THE COMMODORE 64

DISK BOOK

A simple guide to using your disk drive

The Commodore 64 Disk Book

Other titles published by Century Communications and Personal Computer World

Best of PCW Software for the BBC Micro
Best of PCW Software for the Electron
Best of PCW Software for the Spectrum
Best of PCW Software for the Dragon 32

The Database Primer
ROSE DEAKIN

Computer Gamesmanship
DAVID LEVY

Information Technology Yearbook
edited by PHILIP HILLS

Century Computer Programming Course
PETER MORSE and IAN ADAMSON

Educational Programs for the Spectrum
IAN MURRAY

Educational Programs for the Dragon 32
IAN MURRAY

35 Educational Programs for the BBC Micro
IAN MURRAY

The Intimate Machine
NEIL FRUDE

35 Programs for the Dragon 32
TIM LANGDELL

The Spectrum Handbook
TIM LANGDELL

The Microcomputer Handbook: A Buyer's Guide
edited by DICK OLNEY

Microcomputing for Business: A User's Guide
edited by DICK OLNEY

The Commodore 64 Disk Book

Tony Hetherington and Gordon Hamlett

CENTURY COMMUNICATIONS
LONDON

Copyright © Tony Hetherington and Gordon Hamlett

All rights reserved

First published in Great Britain in 1984 by
Century Communications Ltd (a division
of Century Publishing Co Ltd), 12–13 Greek Street,
London W1 and Personal Computer World,
62 Oxford Street, London W1

ISBN 0 7126 0543 6

Filmset by Deltatype, Ellesmere Port
Printed and bound
in Great Britain in 1984 by
Hazell Watson & Viney Limited,
Member of the BPCC Group,
Aylesbury, Bucks

Contents

Introduction 1

1. Getting Started 3
2. Diskette Care 7
3. BASIC Commands 14
4. Disk Maintenance Commands 25
5. Storage of Data:Sequential Files 31
6. Storage of Data:Random Access Files 41
7. Storage of Data:Relative Files 52
8. Changing the Drive Device Number 57

Appendices

i Quick Reference Table of Disk Commands 59

ii Disk-Related Error Messages 61

iii Programs 76

Introduction

This book is designed for the user with little or no practical experience of using disk drives. The aim is for the reader to be able to stop wherever he wants so that if, for example he only wants to know how to load commercial programs, then he need read no further. Hopefully, there will be no need to go searching through the rest of the book, trying to find the particular piece of information that he wants. Also, you will find a table of the various disk commands at the back of this book, to save time while using your disk unit.

It is worth considering at this point the advantages of a disk based system over one which only uses cassettes. The main difference you will notice is the vast saving in time while you are loading programs. A fairly large program could quite easily take fifteen to twenty minutes to load (if you don't get any load errors!) using a cassette based system. Using your disk drive the same program, be it game, word processing program or a comprehensive accounting system, will take about one fiftieth of that time, – quite an improvement!

The next advantage brought about by owning a disk unit is the vast amount of memory that your computer can use. True, it cannot work on the full capacity of a disk at any one time (about 17ØK), but the speed of the system means that large amounts of information can be read from and rewritten to the disk very quickly. This effectively gives your computer a lot

more memory, and allows you to make much better use of it. Should you decide that 170K (170,000 characters) of extra memory is not enough for your particular application, there are two main options you could follow up. The first is that you can run up to five disk drives at any time from your VIC 20 or Commodore 64, but this would be rather costly. The second option, which is considerably cheaper, is to store your programs and data on more than one disk.

Finally, a disk based system is much more flexible than a cassette based one, because two way communication is possible between your computer and disk unit. This means that you can read information from the disk into the computer, change it, and then store it back on a disk very quickly. Although this is possible using cassettes, the process would be so slow as to be almost totally useless. This is why most business programs are disk based.

CHAPTER ONE

Getting Started

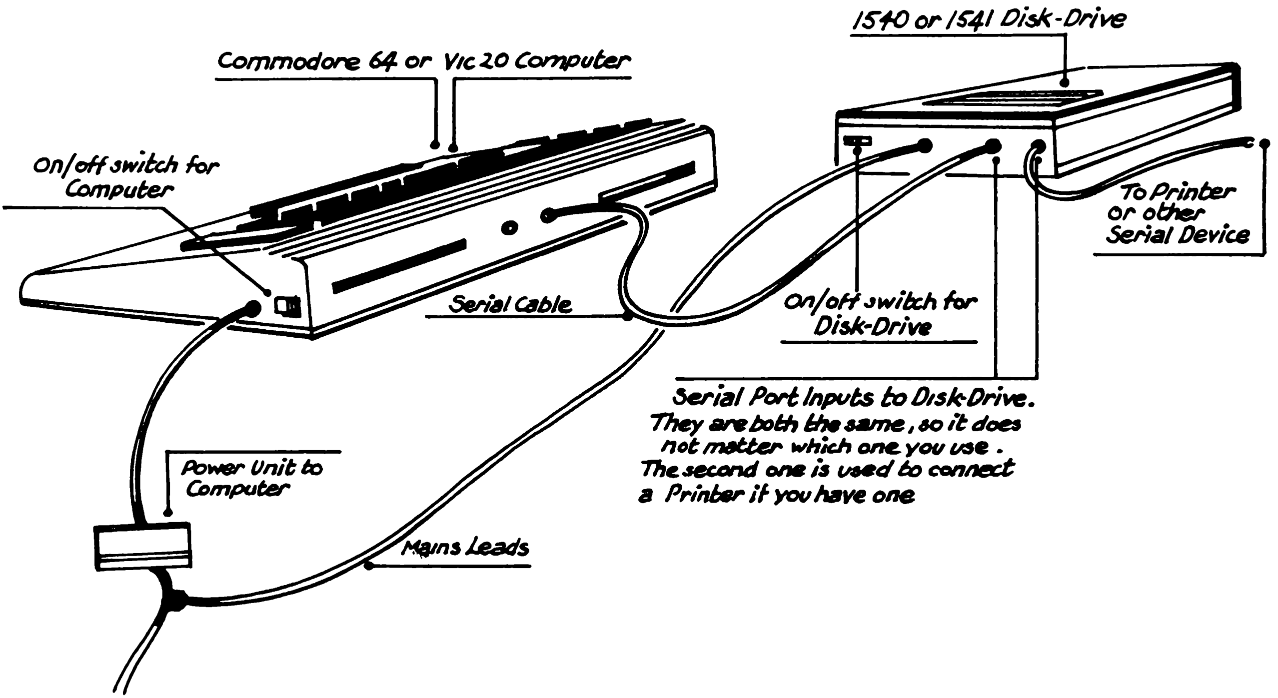
When you buy your disk drive, before you can use it, you must connect it to your computer, and to the mains. You should find two leads packed with it to perform these two functions. Fit a standard 13A plug to the grey mains lead, the other end of which plugs into the socket on the back of your disk drive by the ON/OFF switch. The black lead with the DIN plugs on either end should be plugged into the corresponding socket on the back of your computer (the serial port), and the other end should be plugged into one of the two sockets on the back of the drive.

The finished product should look something like the illustration overleaf.

When you are sure that you have set the system up correctly, you will be ready to switch it on. There is a specific order in which you must do this to ensure trouble free use of your drive. If you don't follow this sequence you are running the risk of losing any information on disks, or possibly even causing damage to your drive!

The sequence for switching everything on is as follows:

1. Remove any disk that is in the drive.
2. Ensure that the television/monitor, computer and drive are all switched off.
3. Switch on at the mains.


CONNECTING YOUR 1540/1541 DISK DRIVE TO YOUR COMPUTER.
1540 or 1541 Disk-Drive
Commodore 64 or Vic 20 Computer
On/off switch for Computer
Serial Cable
On/off switch for Disk-Drive
To Printer or other Serial Device
Serial Port Inputs to Disk-Drive. They are both the same, so it does not matter which one you use. The second one is used to connect a Printer if you have one
Power Unit to Computer
Mains Leads

4. Turn the drive on. The green light (power indicator) will come on permanently. The red light will also come on for a second or so, and the drive will make a slight whirring noise.
5. Turn on the television and computer. The normal Commodore message should appear on the screen, the red light will come on briefly, and the drive motor will start for a while.
6. When the red drive light goes out again, everything is ready to use.

When you have finished using the disk drive and want to turn it off, there is another sequence to follow to ensure that no data is lost, which is as follows:

1. Remove any disk from the drive. Failure to do this may result in loss of data from that disk.
2. Turn the computer off.
3. Turn off the television and disk drive.
4. Turn everything off at the mains.

To insert a disk into the drive, remove the disk from its paper sleeve, making sure that you only handle the disk by its protective plastic holder. If you touch the brown magnetic surface of a disk, the chances are that you will render it totally unusable, as disks are very sensitive to any foreign matter on their surfaces, even the grease on a clean finger will destroy one!

A disk is inserted into the drive with its label facing upwards and towards you, you should see a small cutout section on the left-hand edge. This is known as the write-protect slot. A further description of the construction of a diskette is given later on. Gently push the disk into the slot in the front of your disk unit until you feel it click into place, and then close the door by firmly pulling the flap down and towards you.

To remove a disk from the drive, push the flap inwards and upwards: the disk will spring out slightly, and you may then

pull it out of the drive. Be sure to put the disk back in its paper sleeve while it is not in the drive, to protect it from damage.

NB. There is an error on page 8 of the disk drive manual, if you have the edition with a silver cover which states that 'you should never remove a disk when the green light is on.' As the green light is the power indicator, you would wait for a long time to take your disk out! This sentence should read 'You should never remove a disk from the drive when the *red* light is on.' This is because the red light indicates that the disk drive is either reading from, or writing to the disk, and pulling the disk out during either of these operations will damage the diskette.

CHAPTER TWO

Diskette Care

Before you become familiar with using your disk unit, it is perhaps a good idea to understand the construction of the diskettes you will be using, and how you should care for them in order to prolong their life.

Construction of the Diskette

As you are no doubt aware, the diskette is, as the name implies, a disk of magnetic material. In fact, it is the same material as cassette tapes are made from. A disk on its own would be very fragile, although it is flexible the surface is easily damaged by any grit, dust, or even grease from a fingertip. The disk itself is therefore encased in a protective plastic holder, to minimise the chances of this type of damage. We would also recommend that any disk not actually in use is kept in a disk library case. Sandwiching the actual disk inside its holder are two sheets of a special fabric, which gently wipe the diskette clean as it spins. This serves to lengthen the life of the diskette by ensuring that the surface is kept as clean as possible.

The disk drive accesses the disk through a series of cutout sections in the protective holder and it is worth examining these now, as some insight can be gained from them as to what happens to the disk inside the drive. When you slide the

disk inside the drive, the two semicircular cutouts opposite the label ensure that the disk is in the right position, and match up with two corresponding bars in the drive. The correct vertical position is ensured by two grooves along which the diskette slides as it is inserted. A spindle is inserted through the central hole in the diskette as the flap is closed, and this is then used to spin the disk so that the drive can read and write to it, when required.

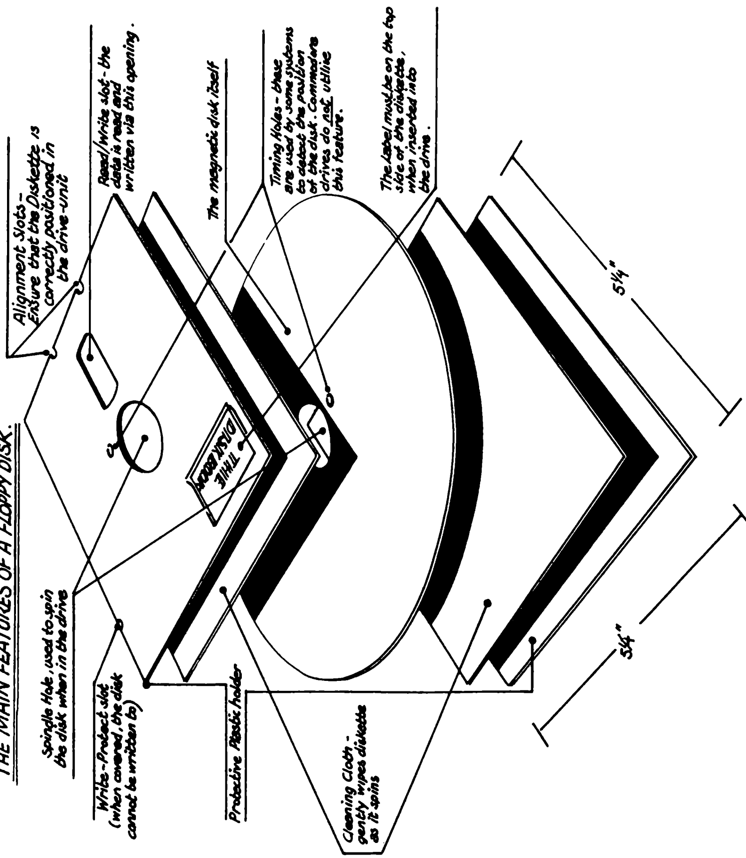
The disk unit has a read-write head, similar to that of a cassette deck, which is lowered to within a fraction of an inch above the magnetic surface. The read-write head "looks through" the large cutout section in the protective casing of the diskette opposite the label.

There are two other cutouts on a standard diskette which have not been mentioned yet. The small hole near the centre of the disk is used by some drives to check that they are spinning the diskettes at the correct speed. The Commodore system does not need this, as it writes special magnetic markers onto the diskette the first time it is used.

