"B-F"DR;T;S

De-allocates the specified block and frees it for further use.

BLOCK-READ

"B-R"CH;DR;T;S

Reads the contents of a block into the buffer of a given channel, up to the position of the buffer-pointer.

BLOCK-WRITE

"B-W"CH;DR;T;S

Writes the contents of a channel buffer into a specific block on disk, up to the position of the buffer-pointer.

BUFFER-POINTER

"B-P"CH;POSITION

Sets the pointer within a particular buffer to a given position.

CLOSE

CLOSE FN

Closes a communications channel between the computer and the disk drive.

CMD

CMDFN,STRING

Redirects primary output from the screen to a given file.

COPY

"C:NEWFILE=OLDFILE"

Creates an identical copy of a file on disk.

COPY AND CONCATENATE

"C:BIGFILE=OLDFILE,NEWFILE"

Copies several files and joins them together to make a single large file.

GET #

GET #FN,VARLIST

Reads a single character at a time from a logical file and stores it as a variable.

INITIALIZE

"IDR"

Resets the disk drive as if it has just been turned on.

INPUT #

INPUT #FN,VARLIST

Reads a whole variable from a logical file.

LOAD**LOAD"FILENAME",DN,SA**

Transfers a copy of the file on disk into computer memory.

MEMORY-EXECUTE**"M-E:CHR\$(LOW BYTE)CHR\$(HIGH BYTE)**

Runs a machine code routine starting at a specified address.

MEMORY-READ**"M-R"CHR\$(LOW BYTE)CHR\$(HIGH BYTE)CHR\$(NO. OF BYTES)**

Reads into the computer the contents of the specified area of disk drive memory.

MEMORY-WRITE**"M-W"CHR\$(LOW BYTE)CHR\$(HIGH BYTE)CHR\$(NO. OF BYTES) VARLIST**

Writes bytes into the specified area of disk drive memory.

NEW**"NDR:DISKNAME,ID"**

Formats a new disk for use. Re-formats an old disk so that it can be used as if it were a new one, if the 'ID' is not included.

OPEN**OPEN FN,DN,SA,"FILENAME,FILETYPE,MODE"**

Opens a communications channel between the computer and the disk drive.

PRINT#**PRINT#FN,DATALIST**Writes **DATA** items to a logical file.**RENAME****"R:NEWNAME=OLDNAME"**

Gives a new name to an existing file.

SAVE**SAVE"FILENAME",DN,SA**

Transfers a copy of the file in computer memory onto a disk for storage.

SAVE AND REPLACE**SAVE"@:OLDFILE",8**

Saves a file onto disk by overwriting an existing file of the same name.

SCRATCH**"S:FILENAME"**

Erases a file permanently from disk.

USER1 (or USERA)**"U1"CH;DR;T;B**

Reads the entire contents of a given block into the buffer of a specified channel.

USER2 (or USERB)**"U2"CH;DR;T;B**

Writes the entire buffer contents of a specified channel to a given block on disk.

VALIDATE**"V"**

Reorganises file storage on a disk in order to maximise the space available.

VERIFY**VERIFY"FILENAME",DN**Checks whether a file has been **SAVED** properly.**WRITE AND REPLACE****OPENFN,DN,CH,"@:FILENAME"**

Writes a file onto disk by overwriting an existing file.

APPENDIX III

Abbreviations

B		B lock
BAM		B lock A vailability M ap
CH		C Hannel
CPS		C haracters P er S econd
DN		D evice N umber
ESC		E SCape
DOS		D isk O perating S ystem
DR		D Rive number
FN		logical F ile N umber
ID		I Dentification number
NLQ		N ear L etter Q uality
S		S ector
SA		S econdary A ddress
T		T rack
VARLIST		V ARiable L IST

INDEX

INDEX

Abbreviating commands	46
Abbreviations	32
Adding files	82
Advantages	75
sequential files	74
relative files	74
direct access files	74
ASCII/graphics set	190
Alternate character set	191
Backspace	183, 185
Back-up copies	17
BAM	14
Bell	183
Bit image modes	186
BLOCK-ALLOCATE	57, 111
Block availability map	14
BLOCK-EXECUTE	61
BLOCK-FREE	59
BLOCK READ	59, 118
BLOCK-WRITE	61, 119
Buffer	42, 56, 149, 185
BUFFER-POINTER	83, 66
Buzzer	180, 183
Carriage return	40, 84, 92, 147, 174, 181
Character editor	135
Character sets	139, 170, 173
Choosing a printer	129
CLOSE	42, 71, 141, 142
CMD	41, 141
Column printing	
Combining files	52
Comma	21, 91, 92, 141, 154
Command string	46
Commercial software	20
Concatenation	52
Condensed mode	176, 183, 188
Connecting up	7
Control code	180
COPY	51
COPY and CONCATENATE	52
Daisywheel printers	133, 139
Database program	80
Data storage	5, 89
DELETE	185
Deleting files	49
Demonstration database program	78
sequential files	121
relative files	98
direct access files	109
Density	16
Device number	21
Direct access commands	72, 110
Direct access files	109
Directory	14
Disadvantages	
sequential files	76
relative files	96
direct access files	109
disk care	10, 18, 19
Disk maintenance commands	46
Disk manipulation commands	25, 206
Disk Operating System	9, 44
Disk structure	12
DOS	9
DOS support program	44
Dot matrix printers	132
Double strike mode	189
Drive-active light	8, 19