The final notch, in the left-hand side of the diskette cover, is known as the write-protect slot. If the drive is instructed to write any information to the diskette, it first shines a light through this slot. If the light is detected on the other side of the slot, the drive will write the data onto the disk. Sometimes, however, it is undesirable to write data onto a disk, for example a disk with a commercial program on. In this case, any write command is blocked by sticking a small label over the write-protect slot. If you try to write data to a disk protected in this way, the red drive light will flash on and off repeatedly. This is the unit's way of telling you that there is something wrong. Later on we shall discuss how to deal with such errors.

Storage of Data

The actual data on a disk is stored in a series of concentric rings, called TRACKS: YOUR Commodore drive has 35 of these,


THE MAIN FEATURES OF A FLOPPY DISK.
Alignment Slots - Ensure that the Diskette is correctly positioned in the drive-unit
Spindle Hole, used to spin the disk when in the drive
Write-Protect slot (when covered, the disk cannot be written to)
Read/Write slot - the data is read and written via this opening.
THE DISK BOOK
The magnetic disk itself
Protective Plastic holder
Timing Holes - these are used by some systems to detect the position of the disk. Commodore drives do not utilise this feature.
Cleaning Cloth - gently wipes diskette as it spins
The label must be on the top side of the diskette, when inserted into the drive.
5¼"
5¼"

numbered from 1 on the outside to 35 on the inside. Each track is also divided into SECTORS, or BLOCKS. As the tracks on the circumference of the disk are longer than those on the inside, in order to make the best use of the available disk area, the tracks on the outside of a disk contain more blocks than those on the inside. In fact, there are 21 blocks on the outermost tracks, and this number falls to 17 per track by the time you reach the centre track.

Each block contains 256 characters, but two are used to tell the drive where the next block in a file is. A file is a series of blocks containing information. This information may be either a program, or data which has been written to the disk for storage. As a file is read, the disk drive looks at the first two characters of each block in order to decide where the next block in that file is. These two characters are known as POINTERS, since they point at the next block in a series. If there is no next block in the file, these will point at track Ø, sector Ø, which does not exist.

The disk drive keeps a record of the files on a disk, using a special file called the DIRECTORY, which always starts on track 18, sector 1. This file contains information about the files on the disk including their names, the filetype (which will be explained later), where on the disk they start, and how long they are, everything that the drive needs to know in order to read (or write to) any file on the disk.

If, for example, you wanted to read a file called "CENTURY", the drive would first go to the directory and search for the details of that file. If it could not find a corresponding entry, then it would tell us by flashing the error light.

Once the correct entry has been found, the read/write head is then moved over the track containing the first block of data, and then waits for the appropriate sector to pass under it. As the required block passes the head, it is read, and the drive then carries on by searching for the next block it needs. By using pointers to control where the head is moved to, the data for any one file can be spread all over the disk, fitting into any spare gaps, thereby making the best possible use of the

memory available on the disk. In order to minimise the time required to read any given file, the directory is always on track 18: this is halfway out from the centre of the disk, so the distance that the head has to move to get to any track is kept to a minimum.

In addition to the directory, the disk unit keeps a record of which blocks on the disk have been used in track 18 sector Ø, so that it can make sure that it does not overwrite any data previously stored on the disk. This record is, in effect, a map of every block on the disk, and is called the BLOCK AVAILABILITY MAP. As this is rather a mouthful, it is often referred to as the BAM.

When it is instructed to write information to a disk, the drive consults the BAM and decides where it should store the data, firstly so that it does not overwrite anything, and secondly so that future access times are kept down. This is done by selecting blocks that are as close to the directory as possible, and as close to each other as possible. Once the data has been written out to the diskette, the drive then updates the BAM, so that it will know the next time around that it cannot use the same blocks.

Buying Disks

There are three important terms used to describe disks, they are: size, density, and whether they are single or double sided.

The size is quoted as the diameter of the disk, and is measured in inches. These sizes range from 8″ disks, down to 3″. Your 1541 drive uses standard 5¼″ disks.

The density of a disk is a manufacturer's term, which describes how much information can be packed onto a disk. You can obtain single, double and even quadruple density disks, but having never had any problems with single density disks, we would recommend that you use the cheaper, single density ones. The others would work just as well, but it would be a waste of money as you would gain no advantage by buying the more expensive ones.

The 1541 only uses one side of a disk, so again, the cheaper, single sided disks are the ones to look for. The alternative is to buy the more expensive double sided disks, and waste the other side, not very sensible!

Prolonging the Life of Your Disks

Whilst cassettes are easily stored, floppy disks are both more fragile and more sensitive to the environment. There are several rules which should be followed in order to prolong the life of your disks.

1. Never touch any part of the disk, apart from the plastic holder (and that carefully).
2. Do not keep your disks on top of the drive unit, as the drive gets warm and also creates magnetic fields, which could destroy any data on them.
3. Keep diskettes away from the following:
   (a) Cigarette smoke (they are non-smokers!).
   (b) Bright sunlight (the warmth and ultra-violet light will destroy data).
   (c) Telephones (they create magnetic fields).
   (d) Coffee (they tend not to work very well after any drink has been poured on them!).
   (e) Any source of magnetic or electrical fields, for example, Hi-Fi units, Televisions (even Monitors!), Radio Transmitters, etc.
   (f) Any source of dust, etc. *However fine*!
4. Do not bend disks. When not in use, they should be put in their cardboard sleeves, and preferably in a plastic box as well.
5. When labelling disks, try to write the label before sticking it onto the disk. If this is impractical, on no account should you use a ballpoint pen or sharp pencil. Pinning labels to disks tends to ruin them rather efficiently too, so don't do that either!
6. Always make sure that you remove disks from the drive before switching the power either on or off, you run the risk

of losing 170K of data if you don't.

7. Never try to remove a disk from the drive while the red light is on as the disk unit is trying to read or write data to the disk. Attempting to interrupt this will almost certainly cause you terrible problems!

If you follow the above rules, your diskettes should last a fair while and give you no problems. It is worthwhile, however, to point out that the electricity board does not often tell you when they are going to turn the power off! Bearing this in mind, a good habit to develop while using your disk drive would be to save any information at least twice, preferably three times if the data is important (such as business accounts, etc.). This way, even if there is a power-cut, the dog dribbles on your disk, or something else happens, you should always have a copy to fall back on!

CHAPTER THREE

The BASIC Commands

Before you can start loading and saving programs and data on your disks, each one has to be prepared for use. This is called formatting, and only has to be done once, when the disk is used for the first time. Commercial program disks will already have been prepared so do not attempt to format them, because the process will wipe out any programs and data that may have been on the disk.

Formatting

Formatting is necessary because the drive has to mark out the tracks and sectors, so that it knows where it is on the disk. It also creates the directory and BAM during this operation, and writes special characters (known as the ID) to each block on the disk, which act as an extra safety check, in that they enable the drive to tell if you have changed a diskette while it is reading or writing, because the ID will be different.

Before you can communicate with the disk drive, in order to tell it to format a diskette (or indeed perform any operation), you must open a channel through which instructions and information can flow. These channels are opened by the computer, using the OPEN command. The OPEN command has the following form:

```
OPEN lfn,dn,sa,"command-string"
```

Where lfn means logical file number (this is a reference number by which you specify the particular channel when giving further instructions to the drive). Dn is the Device Number, normally for a disk drive this is 8, although it may be altered using a program on the demo diskette (DISK ADDR CHANGE). Sa stands for Secondary Address, and is used by the drive to decide on the type of command coming next. Control commands intended to operate on the whole of a diskette are always sent through a channel with a secondary address of 15. The command string contains the actual instruction that you want to send to the disk, and specifies the drive number (normally Ø), and then either the command itself or the name of the actual file that you want to affect.

In order to format a diskette, the command string will consist of the actual instruction "NEW", the drive number, "Ø", the name you want to give to the whole disk, e.g. GAMES, or ACCOUNTS, and the identifier (ID). The ID may be any two characters: We recommend that you try to give each of your diskettes different ID's, as they are part of the disk drive's error checking mechanism.

So the command will look something like this:

```
OPEN1,8,15,"NEWØ:DISKNAME,ID"
```

If you put an unused diskette in your disk unit and type this instruction, the drive will leap into action: the red read/write light will come on, and the disk will be formatted. This process takes a few minutes, as the drive has a lot to do. For a start, it has to create the tracks and sectors on the disk, one by one, writing the identifier to each one as it goes, then it reads each block to make sure that they are correct, and finally it goes and creates the Directory and Block Availability Map. If it finds that anything it wrote to the disk has not been correctly stored when it comes to read it again, the formatting process will stop, and the red drive light will flash in order to tell you that something has gone wrong. All being well, though, the drive light will go out after a few minutes, and the diskette will be ready for use.

Once you have finished giving instructions to your disk unit, you must tell it so, by CLOSEing the communication channel that you opened. This is done by using the CLOSE command (suprise, suprise!). The form of the close command (its SYNTAX) is rather more simple than that of the open command:

CLOSE lfn

In the example above, this would be CLOSE 1.

If you should want to send information, and you have already OPENed the channel to your disk unit, and not closed it, trying to open the same channel again will cause your computer to display an error message on the screen – FILE OPEN ERROR. To avoid this, if you are sure that the channel is open, you can use the PRINT instruction. The syntax of this is:

PRINT lfn,"command string"

Where lfn again is the logical file number (1 in the example), and the command string is the same. This instruction still leaves the file OPEN, and so is very useful for sending lots of commands to the drive. If you are not sure whether or not the file you want to use is open, and wish to avoid the file open error, the simplest way is to issue the CLOSE command first, as above, and then OPEN it again.

Loading and Saving Programs

Once you have successfully formatted a diskette, it is ready for you to start using it to store data on. There are several different types of data, the simplest of which is a PROGRAM file. (Each entry on the directory is known as a file).

A program file consists of a series of blocks of information which, when read into your computer build up a program, hence the name. There are three commands in your computer's language which are designed specifically for use with this type of file. If you are only going to use commercial disk software, such as business or games, you need (normally)

only use the LOAD instruction, as everything else should be deal with by the program you use.

The SAVE instruction

Before you can load a program from disk, you must have a program there. Program files are created by the SAVE instruction, which has the following syntax:

SAVE "dr:filename",dn

We have already met the abbreviation dr for drive number in the OPEN command, this parameter will always be Ø, when using a single drive such as the 1541, but will become important should you decide to get a double drive at any time in the future. For the sake of completeness, we will therefore include references to the drive number, where applicable.

The filename is the name that goes onto the directory, and is the means by which you specify the program when you wish to load it again. It is a good idea to give any programs you save names which help you to remember what the program is. A filename can be any length up to 16 characters long. Device number (dn) will normally be 8, as in the open command.

To illustrate the save command, we shall write a short program, and store it on a disk. If you have no formatted disks, you should prepare one now as the drive cannot save anything otherwise!

Type this short program into your computer:

```
1Ø FOR J=1 TO 1Ø
2Ø PRINT "THIS IS A TEST !!!"
3Ø NEXT J
```

To save this program on disk, now type

SAVE"Ø:DISK TEST",8

The drive should whirr briefly, and then the READY message will appear on your screen. The program you have just typed in is now stored on the disk.

The commonest error at this stage is that the message "PRESS RECORD AND PLAY ON TAPE" will appear on your screen. This means that you have either forgotten the second quotation mark, or the ",8" at the end of the program name. In either case, press the RUN/STOP key, and retype your entry. Another possible mistake is the "? DEVICE NOT PRESENT ERROR" message, which simply means that either your disk drive is not connected to the computer, or that it is not switched on. If you get this error, check the connections, and that the green LED on the front of your drive is on.

If the red light on your drive is flashing at you, this means that some error has occurred in the drive, this may be that you have forgotten to put a formatted diskette in, or, more commonly, that you have tried to save a program with the sane name as one already on that disk. To overcome this, try to save the program again with a different title.

Once you have saved a program, you should always check that you will be able to read it again, just to make sure that it has been saved correctly. This is done using the VERIFY command. This command takes the form:

VERIFY "dr:filename",dn

This is, in fact, the same syntax as used in the save command, so the easiest way to use it is to edit the save command on the screen to verify. In this way you are sure that the filename and device numbers match. Use this command now, to check that your program has been correctly saved, by typing:

VERIFY "Ø:DISK TEST",8

Again, your drive will start up and the computer should display "SEARCHING FOR Ø:DISK TEST", "VERIFYING", "OK", and finally "READY." There are several possible errors here:

1. PRESS PLAY ON TAPE – as above.
2. ? DEVICE NOT PRESENT ERROR – as above.

3. ? FILE NOT FOUND ERROR – means that you have specified the wrong file name, or possibly changed the diskette.
4. ? VERIFY ERROR – this message tells you that the file with the name that you specified is on the disk, but that it differs from the program in the computers memory. This could be either because you have changed the program in memory (check this, if it is practical), or it could mean that the disk drive has been unable to save the program correctly. In either case, you should re-save the program in question, preferably with a different name.

Loading programs

The syntax of the LOAD command is the same as for the save and verify commands, but another parameter is sometimes added, giving the general form:

LOAD"dr:filename",dn,sa

This final term is the secondary address, which is used when loading machine-code programs such as Easy Script, to tell the computer not to treat it as a BASIC program. The reason for this is that BASIC programs are sometimes moved around in memory by the computer. Machine-code programs are normally designed to work only in one place, so they must not be relocated in this manner.

When loading a program, if you do not specify the secondary address, it will default to Ø, and then the program will be treated as a BASIC program. For machine-code programs, or other blocks of memory which must not be moved, the secondary address must be 1.

To see how the load command works for your BASIC program, first type NEW but be careful not to confuse this NEW with the disk new command. Typing NEW and pressing the RETURN key instructs the computer to wipe out any BASIC program that it has in memory at the time, whereas the disk new command formats the disk, wiping everything out!

Now type LIST and when you press RETURN the computer

should respond with READY, but not display anything else. This means that you have erased the program that you have just typed in and saved. If we wanted to get the program from the disk again, we would type the following:

LOAD "Ø:DISK TEST",8 <Return>

The drive will start up, and the computer will read the program into memory, while displaying such messages as: "SEARCHING FOR DISK TEST", "LOADING", and then finally, the familiar "READY.". At this point, type LIST again, and you should find that the computer has got the program that you saved. This can now be run in the same way that you would run any program loaded from cassette or that had been entered through the keyboard.

Possible errors which you may find are the same as those which you can get from the VERIFY command. You should check that you have got the correct diskette in the drive, and that you have spelt the filename correctly. Note that spaces are counted as characters, so the name "DISK TEST" is (as far as the drive and computer are concerned), totally different from the filename "DISK TEST".

Pattern-Matching

This is a very useful idea when reading disk files, either program files or the various types of data files which we shall be discussing later on. It entails the use of two special pattern-matching characters, which can take the place of part (or even all) of a filename. These characters are the asterisk (*) and the question mark (?). The ? replaces any one unknown character in a filename, so if you had a program that you know is either called "DISK TEST", or "DISC TEST", but were not sure which, you could issue a load command with the filename of "DIS? TEST". This would load the first file it came across whose first three characters are "DIS", and whose name carries on with " TEST". You may insert as many question marks into a filename as you wish; each one

will replace one character. We will find this feature useful in other disk commands, where we want to affect several files on a disk. A good example of this is the SCRATCH command, used to erase files on the disk which will be discussed later on.

The asterisk, when used as a pattern-matching character replaces a series of unknown characters on the end of a filename. If you type LOAD "ABC*",8 the computer will load the first program it finds on the disk whose first three characters are "ABC". Anything can follow, but these first specified characters must match. Often you will find that commercial disk based programs give instructions on loading as LOAD "*",8. This tells the computer to load the first program on the diskette, normally this is either a loader which loads the correct program for the particular thing you want to do or will be the application program itself. This means that when using these commercial systems you need not even know the name that the program you want was saved with.

It is useful to note that the VERIFY command uses the asterisk to specify "verify the last program saved" – useful in that whatever name you give a program when saving it, you can always issue the command:

```
VERIFY "*",8
```

Note that because of their use as pattern-matching characters, "*" and "?" are not allowed in a filename when writing a file, either a program or data.

The Directory

As we mentioned before, the disk unit keeps records of each file on a disk, in a special area called the directory. We can examine the directory from a disk by loading it into the computer's memory as a program and LISTing it on the screen, in the same way as a normal BASIC program may be displayed. Note that loading the directory into your computer in this way will cause the loss of any BASIC program you have

in memory at the time and, to avoid this, if you have a program you want to keep, save it on a disk first as we did with "DISK TEST" earlier. The directory can then be loaded into memory and displayed by using the filename "$". Do this now by typing:

LOAD "$",8

When the drive stops and the computer displays the READY message, type "LIST", and you should get a display something like this, which is the directory of the "Test Demo" diskette:

```
Ø  "1541test/demo " zx          2a
13 "how to use"                prg
5  "how part two"              prg
4  "vic-2Ø wedge"              prg
1  "c-64 wedge"                prg
4  "dos 5.1"                   prg
11 "copy/all"                  prg
9  "printer test"              prg
4  "disk addr change"          prg
4  "dir"                       prg
6  "view bam"                  prg
4  "check disk"                prg
14 "display t&s"               prg
9  "performance test"          prg
5  "sequential file"           prg


13 "random file"               prg
558 blocks free.
```

The first line gives the name of the diskette (1541test/demo), the identifier (zx) and a code which tells the computer that the disk is in the correct format for a 1541 drive (2a). The next series of lines are the actual records of the files on the disk, and consist of the length of each file (in blocks), their names in quotation marks, and finally the type of file. All the

files on the demo diskette are programs, so their filetypes are all displayed as "PRG". Other filetypes possible include SEQ (sequential), REL (relative) and USR (user) files, which are used to store data. We will be discussing the various types of files and their merits later on.

Occasionally, you may find an entry in the directory which has an asterisk in front of the filetype. Any file with this has not been properly created, and is therefore unusable. Any such entries should be deleted from the disk using the scratch or validate commands, instructions for which you will find in the section headed "DISK MAINTENANCE COMMANDS"

The final line of the directory is used to tell us how much spare space there is on the disk, in the example above there are 558 blocks spare. This gives us an indication as to how much more information can be put onto the disk before it is filled.

There is a program on the demonstration disk which enables you to view the directory of a disk (and perform other disk commands) without losing any program in memory. If your computer is a VIC 20, load and run the program "VIC-20 WEDGE" – the equivalent program for the Commodore 64 is "C-64 WEDGE".

Once you have the appropriate program in your computer, the use of the disk commands is greatly simplified, as the program automatically deals with opening and closing the communication channel to the disk drive. To display the directory of a disk with this program, simply type <$ – the drive will start up, and the directory will be displayed on your screen. A BASIC program in memory at the time will remain intact, which is a great advantage as it means that you need not save it before looking at what is on the disk.

This program is often referred to as "DOS SUPPORT" (DOS stands for Disk Operating System) and allows the easy entry of commands to the disk drive. Where normally you would type OPEN 1,8,15,"commandstring", with DOS SUPPORT the same effect is given by typing ">commandstring" or "@commandstring". (The > and @ signs are interchange-

able). Note that these commands only work in DIRECT mode, they cannot be used from within a program. Typing > or @ on their own will cause any disk error messages to be displayed on the screen. These error messages are generated by the disk drive when something goes wrong, for example if you attempt to save a program with the same name as one already on the disk. An error also causes the red drive light to flash on and off, this is so that we know if anything has gone wrong, and can take appropriate action.

CHAPTER FOUR

Disk Maintenance Commands

There are a series of commands which are used to perform various housekeeping tasks on diskettes and these are used by sending a command string to the drive, with the OPEN or PRINT# instructions from BASIC, or using the DOS SUPPORT program. In order to cut down the amount of typing required, all the disk maintenance commands may be abbreviated to their first letter, so where a command string could be "NEWØ:TEST DISK,32", it would have exactly the same effect as typing "NØ:TEST DISK,32". We will now discuss each of these commands and what they do.

NEW

We have already met the NEW command in the section on getting started, and know that it is used to format an unused diskette. The NEW disk command (not to be confused with the BASIC instruction: NEW) can also be used to erase a previously formatted diskette, if you do not want any of the information that was on it.

There are two distinct forms of this instruction and in the section on getting started we discussed how to format a completely unused disk using the syntax

OPEN 1,8,15,"NØ:diskname,ID"

If, however, the disk has been previously formatted, you

may leave out the ID, so the command string becomes "N0 :diskname". This may not appear to have any advantages, but it means that the drive knows the disk has been used previously, so it does not have to create each block individually. In fact, all it has to do is change the directory and BAM, a process which takes much less time. Remember, though, that to format a brand new disk, you must always specify the identifier, otherwise the disk unit will indicate an error condition.

Initialise

This instruction is used to tell the disk drive that you have changed the disk in it, and forces it to read the BAM into its internal memory: if you were to change the disk, and not issue this command there is a possibility that the drive would not realise this. If you then tried to write any data to the disk, the drive would use the block availability map from the previous disk and may overwrite data on the new disk. Before any operation which requires the drive to write on a disk, it automatically checks the ID of each block, if this does not match the drive will not write the data, but flash the red error light. By reading the error channel from the drive it is possible to decide which error has occurred, and so what action to take. In the example of an un-initialised diskette, the error reported would be "DISK ID MISMATCH".

However, if the two disks were formatted with the same ID's, this error checking process would fail, and data on the second disk will probably either be totally lost, or worse still become corrupted, without you realising that you have just destroyed last years accounts!

In order to avoid such problems, there are two simple rules, namely **When formatting diskettes, use unique ID's and** after changing diskettes, always INITIALISE the disk drive. Follow these rules and you will minimise your risks of having any problems.

The syntax of the initialise instruction is fairly straight-

forward, the command string is "INITIALISE dr", where dr is the drive number. With the Commodore single disk drives the drive number may be left out because it will always be Ø (leaving the drive number out will cause the disk unit to operate on drive Ø), so the most abbreviated form of this command to the disk unit would be "I". The complete command, then would be:

OPEN 1,8,15,"I", or PRINT#1,"I" (if the command channel 1 is open), or if you are using DOS SUPPORT, >I or @I.

Scratch

This command does not, as one might think, cause the disk drive to relieve an itch in some part of its mechanism, but is used to delete unwanted data from a diskette.

You may have decided that a particular program on a disk is not worth keeping (most Space Invaders games would qualify!) and you want to delete it to free some space on your disk. Another reason you may want to erase a file is if you have altered a program (or a data file) and then wanted to store the new version on the same disk, using the same name as before. We have already said that the SAVE command first checks that there is no file with the same name as the one you give, so that there is no chance of losing anything.

Before you can re-save anything with the same name, therefore, it follows that you must delete the old version first. This is done using the scratch instruction, which has this structure:

OPEN 1,8,15,"SCRATCHØ:filename", or
PRINT#1, "SCRATCHØ:filename", or
>SCRATCHØ:filename(@SCRATCHØ:filename),for
those of you using DOS SUPPORT.

The word SCRATCH can be shortened to "S", and it is permissable to use pattern-matching characters in the file-name, although you should be aware that *all* files matching that pattern will be deleted. This feature is particularly useful

if you have several files, perhaps called "VERSION 1", "VERSION 2", etc. By using the filename "VERSION ?" or "VERSION*", all these files could be deleted in one fell swoop! Reading the error channel will tell you how many files have been deleted, in the 3rd parameter read.

While we are discussing the SCRATCH command, I would like to point out that there is a special case of SAVE, which has the form SAVE"@Ø:filename",8. The effect of the "@" before the drive number is to SCRATCH any existing file of the same name and then replace it with the one you are saving. This appears at first sight to make the saving of programs much easier, but there have been numerous reports of an error in this instruction, which sometimes unpredictably corrupts the disk. It should therefore be avoided, and the sequence of first SCRATCH, then SAVE followed, as both of these commands are reliable!

Validate

After deleting and re-saving several times, the blocks in each file become spread around the disk, and although the files are still treated as discrete units, the time taken to read them will tend to increase. This is because the read/write head has to travel further to get the next block of data and this is especially noticeable when the diskette in question is almost full, the disk unit may have to split a file up in order to fit it into the gaps left between other files. Although the data will read as intended, this process may take longer than usual. Commodore have thought of this, and included the VALIDATE command, which is designed to tidy up the file structure on the disk. Any corrupt files are deleted, and the space they were occupying is freed for use.

When you want to validate a disk, simply send the command string "VALIDATEØ", or "VØ", to the disk drive, using the command channel. The drive will run for a while as it does its spring-cleaning. The length of time taken depends on how much tidying has to be done, and in exceptional cases,

where the disk has been used for a long time without validation, may take up to 1Ø minutes! This may seem like a long time, but is well worth it bearing in mind how much faster it makes access to the disk. Before you go off and validate all your disks, you should not validate any disk which has a type of file known as "RANDOM ACCESS" (shown on the directory as USR), as these will be destroyed by the validate command. This is because they do not have the same structure as other files, and there is usually no record of them in the BAM. For further information on these, see the chapter on random files.

Rename

This command enables you to change the name of any file on the disk, without altering its contents, in the same way that you might peel a label off and stick a new one onto something. Not particularly useful in itself, this command comes into its own in business programs which deal with large data files, where it is impractical to read a whole file into the computer, and write the same data out again to a different file.

The command string to send to the drive in order to rename a file has the general form "RENAMEØ:new name=old name", or "RØ:new name=old name".

Copy and Concatenate

This command allows you to duplicate a file on disk, or even create multiple copies with different names, on the same disk. This feature would be used if you wanted to alter a file, but keep a copy of the original version for future reference. The string to send through the command channel to the drive to accomplish this would have the form "COPYØ:newfile=Ø :oldfile". There must not be anything on the disk with the name "newfile", or the drive will refuse to execute the command, giving an error condition (FILEEXISTS).

Occasionally, you may want to add two files together, to create one larger one containing the data of both the originals. This can be done using a special form of the COPY

command, known as CONCATENATION.

Suppose we had three files, called "FILE1","FILE2" and FILE3", and we wanted to join them together to give a large file with all their data, called "BIG-FILE". To do this we would send the following command to the disk drive:

```
OPEN 1,8,15,"C0:BIG#G-F-FILE=0:FILE 1,0:
                    FILE 2,0:FILE 3":CLOSE1
```

The commas separating the three filenames are taken by the drive as meaning plus, so we are telling it in the above example to create a file called "BIG-FILE" from FILE 1 plus FILE 2 plus FILE 3. It is worth noting that the file names should be kept fairly short in this type of command as the disk drive can only accept command strings which are 58 characters or less. If your command string exceeds this, the drive will not accept it, and reading the error channel will give "31,SYNTAX ERROR,00,00", which means that the drive could not cope with the command. If this problem arises, the same effect can be got by splitting the operation up into smaller steps, like this:

```
OPEN1,8,15,"C0:TEMPFILE=0:FILE 1,0:FILE 2"
PRINT#1,"C0:BIG-FILE=0:TEMPFILE,0:FILE 3"
PRINT#1,"S0:TEMPFILE"
CLOSE1
```

Remember to scratch any temporary files from the disk if they are no longer required, as they will only take up space on the disk which you may need at some later stage.