Elite	176, 186
Emphasised mode	189
Enhanced characters	
Enlarged characters	182, 188
Enlarged mode	182, 183, 187, 188
Epson HX20	192
Epson RX-80	176–201
Erasing files	49
Error channel	120
Error light	8
Error messages	24, 123, 161, 162, 202
ESC	
Escape codes	186–199
Filename	83
File types	74
Format —	159
Format	157
Format 9	151
Format \$	155
Format A	151
Format S	156
Format Z	153
Formatting characters	150–159
Formatting commands	146
Formatting a disk	23
Form feed	180, 181
Form length	196
GET#	40, 56, 92
Graphics Characters	136, 180
Half speed printing	198
Handshaking	178
Identifier	24, 48
INITIALIZE	53, 114
INPUT#	39, 90, 92, 118
Interfaces	136, 139, 177
International character set	192
Line feed	181
Lines per page	196
Line spacing	186, 193, 194
Listen	144, 179
LOAD	5, 21, 32
Logical files	177
Logical file numbers	36
Machine code	34
Margin	197
MEMORY—EXECUTE	71
MEMORY—READ	68, 71
MEMORY—WRITE	69
Multiple byte codes	186–199
Multiple disk drives	29
NEW	23, 47, 48
OPEN	23, 35, 56, 108, 140
Overprinting	169
Paging	135
Paper end detector	183, 198
Pattern matching	33, 49
Perforations	181, 197
Pointer positioning	61
Pica	132, 176, 187
Pointers	61
Portable printers	135
Powering up/down	8
PRINT#	39, 86, 140, 144, 178, 179
Print modifiers	89

Reading blocks	60
Reading the error channel	45, 55, 120
Relative access files	37, 94
RENAME	50, 85
Reserving blocks	37
Reset printer	198
Reverse characters	174
SAVE	5, 26, 108, 165
SAVE AND REPLACE	29
SCRATCH	49
Secondary address	147, 148, 160, 161, 164, 178
Semi colon	40, 89, 90, 92, 98
Sequential files	37, 73
Setting up	6
SHIFT	174, 182
SHIFT OUT	181, 184, 187
Side sectors	97
Single byte codes	181–185
Skipover	160, 173, 181, 195, 197
Spacing	135, 160, 168
Special characters	175, 192
STATUS	92
Storage	3
Superscript	190
Subscript	190
Switching on/off	7
Tabs	89, 153, 184, 194
Terminators	89, 163
Trailing sign	132
Typewriter-printers	135
Underline mode	188
Unidirectional printing	198
Upper case printing	
USER 1	65
USER 2	66
USER A	65
USER B	66
User-defined characters	163–168
User files	37, 88
Underline mode	
VALIDATE	54, 110
VERIFY	30
WRITE AND REPLACE	38
Write/protect notch	11, 17
Writing to blocks	16

ALL COMPUTER OWNERS FIND THAT MOVING ON TO USING DISK DRIVES AND PRINTER SYSTEMS BRINGS THEM INTO NEW AREAS WITH SPECIAL COMMANDS AND FUNCTIONS NOT PREVIOUSLY ENCOUNTERED. THIS BOOK IS DESIGNED TO TAKE THE NEW OWNER, WHETHER AT HOME OR IN BUSINESS, FROM SETTING UP TO USING COMPLICATED DATA BASES OR HANDLING SPECIALLY FORMATTED SCREEN DUMP ROUTINES.

DISK MATERIAL INCLUDES

BASIC COMMANDS
DOS COMMANDS
SEQUENTIAL FILES: RANDOM ACCESS FILES:
RELATIVE FILES:
FULL EXAMPLES PLUS HINTS AND TIPS ARE SHOWN

PRINTER MATERIAL INCLUDES

PRINTER COMMANDS
EXAMPLE ROUTINES
FEATURING COMMODORE PRINTERS AND OTHER
COMPATIBLE MACHINES
SUITABLE FOR USE WITH ALL COMMODORE
MACHINES



PHOENIX
PUBLISHING
ASSOCIATES

£7.95

ISBN 0-9465-7623-8



9 780946 576234



**This was brought to you
from the archives of**

<http://retro-commodore.eu>