CHAPTER FIVE

Storage of Data: Sequential Files

So far, we have been talking about the preparation and maintenance of disks, and the simpler ways of using them such as saving and loading programs, rather than how to get the best use from them. In the next few chapters, we will discover how to create and access the various types of data files, and hopefully you will begin to see the vast power of your disk drive.

We will start by looking at the simplest type of data file available on your system, known as the SEQUENTIAL FILE.

As the name suggests, items of data in a sequential file are stored one after the other, although their physical positions on the disk may not be. The system of pointers which we mentioned earlier ensures that the data is read from the disk into the computer in exactly the same order as it was stored. There is another filetype called USER files, but these are identical to sequential files in all but the filetype shown on the disk directory. To use them, treat them the same as sequential, but replace "s" by "u" when specifying the filetype.

The data which you store in a sequential file can be either numeric, or string data (text), and your computer has several commands built into its language designed to manipulate this information.

Before a data file can be used, we must first open a communication channel to the disk drive, in the same way as

we opened the command channel to send instructions to it. As before, we use the OPEN statement in BASIC, which you may recall has the general form:

OPEN lfn,dn,sa,"dr:filename,ft,mode"

Where lfn is the Logical File Number, used to specify that channel in subsequent instructions (this can range from 1 to 255). Dn is device-number, which in the case of your drive will normally be 8. Sa stands for "secondary address". Previously, when we opened the command and error channels, this was always 15, but when specifying that you want to pass data to the drive (or read data), the secondary address can range from 2 to 14 (secondary addresses Ø and 1 are reserved by the computer and disk drive for loading and saving programs).

The drive number, as before will be Ø if you are using a single drive (this can be left out if you wish), and the filename refers to the actual name which appears on the directory (pattern-matching may be used when reading, but not writing). The next parameter (ft) is the filetype, which in the case of sequential files will be SEQ, (or just S), but can also be PRG (Program file), USR (User defined file), or REL (Relative). These filetypes correspond to the right hand column displayed on the directory. Mode specifies whether data is to be written onto the disk in which case it is WRITE (or just W), or read from the disk, when mode is set to be READ, (R).

As with the save command, there is a special case of the OPEN command, where the drive number is preceded with an "@" sign. When opening the channel in order to write data to the disk, this extra character tells the drive to delete any file with the same name. Unfortunately, this risks the same error as the "SAVE @" instruction, and should be avoided to ensure a quiet, trouble free life!

Once the file (data-channel) has been opened, we are ready to start reading or writing data to the disk. The three commands provided in BASIC to do this are:

1. PRINT# lfn,data – to write to a file opened with mode set to WRITE,

2. INPUT# lfn,variable – to read a series of characters from a file which has been opened to read, and
3. GET# lfn,variable – to read a single character from a file opened to read.

When you have finished reading the file you are interested in, or stored all the data you intend, it is important to tell the drive this, so that it can update the BAM, and tidy up the end of the file.

The instruction to do this is the direct opposite of the OPEN command – CLOSE. This has the syntax CLOSE lfn, where lfn is the number that you specified in the OPEN statement.

Error checking and ST, The Status Variable

When communicating with your disk unit, various things may happen which can cause serious problems if nothing is done about them. If you have read what has gone before you will know that the command channel (secondary address 15) can be used to read information about the status of the disk unit. When an error condition occurs, the red drive light flashes, and will continue to flash until the error channel is interrogated. This may be annoying, but it ensures that we know that something has gone wrong, and gives us a chance to do something about it!

It is always a good idea when writing programs which access any disk files to open the command channel first, so you can give the disk unit control instructions and detect any errors which occur, in order that your program may take appropriate steps to protect data which could otherwise be lost. After each step which reads or writes to the disk, you should then check the error channel to make sure that everything is going smoothly. This is best done by incorporating something like this in your program:

```
10 OPEN 1,8,15:REM OPEN DISK COMMAND/ERROR
CHANNEL
```

The main part of your program will go here.

```
60000 REM *** READ DISK ERROR CHANNEL
60010 INPUT#1,ER, ER$,ET,ES
60020 IF ER<20 THEN RETURN:REM *** EVERYTHING IS
OK, SO CARRY ON.
60030 PRINT "DISK ERROR NUMBER ";ER;" MEANS ";ER$
60040 PRINT "AT TRACK ";ET;", SECTOR ";ES
60050 PRINT "IS IT OK TO GO ON ? (Y/N)"
60060 GETA$:IF A$="" THEN GOTO 60060
60070 IF A$="Y" THEN RETURN
60080 IF A$<>"N" THEN 60060
60090 PRINT "PROGRAM ABORTED"
60100 CLOSE1:END
```

This routine, when called, reads the error channel from the disk drive, and reports on any error conditions detected. It then asks the user whether it should allow the program to continue. If the operator presses "Y", then the program is allowed to continue, but if the operator considers that the error is liable to destroy any data on the diskette, he should press "N". If this is the case, the program closes the command channel, which also closes any files that were left open on the disk. This means that even if one file has become corrupted because of the error, any others should survive, thus minimising the damage.

Obviously, this routine may not be satisfactory for any one specific application, but the basic idea is there, and you should have a routine of this type in all of your programs. In order to help you decide whether an error is fatal or not, you will find a list of the possible errors reported by the disk unit at the back of this book, along with a short description of each.

When you read a sequential or user file from a diskette, unless you know how many data items there are, or have marked the end of the file with some sort of code there is a chance of reading past the end of the file and any data read in when this happens would be totally meaningless.

When you reach the end of a file, however, the STATUS VARIABLE, ST is set to 64. This means that by checking the

value of this variable, you can detect when you have reached the end of a file. When this happens, you should CLOSE the file.

Now that we know about error checking, and how to detect when we have reached the end of a file, we are in a position to start putting what we have learnt into practice by writing a short program to store some information in a sequential file, and then read it into the computer again.

Enter the command channel opening and error checking routine listed above, and then type in the following program:

```
20 OPEN8,8,8,"0:DATAFILE,SEQ,WRITE"
30 GOSUB 60000 : REM CHECK ERROR CHANNEL
40 PRINT "ENTER DATA – (END TO FINISH)"
50 INPUT A$:IF A$="END" THEN 80
60 PRINT#8,A$;CHR$(13);:GOSUB 60000
70 GOTO 50
80 CLOSE 8:GOSUB 60000: REM*** CLOSE THE FILE NOW
WE'VE FINISHED WRITING!
90 REM *** NOW READ THE FILE AND DISPLAY IT
100 OPEN 8,8,8,"0:DATAFILE,SEQ,READ"
110 GOSUB 60000
120 INPUT#8,A$:LET X=ST:REM *** X WILL BE USED TO
CHECK FOR END OF FILE.
130 GOSUB 60000
140 PRINT A$:IF X=0 THEN 120
150 REM *** WE'VE GOT TO THE END OF THE FILE!
160 CLOSE 8:GOSUB 60000
170 END
```

This program allows you to type lines in at the keyboard, bearing in mind the limitations of the BASIC INPUT command, and store them in a sequential file on the disk, called "DATAFILE". When you enter "END", the program will close the file, and then re-open it to read. The file must be closed and then re-opened, as the MODE is changed from write to read for attempting to input data from a file opened to write would cause the error message? NOT INPUT FILE to be

displayed by the computer.

If you run the program as it stands a second time, you will find that the error checking routine detects that you are attempting to create a file which is already on the disk. When this happens, you should stop the program by pressing "N", and add in an extra line to delete the file before you create it again.

```
15 PRINT#1,"S0:DATAFILE"
```

This line deletes any old file with the same name, thus avoiding the "FILE EXISTS" error.

In the reading part of the program, on line 120, the status variable ST is used to set X, another variable. This is done because the PRINT instruction used to display the data read from the disk would reset ST to 0, so we would never be able to detect the end of the file! Instead, by setting another variable (X) from ST, and then checking that after printing the data, we can successfully decide when we have reached the end of the file.

You may be wondering why line 60 in the above example is sending the data and then "CHR$(13);". CHR$(13) is the code for a carriage return, and is used as a delimiter between separate items of data. In the same way as appending a semi-colon (;) on a PRINT statement to the screen stops new data being printed on the next line, putting a semi-colon at the end of a PRINT# statement, as in line 60 ensures that no extra unwanted characters are written to the disk, such as line-feeds, which are sometimes sent by the computer. This is a method of ensuring that the minimum amount of disk space is taken up to store the information. If the semi-colon was replaced in this line by a comma, for example (which spaces data out when printing to the screen), the data on the disk would be padded out with spaces. As these spaces are not part of the information you wish to store, they would waste space on the disk. Line 60 therefore makes sure that text you enter at the keyboard is stored in the most efficient way possible in the sequential file.

In certain applications, you may want to read a sequential file which contains various characters which act as delimiters to the INPUT# instruction, such as commas, colons, semi-colons and carriage returns, or other control characters which would not be easy to detect, (such as colour control characters).

We mentioned before that there is another statement in the BASIC vocabulary for file handling, GET#, which reads a single character at a time (in the same way as GET reads a single character from the keyboard). If we replaced the INPUT#8,A$ in line 120 which reads a whole string into A$ by GET#8,A$ the program would read the file a single character at a time. The PRINT statement in line 140 must then be followed by a semi-colon (otherwise each character read in would be printed on a new line).

The advantage of reading a file a single character at a time is that as each one is read into the variable (A$), we can check it individually. This allows us to do other things which would normally be impossible using the straightforward INPUT# instruction. As an example of this, numbers can be stored in a compacted form, taking less space on the disk. Suppose we wanted to store a series of prices or values ranging from 0 to 500.00 on a disk. If we were to use the INPUT# command to read them back, the disk file would have to have a structure something like this:

324.54[CR]87.91[CR]0.21[CR] – etc.

Each number in this example is taking up to six bytes in the disk file (6 characters), and each number must be separated by a delimiter character, in this case [CR] which is a carriage return, or CHR$(13).

Using the GET# instruction to read the data it is possible to make vast savings in disk space. In fact the same numbers could be compressed to two characters, and we could dispense with the delimiting carriage return. A certain amount of processing must be done by the computer to compact the data before sending it to the disk drive and when

reading it back, but this is easily done by using these two short subroutines, the first of which takes the number in variable A, and compresses it, using the CHR$ function into a two character string in A$ to send to the disk. The second subroutine will unpack a two character string (A$), and return with the original number in the variable A.

```
1000 REM *** COMPRESS "A" INTO "A$"
1010 A=INT (A*10):A1=INT(A/256):A2=A-(256*A1)
1020 A$=CHR$(A1)+CHR$(A2):RETURN
```

Now we can send the compressed number to a previously opened sequential file with the statement PRINT#8,A$; – note that we no longer need the delimiting "CHR$(13);" in this instance.

To read the data back in, we would have a routine something like this:

```
120 GET#8,A1$:IFA1$=""THENA1$=CHR$(0)
121 GOSUB 60000:REMEMBER THE ERROR CHECK!
122 GET#8,A2$:LETX=ST:IFA2$=""THENA2$=CHR$(0)
123 GOSUB 60000:REM ERROR CHECK
124 A$=A1$+A2$:GOSUB 2000:REM CALL UNPACKING
ROUTINE...
2000 REM*** UNPACK DATA FROM A$ INTO A
2010 A=(256*ASC(LEFT$(A$,1))+ASC(RIGHT$(A$,1)))/10
2020 RETURN
```

For more information about the BASIC functions used in these routines (CHR$, ASC, RIGHT$ AND LEFT$), see the instruction manual which came with your computer, or the Programmer's Reference Guide relevant to your machine.

As each character is read from the disk, there is a specific test for the null string(""). This is because CHR$(0) is read in as an empty string, which would cause the error message "? ILLEGAL QUANTITY ERROR" when we called the unpacking routine at line 2000.

Remember to check the error channel each time you access the disk unit as before, in order to avoid reading

spurious data in.

Data types in Sequential Files

As you may already know, your computer can deal with either text or numeric data by using different variable types. These are strings (followed by a $ sign), simple variables and integer variables (which have a % suffix).

Each of these types of variable can be handled by your disk drive and can be stored in sequential files although you must remember to specify the type that you are trying to read from the disk in the variable name you specify in each INPUT# or GET# command. If you are in doubt, always read the data into a string variable, as trying to read a piece of text into a numeric variable will cause the error message "? TYPE MISMATCH ERROR" to be displayed, and your program will stop. If this should happen, make sure that any files left open on the disk are closed properly, by closing the command channel. Fail to do this and you run the risk of not being able to read your data again! Reading numeric information is not too bad, as you can convert the string into a numeric variable using the VAL function in BASIC.

We recommend that you keep a note of the order in which you write data out to a file (the file structure), when writing programs, so that you know what type of data to expect when you come to reading it. This also helps you to decide what each item is.

Normally, a file will consist of a series of RECORDS: each record may then be further sub-divided into several items, known as FIELDS. Usually a file is designed so that the INPUT# command will read one of these fields, so by keeping a count of the number of fields it becomes relatively simple to decide what type of data will be coming from the disk next. To help with the reading or writing of sequential files, the disk drive has pointers which tell it where within the file it has got to. The user does not normally have access to these pointers, which is why this particular filetype is called "SEQUENTIAL" – the disk drive can only follow it through

from the start. Each time an INPUT#, GET# or PRINT# instruction is sent, these pointers are updated so that the drive knows where on the disk to go for the next item of information.

A more powerful file structure would be one in which we could both read and write data into the middle of the file, without having to start from the beginning and work our way through to the point we want to get to. Large savings in time could be achieved, especially when dealing with large amounts of data.

CHAPTER SIX

Storage of Data: Random Access Files

This idea of being able to start where we want in a disk file leads us on to the next type of storage available to us while using a Commodore disk drive: these are known as RANDOM ACCESS FILES.

By dispensing with the sequential file format we no longer need the pointers set up by the disk operating system (DOS) at the start of each block in the file. However, this means that we must keep some form of plan of our file so that we know which particular block to read in order to gain access to a specific record. This is usually done by using a sequential file which is read into the computer at the start of a program, which is then used to find the way around the actual file containing the data. This sequential file is usually termed a KEY-FILE, or INDEX-FILE, as it holds the KEY or INDEX for the random access file.

In order to facilitate access to specific blocks on a disk, the DOS has been programmed to understand a series of instructions which either act on the internal memory of the disk drive, or cause data to be read from or written to the diskette itself. These commands are passed to the DOS by the computer by means of the PRINT# instruction to the

COMMAND CHANNEL (channel number 15). Inside your disk drive there are a total of sixteen different areas of memory, known as buffers, and the drive actually writes data to these buffers and from there it is transferred to the diskette itself. Similarly, when reading a file, what is actually happening is that the drive reads a block of data into a buffer, and passes it to the computer from there. In an OPEN statement in BASIC, the secondary address is really the channel number used by the DOS to refer to a specific buffer.

The disk operating system reserves channel numbers Øand 1 for use with the LOAD and SAVE commands, so secondary addresses less than 2 should be avoided unless you specifically require the LOAD or SAVE functions. This is very unlikely, as you already have the load and save commands in BASIC. Channel number 15, which we have already used, is the error and command channel to the DOS, but can also be used to send instructions directly to the disk operating system, as we shall see later. All the other channels (2–14) are available to the user to pass data between computer and disk drive buffers.

These channels are allocated by the OPEN command (in the secondary-address). If a channel is to be used for direct access purposes, the filename specified in the OPEN command must be "#": this symbol is called a hash-mark, not a pound sign as the manual says. A specific buffer may be allocated to that channel by appending a number to the # in the filename. The statement OPEN3,8,7,"#" would therefore open channel number 7 and allocate the first available buffer to it. The file would be accessed by PRINT#, INPUT# or GET# statements using a logical file number of 3. If, for some reason, we wanted to allocate buffer number 1 to this channel, the open statement would be OPEN 3,8,7,"#1". In the event that buffer number 1 has already been allocated by another OPEN statement, the DOS will give a "NO CHANNEL" error condition. (See the section on ERROR MESSAGES GENERATED BY DOS.)

Once a channel to the disk drive has been opened in this way, we can start to communicate with its operating system using the commands in the DISK UTILITY COMMAND SET, which we will have a look at now, and reading and writing data directly onto the diskette.

These commands are passed to the drive by means of the command channel using the PRINT# statement, in the same way as the disk maintenance commands NEW, COPY, RENAME, etc., and may be abbreviated to the first two characters of the keywords to save space, should this be desired. Thus the first instruction we shall look at could be given as "BLOCK-READ"ch;dr;t;s or "B-R"ch;dr;t;s : both statements are interpreted by the disk operating system as meaning the same thing.

"BLOCK-READ:"ch;dr;t;s

This instruction is used to tell the disk drive to go to a block specified by the track and sector terms t and s, on drive number dr and read it into the buffer corresponding to channel number ch inside the disk drive's memory (not into the computer!). Once the data is in the drive's buffers, it may be manipulated by the use of PRINT#, INPUT# or GET# statements, through the data channel (the one we OPENed with the filename "#"). There is another command used to read a specific block of data: the USER1 command. See the section dealing with this, as there are some slight differences in the way it operates.

"BLOCK-READ:"2;0;27,5

("BLOCK-READ:" may be shortened to "B-R:"). This example would tell the DOS to go to track 27, sector 5 on drive 0, and read the data in that block into the buffer allocated to channel number 2.

"BLOCK-WRITE:"ch;dr;t;s

As the name implies, this instruction tells the drive to write the

data which is in the buffer allocated to channel number ch to drive number dr (this is always Ø when using a single disk drive). The data is written to the block specified by the track and sector parameters, t and s. These two instructions form the basis of RANDOM ACCESS files, as they provide the computer with the means of reading or writing to any given block on a diskette. Compare this with the USER2 (U2) command, discussed later.

"BLOCK-EXECUTE:"ch;dr;t;s

This instruction causes the block specified by the parameters dr,t and s to be read into the buffer associated with channel number ch, and run, as a 65Ø2 machine-code program inside the disk drive's memory. This command is useful when a particular function you require is not implemented normally by the DOS, and is often used as part of anti-copying methods employed to protect commercial diskettes from illegal duplication. The routine loaded and run in the drive's memory in this way must end with a return from subroutine (RTS) instruction, or the disk operating system will probably crash in the same way as your computer is liable to when using machine-code routines which do not hand control back to BASIC. If the routine is position dependant (as machine-code often is), a specific buffer may be allocated to the channel in question when OPENing that channel, by using the filename "#buffer".

"BUFFER-POINTER:"ch;p

The DOS has internal pointers which tell it where, within each buffer the next character to be affected by PRINT#, INPUT# or GET# is. A program can set the position of the pointer for each channel to anywhere within a range from Ø to 255. This command allows us, once a particular block has been read into the disk buffer, to set this pointer to any given byte in that block, and hence any specific byte on the disk. Once the buffer pointer has been set, the next GET#, PRINT# or

INPUT# instruction to the direct access channel will affect the data pointed at by the pointer. Suppose that we wanted to know what the 31st character is in the block at track 25, sector 3. First, we would open the command channel, with OPEN1,8,15, and a direct access file, with a statement such as OPEN 3,8,3,"#". The next stage would be to read the specific block we wanted into the buffer allocated to channel 3, the one we have just opened, using the statement PRINT#1, "B-R:"3;Ø;25;3. At this point, the data from the block we want is read into the buffer, but the pointer will be pointing at the first byte of the buffer. This is set to point at the 31st byte (which is numbered 3Ø, as they start with byte Ø) by issuing the statement PRINT#1,"B-P:"3;3Ø. The actual character at that location can then be read into the computer by the GET# command, thus – GET#3,A$. If we wanted to alter the character at that location on the diskette, the sequence would be:

1. OPEN the channels, as before.
2. Read the block into the buffer, using the Block-Read command.
3. Position the pointer to the required location within the buffer.
4. Use the PRINT# statement to write the required data to the disk buffer, and then
5. write the block back to the diskette, by means of the Block-Write command.

You must remember to close the channels when you have finished with them, just as when writing sequential files.

"BLOCK-ALLOCATE"dr;t;s

The Block-Allocate instruction is used to tell the disk operating system that a specific block is not to be used when writing sequential or program files. This may be because you have stored data in that block using the block-write command and do not want to run the risk of it being overwritten by future data. It actually works by setting the block allocation map

(BAM) to indicate that the requested block is in use. If the block has already been allocated, the DOS will return the error condition 65, NO BLOCK. The track and sector of the next highest free block are also returned (see the section on error messages).

These, then, are the commands used to operate directly on the diskette and data in associated buffers. There are three commands which can be sent to the disk drive which operate directly on its internal memory in the same way as the commands PEEK, POKE and SYS are used in your computer from BASIC.

As in the case of the previous instructions, they are sent to the DOS via the command channel as a string of data, where they are decoded and executed. However, in the case of these commands, the DOS will only understand the abbreviated forms of the commands, and data must be sent in the form of characters rather than numerics, so to send the number 1Ø, we would have to send CHR$(1Ø) in the command-string.

Memory-Write

The memory-write instruction allows data to be stored in the disk memory, in the same way as POKE does in BASIC. Your drive has 16K of Read Only Memory (ROM) and 2K of RAM. In the same way as you cannot alter the contents of a ROM in your computer, data may only be written to the RAM in the drive unit. This data is usually a machine-code program, which will later be executed by the memory-execute command (analogous to SYS), or one of the USER (U) commands. For details of these commands, see the relevant section, below.

This instruction takes the general form of /M–W:"adl" adh-/nc-/data, where adl and adh are the low and high bytes of the address where the data is to be stored, nc stands for the number of characters, and data is the actual information you want to put there. Note that each of these parameters must be in CHR$ codes.

An example of this command would be:

```
"M-W:"CHR$(0)CHR$(16)CHR$(3)CHR$(108)
CHR$(252)CHR$(255)
```

This instruction would send the three bytes 108, 252 and 255 and store them in the disk drive's memory. This would be the equivalent to the single 6502 machine language instruction JMP($FFFC). This is the power-up vector; when the drive is first switched on its 6502 does this anyway, so the instruction would simulate the drive being turned on. This function is also available through the "UJ, or U:" command (see the relevant section). A maximum of 34 characters can be sent to the drive in this way, but successive M-W instructions can be used to build very complex programs up in the DOS memory.

Memory-Read

This function is the reverse of memory-write, in that it can be used to read information from the DOS memory. It is the DOS equivalent of the PEEK instruction, and can be used to examine the contents of either the RAM buffers or the ROMs containing the operating system. Data is read into the computer through the command channel, using the GET# statement, where it could be dis-assembled to provide information about how the DOS works. The format of the string sent through the command channel is "M-R:"adl/adh, where adl and adh are the CHR$ representations of the low and high bytes of the address you wish to examine. These address bytes are arrived at by the formula:-

```
adh=CHR$(INT(address/256)), and
adl=CHR$(address-(INT(address/256)*256)).
```

If we wanted to read the disk controllers memory, starting from location 4100 ($1004 HEX), the low part of the address would be 4 and the high part 16: here is a short program to read that location.

```
10 OPEN1,8,15
20 PRINT#1,"M-R:"CHR$(4)CHR$(16)
30 GET#1,A$:IF A$=" " THEN A$=CHR$(0)
```

```
40 PRINT"THE VALUE IS ";ASC(A$)
50 CLOSE1
```

Different locations can be examined by changing the values in line 20, but a more flexible approach would be to use variables in the command. The contents of a series of locations can be read by executing more GET# instructions, as in Commodore's example on page 38 of the manual. Once the actual data has been read it may then be processed in any way you wish, for instance you could disassemble it or display the ASCII codes depending on what you need to know about the area of memory you are looking at.

As the manual states, using the INPUT# command with this is liable to give odd results. This is because any delimiting characters in the data (carriage returns, commas, colons or semi-colons) will be treated as meaning "end of input", so they will never appear in the input string, which will be an indeterminate length. If there are no delimiting characters, then the INPUT# statement would attempt to read for ever, the result of this would be either the system "hanging up", or eventually you would get ?STRING TOO LONG ERROR. As this situation is undesirable, you should always use the GET# statement with this command.

Memory Execute

This instruction performs the same function as SYS in BASIC, but rather than running a machine-code program in the computer's memory, runs one in the disk unit's memory. It is used by sending a command to the DOS in the form:

PRINT#1,"M/E:"adl-adh, where logical file number one has been previously OPENed as the command channel, and the terms adl and adh are the CHR$ representation of the address, as in the other MEMORY commands. The program run in this way may be anywhere in the drive's memory, RAM or ROM, so this allows you to use routines built into the operating system.

The USER functions

The final series of instructions understood by the DOS are the USER functions, some of which perform specific operations on the disk while others are, as the name suggests user-definable.

In general, they are called by sending a string to the DOS via the command channel consisting of "U" (for "user function") and a character specifying which function is required. This character may be either a single digit between 1 and 9, a letter from A to J or a colon, plus sign or minus sign.

USER1 (U1 or UA)

This instruction does the same as the BLOCK-READ function, but operates in a slightly different way. This difference is in the way each command deals with the buffer-pointer. Block-read will read data from a specific block until a pointer within that block tells it that there is no more data to come. At this point, Block-Read will stop reading data, bad luck if there was really anything more to come! U1 forces this pointer to 255, before reading the block, and so ensures that the whole block is read into the appropriate buffer. As with the B-R: command, we must give U1 parameters for the buffer it is to read the data into, the drive number, and finally the track and sector numbers for the block we want to read. If we wanted to read the BAM (Track 18, Sector Ø) into the buffer allotted to channel number 3, we could issue the command:

```
PRINT#1,"U1"3;Ø;18;Ø
```

In the same way as with the Block-Read command, the data read into the buffer can now be passed onto the computer by using the GET# or INPUT# commands, along with buffer-point where necessary.

All the User commands can be replaced by string variables (provided that they have been previously set up), so the same command could be given as:

A$=U1:PRINT#1,A$;3;0;18;0 or
A$="U":B$="1":PRINT#1,A$;B$;3;0;18;0, or even
X=49:PRINT#1,"U";CHR$(x);3;018;0

In the last example, the variable X is set, and sent as part of the string using the CHR$ function. 49 is the ASCII code for the digit "1". The important thing to remember is that the DOS is expecting the commands to be sent to it in the form of a string, not numeric data. Having said that, I should point out that the following parameters are numeric, sending these in the form of strings will cause the DOS to give you a SYNTAX error because it would not be able to understand! The "U1" string can also be sent as "UA" – the two are interchangeable, although it makes understanding listings of programs simpler if you stick to one or the other. These commands are complicated enough to understand anyway; no need to make life more difficult than it is!

USER2 (U2 or UB)

This instruction is a replacement for the Block-Write command. As with U1, U2 does not alter the value of the buffer-pointer already stored on the diskette. The main use for this command is when the buffer-pointer in the disk memory would be altered. Suppose we wanted to change a single character half way through a block. Using the B-R:, B-P:, PRINT# and B-W: instructions would do this, but the end of block pointer would be set to the character we had just written: if there was any other data after that point it would be lost because future block-read operations would only read as far as the pointer. The U2 (UB) command avoids this problem by leaving that pointer in its original position, regardless of what happens to the pointer in the buffer. So, to achieve the same result as our previous example without risking losing any data, we would use B-R:, B-P:, PRINT# and finally U2.

The parameters required by the U2 command are the same as those for the Block-Write command, so the syntax of the command is:

PRINT#1,"U2"channel;drive;track;sector

As with U1, UB is a valid alternative to U2.

User functions 3 to 9

This series of functions can be used to call machine-code routine residing in the drive's memory, and work by causing the DOS to go to specific locations in its memory. By storing JMP instructions at these locations, the DOS can be directed to anywhere in its memory. The actual addresses that are jumped to for each command are given below in hexadecimal

U3 (UC) Jumps to $0500
U4 (UD) Jumps to $0503
U5 (UE) Jumps to $0506
U6 (UF) Jumps to $0509
U7 (UG) Jumps to $050C
U8 (UH) Jumps to $050F
U9 (UI) Jumps to $FFFA
U: (UJ) Jumps to the reset vector.

As the addresses for U9 and U: (colon is the next ASCII character after "9") are both in the ROM of the disk-unit, they cannot be altered, but the others are in RAM, so they can.

The U9 (UI) command is used to alter the speed of the drive. This is because the VIC 20 and CBM 64 are programmed to accept data from the disk at slightly different rates (Why, I don't know!). This command needs to know whether it is expected to set the speed for a VIC 20 or CBM 64: adding a "+" to the command sets it up for the 64, while adding a "−" tells the DOS that it has to give data in a form understandable to the VIC 20.

U: (UJ) causes the DOS to jump to the reset vector, which is where it automatically goes when first turned on. This means that it executes all the routines that it does when first powered up. It is therefore a software method of effectively turning the drive off and on again. If you use this command, any open files are not closed on the diskette and the BAM is not written back, so extreme care should be taken.

CHAPTER SEVEN

Storage of Data: Relative Files

As previously mentioned, there is another type of file structure available for data storage with your 154Ø/1541 disk drive, known as RELATIVE FILES.

This system is more convenient than the random access file method, because the DOS does a lot of the pointer handling for you, so we do not have to create extra keyfiles in the way we did with random access files. By specifying the size of each record in a relative file, while creating it, we can instruct the DOS to read specific records later, simply by giving it the record number! The DOS is able to do this, because when a relative file is created (or enlarged) it decides exactly where each record is going to go on the diskette. It allocates those blocks in the BAM (so they are not accidentally overwritten), and then stores pointers to each record in the file, in a series of blocks associated with the file, called SIDE SECTORS. There can be up to 6 side sectors in a relative file, each one pointing at the start of a record in the file. A simple calculation shows us that each file could contain up to 72Ø records (6 side sectors x 12Ø records per sector): each record in a relative file can be up to 254 characters long. Unfortunately, this would take 72Ø blocks for the data, plus 6 more for the side sectors which go with the file, which is more than the capacity of a diskette! We can therefore totally fill a diskette up with one relative file (although more are allowed), and let the Disk Operating

System do all the hard work of maintaining the pointers for us!

Creation of a relative file

A relative file is created by using a special form of the OPEN statement from BASIC.

OPEN lfn,dn,sa,"Ø:filename, L,"+CHR$(length)

In this statement, lfn,dn,sa and filename are the same as when OPENing a sequential file. By specifying "L" after the filename, the DOS knows that we want to create a relative file; it needs therefore to know the record length, so we send that as a character, using the CHR$ function. To create a relative file called "RELTEST", which is to consist of records of 42 characters, we could use the statement:

OPEN 7,8,9,"Ø:RELTEST,L,"+CHR$(42)

This will create the entry for our file in the directory, and the first side sector, but will not actually create any records. This is done by the PRINT# statement, just as with sequential files, the difference being that we tell the DOS which record we want to write to by sending a string to it through the command channel before the PRINT# to send the data.

Assuming that the command channel is already open and allotted to logical file number 1, if we wanted to write data to record number 1ØØ, we would first have to set the file pointer to record number 1ØØ:

PRINT#1,"P"CHR$(channel)CHR$(reclo)CHR$(rechi)
CHR$(position)

The term channel refers to the secondary address (and hence channel) associated with the relative file – in the example above this would be 9. Reclo and rechi are the actual record number, two bytes being needed because the file can hold up to 72Ø records. These parameters can be generated by the formulae:

RECLO=RECORD NUMBER—(256*(INT(RECORD NUMBER/256)))

and

REHI=INT(RECORD NUMBER/256)

Compare these with the formulae given in the section on the Memory commands, to derive the two bytes of an address in – they are the same!

The final parameter in the expression can be left out if we want to start at the beginning of the record, but if included it tells the DOS where WITHIN THE RECORD to access. This is a very powerful feature, as it means that we can read or write to any part of our relative file.

Once the pointer is positioned in this manner we can treat the relative file in the same way as a sequential file, PRINT#ing, INPUT#ing or GET#ing data as we wish. Note that the relative file does not need to know whether it is open to read or write; this makes the system even more powerful as we do not have to CLOSE the file in order to change the mode, as we did with sequential files, although we must still remember to close it when we have finished with it. If the file has already been created, the "L" and record length parameters are not needed in the OPEN statement, as these details are stored in the files directory entry, so the DOS automatically knows that it is dealing with a relative file.

Locating a Record

The relative file structure allows us to go straight to a specific record, provided we know its number. By careful design of the program to deal with the numbering of records, we can search through a relative file for a certain record. This is done by keeping records in alphabetic order: suppose we want to find the record for "SMITH K." in an address file. Well, we could start at the beginning of the file and check each record until we got the right one. This would obviously take a long time, especially if we have a large file. A better way would be to keep the records in the file in alphabetic order. Then, to

find the record we would choose one from the middle of the file and read that. The chances of getting the one we want first time are small, but if we compare the name with the we want we can exclude the half of the file which does not have the one we want. Choosing a record halfway through the remaining portion of the file, we can do the same again to narrow the search down, and by repeating this we will find the particular record we want very rapidly. In fact, doubling the total number of records in the whole file will only mean that we will have to check one more record, so this powerful idea is known as a BINARY CHOP SEARCH, because each time we read and compare a record we can "chop" half the file out of our search.

Record Structure

As a relative file has records of a fixed length, before you create it you should decide exactly what size each record is going to be, and what data you wish to store in the separate fields within each record. The simplest way to do this is to use a chart like the one below which shows details of the file structure. This type of chart is also invaluable when writing programs which use any files, as you can keep track of what data will be read in, thus avoiding the dreaded ?BAD DATA ERROR.

BOOK LIST RECORD STRUCTURE

Field Name/meaning	Length	Type
Title	20	alpha
Author	20	alpha
ISBN Code	10	alpha
Price (Gross)	5	numeric
Price (Nett)	5	numeric
Current Stock Level	5	numeric
Re-Order Level	2	numeric

Length of each record 67 characters

This example would be suitable as the record structure of a simple stock recording program for a small bookshop: while designing the program to deal with it, the programmer should think of every item of data that is liable to be required, and set the record size accordingly. Care should be taken to include delimiting characters in each field, as these are useful when using INPUT# to read the data. Carriage return characters are sent by the computer at the end of each PRINT# statement, unless they are suppressed by finishing the PRINT# statement with a semi-colon (;). You may also find it useful to keep a note of the variable names which your program uses for each item. This assists in correcting bugs which may crop up.

If we were to write data to a record which does not exist in a relative file, the DOS would create it, and set up the required pointers to it in the side sectors. As a by-product of this action, we can force the DOS to create several records in one go, as it also creates all the records below the one we specify if they do not already exist.

If we know roughly how many records we will want in a relative file it is a good idea to force the drive to create each record when we first open the file. For example, if we want to keep 100 records after opening the file we could force the DOS to create space for 100 records (assuming there is enough room on the disk!), by positioning the pointer to the 100th record, and writing data to it. This would force the operating system to create all intermediate records: it may well take a few minutes, but that time will be saved while using the file, as the DOS will not have to stop and create them when they are accessed.

The Commodore manual has two examples of relative file programs on page 36.

CHAPTER EIGHT

Changing the Drive Device Number

If you are using two or more drives, it is often useful to be able to alter the numbers by which you refer to them. This is necessary because if they are both device 8, they will both attempt to communicate with the computer at the same time; the usual result of this conflict would be to cause a crash!

There are two distinct methods of changing the device number of a drive. The first is temporary, and only lasts as long as the drive is turned on: the second method is permanent, and should only be employed if you are going to be using two drives often.

Software Method

This method involves overwriting a part of the DOS memory which is used to store the device number. This is done by using the "M-W:" command. This short program allows you to alter the device number of your drive. Remember that if you have both drives turned on when you run this, both will be given the new number, so be sure that only one is turned on at the time! After running the program, turn the second one on: this will still be device number 8, but the other will have the number you allotted it while running the program. It is worth noting that disk drives can only have device numbers between 8 and 11 inclusive, this is due to the actual way in

which the system operates, but still allows you to use four drives on each computer, which is enough for most people.

```
10 OPEN1,8,15
20 INPUT "NEW DEVICE NUMBER (8-11)";dn
30 IF dn<8 OR dn>11 THEN 20
40 PRINT#1,"M-W:"CHR$(119)CHR$(0)CHR$(2)
CHR$(dn+32)CHR$(dn+64);
50 PRINT "DEVICE NUMBER IS NOW ";dn
60 CLOSE1:END
```

Hardware Method

The hardware method is explained in fair detail in the User manual (page 40), and basically involves cutting specific pieces of wire inside your drive. Before you do this, We would like to point out (they don't in the manual) that the guarantee on the disk unit will be voided as soon as you open the case. For this reason, it is probably better to stick to the software method above.

APPENDIX ONE

Quick Reference Table of Disk Commands

(As sent to the DOS through the command channel.)

Command	Syntax
NEW	"N0:diskname,ID"
COPY	"C0:newname=0:f1,0:f2,0:f3..."
RENAME	"R0:newname=0:oldname"
SCRATCH	"S0:filename"
INITIALISE	"I0" or "I"
VALIDATE	"V0" or "V"
BLOCK-READ	"B-R:"ch;dr;t;s
BLOCK-WRITE	"B-W:"ch;dr;t;s
BLOCK-ALLOCATE	"B-A:"dr;t;s
BLOCK-FREE	"B-F:"dr;t;s
BUFFER-POINT	"B-P:"ch;position
USER1	"U1:"ch;dr;t;s
USER2	"U2:"ch;dr;t;s
POSITION	"P"CHR$(channel)CHR$(reclo) CHR$(rechi)CHR$(position)
BLOCK-EXECUTE	"B-E:"ch;dr;t;s

MEMORY-READ	"M-R:"CHR$(adlo)CHR$(adhi)
MEMORY-WRITE	"M-W:"CHR$(adlo)CHR$(adhi) CHR$(number of bytes)[data in CHR$]
MEMORY-EXECUTE	"M-E:"CHR$(adlo)CHR$(adhi)

User commands "Un:" with parameters where required.

APPENDIX TWO

Disk-Related Error Messages from BASIC

In this section, we will go through the various error conditions you may come across while using your disk unit. If the error is detected while running a program in BASIC, the messages will be displayed on your screen together with the line number where the error occurred. This feature is useful when writing programs, as it tells you where you must look for errors, thus assisting in the de-bugging of your programs. You should note that any error condition reported in this way will stop the execution of your program, so you must CLOSE any open files in order to ensure that your disk does not become corrupted. This is most easily done by the following series of instructions, to be entered through the keyboard:

```
CLOSE1:OPEN1,8,15:CLOSE1
```

The first CLOSE statement ensures that logical file number 1 is closed. The command channel is then opened, using logical file number 1, and immediately closed again. This also closes any open files on the disk correctly, thereby minimising the risk of losing any information.

? Bad data error

While reading a file with an INPUT# or GET# statement

(this applies to either disk or tape files), the program was expecting to receive numeric data but string data came in. This error can be avoided by keeping a note of your file structure, or, where this is impractical, data should be read into string variables, which then may be converted to numerics using the BASIC function VAL.

?Device not present error

This message is displayed on the screen whenever the computer makes an attempt to access a device on the serial bus (including the disk unit), and receives no response. There are three possible causes for this:

(a) The disk drive is not connected to the serial port, in which case you should connect it using the black lead supplied with the drive.
(b) The drive is not turned on, or not connected to the mains. If it is, the green power indicator should be lit, or
(c) you may have specified the wrong device number in a command – when you first switch on, the disk drive will always be device number 8.

?File not found error

A File Not Found error will occur if you try to open a file to read or load, and the disk drive cannot find an entry corresponding to the name given in the directory. It will also happen if you attempt to LOAD a program not on the diskette in the drive. Possible causes include:

(a) that you have mis-spelt the name of the file you want,
(b) that you have got the wrong diskette in the drive, or
(c) that you have deleted the file previously. When using the tape system, this error can be caused if a special end of tape marker is found.

?File not open error

This is caused by attempting to read or write to a logical file which is not open. It may be that you have forgotten to OPEN the file, or that the file has been closed, either by your program or by an error condition. In either case, you should close any other open files, and check your program to make sure that it does open the required file before attempting to access it.

?File open error

Once you have opened any file with a specific logical file number (the first parameter in the OPEN statement, also used in CLOSE, PRINT#, INPUT# and GET#), that logical file number is allocated in the computer to that particular file, until it is de-allocated by a CLOSE command. An attempt to use the same logical file number in another OPEN statement will result in this error condition and to avoid this, you should get into the habit of closing any file as soon as you have finished with it. If you need to have two or more files open at any given time, make sure that you specify different logical file numbers for each in their relevant OPEN statements.

?Illegal direct error

Basic uses a single 80 character buffer to process both commands being entered from the keyboard, and data being read in from an external device, such as the disk unit. As any command which reads data into the computer would overwrite the input buffer, commands such as INPUT, GET, INPUT# and GET# are only allowed to be used from within a program. If you try to enter any of these commands directly from the keyboard (this is known as DIRECT MODE), the computer's operating system gives this error message. In order to read the error channel if you do not have a program running, you should enter a program and run

it, or use one of the utility programs available, such as the DOS SUPPORT program mentioned earlier.

?Not INPUT file

This error is generated by the operating system if a file has been opened in order to write to it, and an attempt is made to read data from it. If you wish to read the data from a file, it must first be closed correctly, and then re-opened with the MODE parameter specified as READ. For an example of this, see the section on reading and writing sequential files.

?Not OUTPUT file

This error condition is the exact opposite of the previous one, in that it occurs if we try to write data out to a file which was specified as an input file in its relevant OPEN statement. The cure is the same as for the Not Input File error, namely to close the file and re-open it with the correct mode parameter (Write).

?VERIFY error

When you SAVE a program (either to disk or tape), you should always issue a VERIFY command to check that the program has been saved correctly. The VERIFY command actually reads the disk (or tape) and compares it with the program in memory at the time. If any differences are detected, the operating system will tell you by giving this message. If this occurs, you should delete the file, using the scratch command to the disk drive and try to save and verify it again. Note that certain files, such as screen dumps will not verify correctly, as the memory contents will be altered by messages being printed, etc.

A successful verification gives the message "OK" when complete.

Disk-Generated error conditions

Certain types of errors sometimes occur when using disk

drives; these are detected by the disk unit, but not by the computer controlling it. With these errors, the drive tells the operator that something has gone wrong by repeatedly flashing the red drive light on and off. The drive will also set up a string of data which can be read into the computer through the command channel (secondary address number 15): a program can then warn the user, or automatically take corrective action where applicable.

The error channel can be read at any time for such information, and should be checked after each operation which accesses the drive. In the chapter on Sequential Files, you will find an example of a suitable error channel checking subroutine, which should be called immediately after each PRINT#, GET#, INPUT#, OPEN or CLOSE statement, in order to avoid any problems.

This routine reads in four parameters from the error channel: ER, ER$, ET and ES. ER stands for error number, and corresponds to any error condition. The possible values of ER are listed below together with a short description of what exactly they mean. ER$ will contain a (very) short description of the actual error detected, this is useful to display on the screen in order to let the user of the system know what has happened. A lot of the possible errors are related to the specific block which the drive was trying to access, so the track and sector numbers (ET and ES), are passed back to the computer as the last two parameters read in from the error channel (where they apply). These are also useful to know when using random access files, as they contain the track and sector of the next available block on the disk after using the block-allocate instruction.

00, OK

If reading the error channel gives this result, you are, as it says, OK! This is the result that we always want, because it means that nothing has gone wrong. Because of this, it may be safely ignored, and the program allowed to continue.

01, FILES SCRATCHED, XX

This is not really an error message, but allows the disk drive to tell the computer how many files have been deleted from the disk after issuing a command to the drive to SCRATCH files. This is especially useful when deleting files by using the pattern-matching symbols, "*" and "?". The actual number of files scratched is returned as the next parameter (XX): if this is more than the number you expected, the chances are that you have deleted something that you did not mean to!

Error numbers between 2 and 19 are not used, and should be ignored if they occur, since they have no meanings.

20, READ ERROR,TT,SS (Block Header not found)

Reading this in through the error channel means that the disk operating system has been unable to find the block-header that it was looking for. Part of the formatting process when NEWing a diskette includes the creation of a header at the start of each block, which is later used by the drive to "find its way around". If these block headers became corrupted in any way the drive will get very upset, and give the error!

21, READ ERROR (No sync character)

This condition is very similar to error number 20, but instead of not being able to find a block-header, the drive cannot find a mark (known as the sync mark), on the track it is trying to read or write to. The sync mark, or synchronisation mark as it should be called, is written into each block during the formatting process, and is used by the drive to warn it that a block of data is arriving underneath the read/write head. This error can be caused by the drive being badly set up (do not try to adjust it yourself!) or there being an unformatted (or no) diskette in the drive. If you are sure that neither of these is the cause, I'm afraid that the disk has probably become totally useless, as you will not be able to read it!

22, READ ERROR (Data block not present)

This error indicates that your disk drive has tried to read a block which has not been properly created. This is the sort of thing that is liable to happen to you if you forget to CLOSE files after you have finished with them. The most sensible thing to do is to delete the file, and re-create it where possible, as data read from it will not be reliable. This shows the need to keep security copies of any important data files and programs on separate diskettes, as they can be used to recover from such errors in certain circumstances.

23, READ ERROR (Checksum error in data block)

In each block of data, the disk drive stores a checksum, which is checked each time that block is read. This is calculated from the data that is being written into the block in question, and is used to check that the data has been read from the block correctly. If any discrepancies occur, the drive knows that the data it has read is incorrect, and gives this error condition.

24, READ ERROR (Byte decoding error)

This error means that data has been read into the disk drive's memory (not the computer's memory), but it is incorrect. Once data is read from the diskette, it goes into an area of memory inside the drive unit where various complex error checks are performed on it. If any of these error checking procedures fail, this message is set up, to be read through the command channel into the computer.

25, WRITE ERROR

After writing any data to the diskette, the drive unit automatically re-reads it in order to check that it has been correctly written. Should the data read back into the drive's memory not correspond to that which it wrote out, this error is generated. The idea behind this is much the same as the

reason for VERIFYing a program you have SAVED, but it is carried out by the drive on every block that is written to the disk.

26, WRITE PROTECT ON

This condition indicates that the disk drive has been asked to write information to a diskette, but that the disk in the drive has got a label stuck across the write-protect slot. No data can be written to a disk with a write-protect tab on, although the disk can be read freely. This is a very effective method of protecting a diskette from being accidentally overwritten, and you will find that most commercial programs have labels stuck over the write-protect slot for just this reason.

27, READ ERROR (Checksum error in header)

This is similar to error number 23, in that a checksum byte in a block does not match the actual contents of that block. The difference in this case is that the checksum is performed on the 6 control characters at the start of the block. These are the SYNC character: an 8 (presumably the device number that the disk was formatted on, but nobody seems to know!), two characters for the ID of the disk (used to make sure that nobody has changed the disk), and the track and sector numbers for that block.

28, WRITE ERROR (Long data block)

After writing a block of data onto the diskette, the disk operating system always tries to read the SYNC mark of the next block. If this is not found, the drive will generate this error, which can be taken as meaning "I'm terribly sorry, but I've just wiped out half the next block!". It is a little bit late by this time, as the damage is already done, but it is nice of the machine to tell you!

29, DISK ID MISMATCH

When a diskette is inserted into the drive the Block Allocation

Map (BAM) should be read into the internal memory of the drive unit. This is normally achieved using the initialise command, but is also done when a disk is formatted by the NEW command. In addition to the BAM being read, the disk operating system (DOS) reads other information relating to the diskette, amongst which is the identifier (ID). In all future read or write operations, the DOS checks the ID bytes in each block-header before continuing with its job. The reason for this is that at any time someone could pull the current diskette from the drive and replace it with another. Obviously, this would lead to corruption of the second diskette, as the drive could unwittingly overwrite files on the wrong disk! By checking the ID of each block prior to using the data, then, the ID is checked. If the diskette has been changed over, these will not match (unless they both had the same IDs), and the operation would be aborted. This is a safety measure to prevent corruption of data in this manner, but can also occur if the diskette has been corrupted in such a way that the ID of any given block has been changed from that of the rest of the disk. This, however, is extremely unlikely to happen.

30, SYNTAX ERROR (General Syntax)

The Disk Operating System is really a machine code program running inside the disk unit, in the same way as the BASIC language is only a program which runs inside your computer, and in the same way as BASIC, it has been programmed to understand a limited series of commands. If you give the BASIC interpreter program something that it does not recognise, it will tell you by giving you an error message, SYNTAX ERROR. Similarly, sending commands to the disk drive which are improperly constructed will cause this error. An example of this would be if you forgot to specify the disk name in the NEW disk instruction.

31, SYNTAX ERROR (Invalid Command)

This error condition is generated by the DOS if it does not

recognise the command it has received. Error number 30 is generated if a command is recognised, but the parameters following it are invalid. This condition occurs if the command is not understood at all! It is worth noting that the actual command required must start at the first character of the command string passed to the DOS – any leading characters will generate this error condition.

32, SYNTAX ERROR (Long line)

BASIC uses an area of memory called an INPUT BUFFER to process commands being entered at the keyboard. In the same way, the Disk Operating System uses an input buffer in its internal memory to process incoming commands. The size of this buffer gives us an upper limit of 58 characters on the length of each command string we can send to the disk drive. Exceed this, and the DOS will decide that it cannot cope, and will give this error. The only time that this is liable to occur is when you use the COPY command to concatenate several files with long names. For this reason, if a series of files are to be joined in this way, it is wise to keep the length of each one's name to a minimum.

33, SYNTAX ERROR (Invalid Filename)

Pattern-matching characters (* and ?) are allowed in filenames when opening a file to read, as the DOS selects the first file on the directory which matches. However, these characters would cause difficulties if they were allowed to be used as part of a real filename, so they are disallowed. Attempting to create a file with these characters embedded in the name will therefore generate this error.

34, SYNTAX ERROR (No Filename)

If reading the error channel returns this message, it implies that the file name has been left out of a command to the DOS. The most likely reason for this is that the colon has been

omitted from the front of the filename, so the DOS does not recognise it.

39, SYNTAX ERROR (Invalid Command)

This condition is much the same as error number 31. In this instance though the command not recognised has been sent to the command channel, which has a secondary address of 15.

50, RECORD NOT PRESENT

This error, as its name suggests is generated by the disk operating system if it has been instructed to read a record which is not there. This can happen if you try to read past the end of a file, using INPUT# or GET#, in which case any data read in will be meaningless.

51, OVERFLOW IN RECORD

The relative file format requires that the record length is set when the file is first created. In all future operations on that particular file, the DOS refers to the record size, stored on the diskette in order to find the record required. If you attempt to write more data to any one record than the record length the DOS will detect the fact and give this condition. The extra data that has been sent is lost, as the DOS has nowhere to put it. Remember that a carriage-return is sent at the end of a PRINT# statement, so allow for this when calculating your record size.

52, FILE TOO LARGE

As you write successive records to a relative file, the DOS creates more as required. Attempting to create more records than there is space for on the diskette will cause this error. It is similar to error number 72, which indicates a full diskette.

60, WRITE FILE OPEN

Having created a file, it must be closed before you attempt to read it. If you do not, the DOS will give this message on opening that file to read. To avoid this, make sure that you close any write files before opening them again to read.

61, FILE NOT OPEN

If you attempt to access a file on the disk which has not been opened (or has been closed because of some other error), the drive will tell you with this message. A similar message is generated by BASIC, but it is possible for BASIC to think that a file is open while it is not really. For example if you turn the drive off and on again, any open files will be closed (and probably corrupted, too!), but BASIC will still consider them to be open! In this case the BASIC error message would not be generated as this DOS error appears to have been inserted as an afterthought when Commodore realised this possibility.

62, FILE NOT FOUND

Attempting to LOAD a program or open a file to read from which does not exist on the diskette in the drive will cause this error message. Should it happen, check that you have not misspelt the filename, and that you have got the correct diskette in your drive.

63, FILE EXISTS

This indicates that you are trying to create a file on the diskette when there is already something on that disk of the same name. If it happens, there are three possible courses of action. Firstly, you could delete the offending file, using the SCRATCH command to the drive and then try again. Another solution would be to change the name of the file you are attempting to write (or the name of the existing one), or finally, you could decide to create your new file on a different diskette.

64, FILETYPE MISMATCH

When opening a file to read, the filetype parameter in the open statement (LOAD assumes that you mean a program) must match with the filetype of the file on the diskette which you wish to access. This is to prevent the accidental reading of data which would then come in an unexpected format. If the wrong filetype is specified, then the DOS will report the fact by means of this message.

65, NO BLOCK, TT, SS

This is the resulting message from attempting to allocate a block which is already "spoken for" according to the BAM, using the B-A command. The track and sector values returned will not be those of the actual block you tried to allocate, but the address of the next free block. This makes this particular error very useful in random access applications, when you want to know where the next free block is, so you can go and store data there. If both the track and sector values read in this way are zero, this indicates that there are no more free blocks further on in the disk which are free. If this is the case, you should search for a block with lower track and sector numbers than the current one.

66, ILLEGAL TRACK AND SECTOR

Receiving this code through the error channel indicates that the disk operating system has tried to access a track or sector which is outside the range possible for a correctly formatted diskette. This could indicate that the pointers used to chain consecutive blocks of a file together have become corrupted. Using direct access methods it **may** be possible to correct these, but normally it is simpler and quicker to re-create the file in question from a backup copy.

67, ILLEGAL SYSTEM T OR S

This is a special error message generated by the DOS which

indicates that it is trying to access data from non-existent blocks on the diskette. This would suggest that control data stored on the diskette has become corrupted. Unfortunately, Commodore do not appear to have published very much about this condition. I would suggest therefore, that if you encounter this error (which is very rare), you should immediately make a copy of the disk that it occurred with, just to be on the safe side!

70, NO CHANNEL

If you attempt to open more than five sequential files at any one time, or the channel you requested in a direct access application has already been allocated by the DOS, this error condition is generated. It would be most unusual that you should find it necessary to have more than five files open at any one time, but if you do you will have to close some of them down before opening others. It could indicate that you are not closing files correctly after using them, which is not good programming practice!

71, DIR ERROR (Directory error)

This (thankfully!) unusual condition happens if the Block Allocation Map in the memory of the disk drive does not match up with the Block Allocation Map on the physical diskette. It means that possibly you have overwritten part of the BAM in the DOS memory (have you used M-W?), or could indicate a hardware failure. Should this occur, a possible solution is to re-initialise the disk (using I0). This would force the disk drive to re-read the BAM from the disk, but any active files will be closed, thus possibly causing problems when you come to read a file which was being written to at the time.

72, DISK FULL

As the name implies, receipt of this message means that the disk you are using is full. There are two ways in which a disk can be full, either of which causes this message to be sent.

Firstly, the BAM could be indicating that there are no more blocks free, so no more data can be stored (in which case the message is sent when there are two blocks left, so that the current file can be correctly closed), or it could be that there is no more room in the directory for details of more files to be stored: the maximum number allowed being 144 entries. Validating the disk may free a few blocks up, but generally the best solution would be either to delete a few files, or failing that start using another diskette.

73, DOS MISMATCH

It is possible for you to read disks created on certain other Commodore disk drives, apart from the 1541, but not be able to write data onto them. An example would be a diskette originally formatted on the old 3040 dual disk drives used with the Commodore PET machines. Disks formatted on other Commodore drives, such as the 8050 will be totally unreadable, as they have a different number of tracks and sectors compared with the 1541. It is possible, using various direct access techniques to change the format of a diskette. This idea is sometimes used to prevent commercial programs from being copied illegally.

74, DRIVE NOT READY

This is a fairly common error message returned by the DOS, and indicates that you have tried to read or write to a diskette when there is none in the drive. This will also occur if you have not inserted the diskette correctly, or forgotten to close the flap, or are trying to access a disk which has not been formatted correctly.

APPENDIX THREE

Programs

In this section, we will give you a couple of useful programs for dealing with disk files. The first of these is very simple, and allows you to display the contents of a sequential file on the screen. Program files could also be examined in this way, but they are stored in the form they are in your computer's memory with all their keywords tokenized, so it would be difficult to interpret the results. You could, however, modify the program to detect tokens and print them out as they would be listed, but I'll leave that one to you!

```
100 open1,8,15,"ui+":rem change the "+" to "-" if
you are using a vic 20!
110 print"[CLS][CD][CD]   file-lister[CD]"
120 print"please insert correct disk in drive."
130 gosub 300:rem *** wait for keypress
140 print#1,"i0":rem *** initialise disk
150 input"filename to
examine";f$:open8,8,8,f$+",s,r"
160 gosub 330:rem *** check error channel
170 get#8,a$:x=st:rem *** get a character
180 gosub330
190 printa$;:getx$:ifx$<>""then gosub 300
200 ifx=0 then 170
210 print:print "end of file":close8:close1:end
300 rem *** pause feature
```

```
310 print:print "    press any key to continue"
320 wait 198,15:getx$:return
330 rem *** check error channel
340 input#1,er,er$,et,es
350 if er=0 then return
360 print "error number";er;" means ";er$
370 print "track ";et,"sector ";es
380 print "press 'c' to continue or 'a' to abort"
390 gosub320:if x$="a" then close8:close1:end
400 if x$="c" then return
410 goto 390
```

The next program we have for you is a very useful utility which allows you to recover disk files which have been deleted using the scratch command. Provided that you have not written any data to the disk since the file was deleted, this should work, but if you have, well I'm afraid that you have probably lost it for good! After running this program, it is a good idea to use the file lister on it, as the program has no way of telling if the data recovered in this way is correct, only you can tell that! The program works by looking through the directory for the required file. If it finds it, the program will correct the directory entry and then trace it through block by block allocating each as it goes, thus making sure that the file cannot be overwritten in the future. If the pointers in any block are pointing somewhere that is impossible, the file is considered to be too corrupt to recover, and the program will stop after scratching the file again.

```
100 open1,8,15,"ui+":rem change the "+" to "-" if
you are using a vic 20!
110 print"[CLS][CD]          [R/ON] file recovery
program.  [R/OFF][CD]"
120 dimft$(4): ft$(0)="del": ft$(1)="seq":
ft$(2)="prg": ft$(3)="usr": ft$(4)="rel"
130 ef$=chr$(0):forn=1to4:ef$=ef$+ef$:next
140 print"[CD][WHITE] this program is designed to
recover"
150 print"accidentally deleted programs from a"
```

```
160 print"diskette.if you have issued a 'scratch'"
170 print"command,possibly using pattern-matching"
180 print"characters, and discovered that your"
190 print"favourate program has gone, [COM/RED]do
not try"
200 print"to write anything else to that
diskette!"
210 print"[WHITE]run this program,and supply the
name &"
220 print"filetype when requested. the program"
230 print"will then do it's best to recover the"
240 print"lost file, but we cannot guarantee 100%"
250 print"success. any filetype may be recovered"
260 print"using this utility. [R/ON] good luck!!!
"
270 printspc(6)"[5 CD's][R/ON] press any key to
continue [R/OFF][YELLOW]"
990 gosub61000
1000 rem ** start of actual program. **
1010 print"[CLS][CD] please insert disk to operate
on, then"
1020 printspc(12)"[CD]press any key"
1030 gosub61000
1040 print#1,"i":gosub60000
1050 print"[CD]    please enter filename to
recover ":inputf$
1060 if f$=""or len(f$)>16 then print "illegal
filename!!!": goto1050
1070 open2,8,2,"#":t=18:s=1
1080 gosub10000: print#1,"b-p:"2;0: gosub60000
1090 gosub10300:nt=asc(a$): gosub10300: ns=asc(a$)
1100 f=0:rem directory entry number
1110 gosub10400
1120 ifleft$(n$,len(f$))=f$thenprint"[R/ON] found
file [R/OFF]":goto2000
1130 f=f+1:iff>7thent=nt:s=ns:goto3000
1140 ifn$=ef$thenprint"[CD][CD]   [R/ON] file not
found on this diskette! [R/OFF]":close2:close1:end
1150 goto1110
2000 rem correct file found - now try to fix it!
2010 input"filetype (p,s,u,r)";ty$
2020 ifty$<>"p" and ty$<>"s" and ty$<>"u" and
ty$<>"r" then 2010
2030 ifty$="s"thenaa=129
```

```
2031 ifty$="p"thenaa=130
2032 ifty$="u"thenaa=131
2033 ifty$="r"thenaa=132
2040 print#1,"b-p:"2;(2+(32*f)): gosub60000
:print#2,chr$(aa);: gosub60000
2045 gosub10100
2050 forb=2tonb:t=dt:s=ds:gosub10000
2060 print#1,"b-p:"2;0: gosub10300: nt=asc(a$):
gosub10300: ns=asc(a$)
2070 print#1,"b-a:"0;t;s:gosub60000
2080 ifnt>35 or ns>20or nt=0then print "[R/ON]
file too corrupt!
[R/OFF]":print#1,"s0:"n$:close1:end
2090 print"track "t,"sector "s
2100 dt=nt:ds=ns:next
2110 print"file [R/ON]may[R/OFF] be
ok...":close2:close1:end
3000 ift<35ands<21then1080
3010 n$=ef$:goto1140
10000 rem *** read block t/s
10010 print#1,"u1:"2;0;t;s:gosub60000
10020 return
10100 rem *** write block t/s
10110 print#1,"u2:"2;0;t;s:gosub60000
10120 return
10300 rem get# a character
10310 get#2,a$:ifa$=""thena$=chr$(0)
10320 gosub60000:return
10400 rem *** read directory entry
10410 print#1,"b-p:"2;(2+(32*f))
10420 gosub10300:cl$="
closed":a=asc(a$):ifa<128thencl$="unclosed"
10430 ifa>127thena=a-128
10440 ifa>4thenft$="???":goto10460
10450 ft$=ft$(a)
10460 gosub10300:dt=asc(a$):gosub10300:ds=asc(a$)
10470 n$="": forn=3to18: gosub10300: n$=n$+a$:
next
10480 forn=19to27:gosub10300:next
10490 gosub10300: nb=asc(a$): gosub10300:
nb=nb+(256*asc(a$)): if n$=ef$ then return
```

```
10500 print" "chr$(34)n$chr$(34)"   type "cl$"
"ft$:print" size="nb"blocks."
10510 return
60000 input#1,er,er$,et,es: if er=65 or er<20 then
return
60010 print"[CD]error number "er" - means "er$
60020 print"detected on track "et" , sector "es
60030 print"[CD]        [R/ON]a[R/OFF]bort program,
or [R/ON]c[R/OFF]ontinue?"
60040 gosub61000:ifx$="c"thenreturn
60050 ifx$="a"thenclose1:print"[R/ON] program
aborted. [R/OFF]":end
60060 goto60040
61000 rem ** wait for keypress **
61010 wait 198,15:getx$:return
```

Have you ever wanted to protect your diskette from being written on? You could always stick a label over the write/protect slot, but they tend to fall off after a while. Another way is to alter the diskette so that when the DOS checks if the diskette is in the correct format to write onto, it decides that it cannot. This is achieved by altering the character which is used as an indicator of the format. This byte is the third one in the BAM block (Track 18, sector Ø) and normally holds an "A" [CHR$(65)]. If this is altered, the disk becomes write-protected, so much so that although the disk is readable you cannot even change the appropriate byte back again! Another useful side-effect of this is that if someone tries to copy your disk, using a dual drive, or block-by-block, as soon as the BAM block is written, they copy the write-protection across. This means that they will find it difficult to copy the whole of your disk!

```
10 rem *** disk write protect program.
20 open 1,8,15,"ui+"
30 print"[CLS][2 CD's] please insert the disk to
protect."
40 print "[cd] and then press any key."
50 wait198,15:geta$
60 print#1,"i0":rem initialise the disk.
```

```
70 open2,8,2,"#":rem open channel number 2 for
direct-access
80 print#1,"u1:"2;0;18;0:rem read BAM block
90 print#1,"b-p:"2;2:rem position pointer to 3rd
byte (they start at 0!)
100 print#2,chr$(66);:rem could be anything other
than 65!
110 print#1,"u2:"2;0;18;0:rem write block back to
disk
120 print#1,"i0":re-initialise the diskette
130 close1:print "finished!"
```

The final program in this section is designed to allow you to change the name and/or ID of a diskette. Normally, this can only be done by NEWing the disk, but this wipes everything off it. The main use of this is so that you can alter the ID's of your diskettes so that each one is unique. This means that should anyone change the diskette in your drive at the wrong time, the DOS has a chance to detect the fact before overwriting your precious data!

```
10 rem  ********************************
20 rem  this program is designed to run
30 rem  on a commodore 64 - the disk
40 rem  commands are the same for a vic
50 rem  except line 200, where the "+"
60 rem  should be altered to "-". this
70 rem  ensures that the drive runs at
80 rem  the correct speed.  the screen
90 rem  layout should also be corrected
100 rem      if you have a vic 20.
110 rem ********************************
160 tr=18:ns=144:is=162
170 print?[CLS][3 CD's] put disk requiring
renaming in drive.[CD]"
180 print"   please press 'return' to
continue":print
190 geta$:ifa$<>chr$(13)then190
200 open15,8,15,"ui+":print#15,"i":open2,8,2,"#"
210 print#15,"u1:"2;0;tr;0
220 d$="name":b=ns:r=16:gosub500
```

```
230 d$="id":b=is:r=2:gosub500
240 ifv$=""thenv$="nothing"
250 print#15,"u2:"2;0;tr;0:print#15,"i"
260 close2:print"* ";v$;" changed *"
270 close15
280 open15,8,15:print#15,"i":close15
290 end
500 print#15,"b-p:"2;b
510 print"old ";d$;" is ";chr$(34);
520 forj=1tor: get#2,a$: printa$;: next:
printchr$(34): print
530 print"please enter new ";d$:print"press
'return' for no changes"
540 input"[2 C/RIGHT's]*[3
c/LEFT's]";x$:print:iflen(x$)>rthenprint"[R/ON]inv
alid length![R/OFF]":print:goto540
550 ifx$="*"thenreturn
560 iflen(x$)<rthenx$=x$+chr$(160):goto560
570 v$=v$+w$+d$:w$=" and "
580 print#15,"b-p:"2;b:print#2,x$;
590 return
```

The Commodore 64 disk drive has an extremely complex manual. This simple book is designed for the first-time disk user. It explains all the commands available and the best way to use a disk drive. It is easy to read and virtually essential for the new disk drive owner.

ISBN 0 7126 0543 6
CENTURY COMMUNICATIONS LTD

This was brought to you

from the archives of

http://retro-commodore.eu