

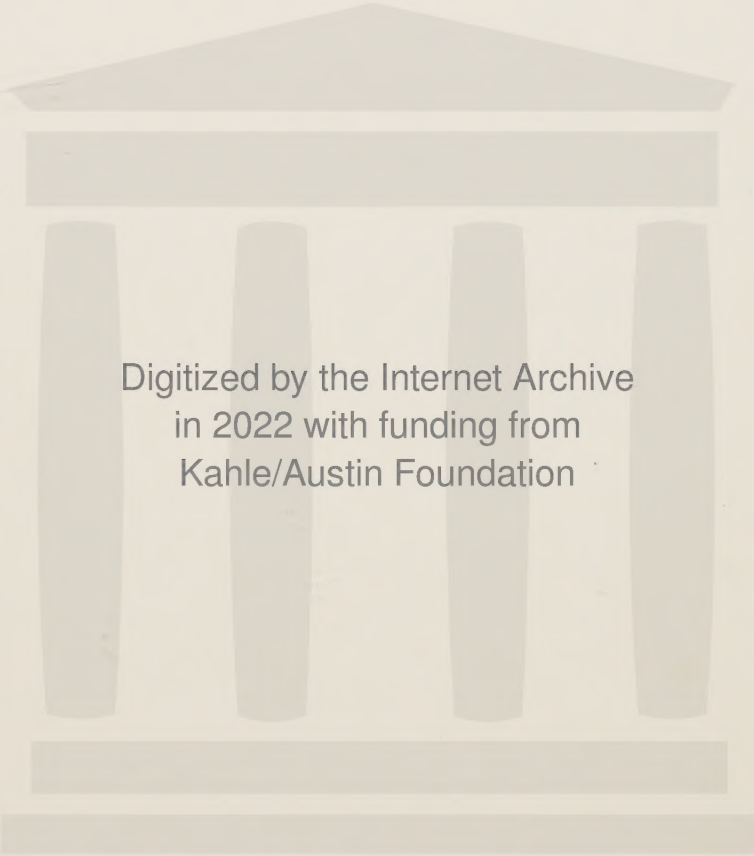
BBC MICRO DISK DRIVES

R.D. Bagnall



MICRO PRESS

BBC Micro Disk Drives



Digitized by the Internet Archive
in 2022 with funding from
Kahle/Austin Foundation

BBC Micro Disk Drives

R. D. Bagnall



MICRO PRESS

First published in 1985 in the United Kingdom by
Micro Press
Castle House, 27 London Road
Tunbridge Wells, Kent

© R. D. Bagnall

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

British Library Cataloguing in Publication Data

Bagnall, R.D.

BBC Micro disk drives.

1. BBC Microcomputer 2. Data disk drives

I. Title

001.64'42 QA76.8.B35

ISBN 0-7447-0028-0

Typeset by MC Typeset, Chatham, Kent
Printed and bound by Mackays of Chatham Ltd.

Contents

Foreword vii

Chapter 1 Basic Information About Disk Drives 1

Chapter 2 Disk Drives and the BBC Micro 9

Chapter 3 Using Your Disk Drive 14

Chapter 4 Random Access Files 45

Chapter 5 Problems 55

Chapter 6 Further Work with Disk Drives 58

Reference Section 64

Index 103

Foreword

Microcomputers are basically simple machines, and books about them should be simple too. Unfortunately, the world of microcomputers suffers from a lot of unnecessary jargon, and all too often an author starts well, only to slip into the jargon without proper explanation, leaving the novice reader baffled and disappointed. Many authors pad out their books with 'helpful' demonstration programs which sometimes occupy several pages, take a long time to type into the computer, and may have an annoying misprint which stops the program running properly.

What I have needed, and what I still need today, are simple concise books which provide the basic information without frills or jargon, aren't stiff with unnecessary programs, and which provide a reference guide at a glance. Unable to find the books that I need I have decided to write them myself, and I hope that others will find them equally useful.

This is the first such book. It concerns the use of disk drives with the BBC Micro, and explains all of the things that you need to know in plain simple English. I hope you like it.

June 1984

R.D. BAGNALL

Basic Information About Disk Drives

Microcomputers have only a limited memory, and this is normally erased when they are switched off. If precious programs are not to be lost, some way must be found to keep a separate permanent record of them. The simplest way has been to transfer them to cassette tape using an ordinary tape recorder. When a program is required again later, the tape is simply played while the computer 'listens' through the earpiece socket of the tape recorder. This system should be familiar to microcomputer users.

Unfortunately, cassette tapes have three drawbacks. Firstly, they can stretch in use so that programs become distorted and do not 'sound' right to the computer, just as music would not sound right to you. Secondly, they are painfully slow at transferring programs, and with longer ones may require several minutes to do so. Thirdly, they do not allow immediate access to all parts of the tape, and it can be very frustrating to have to search through a tape looking for a particular program.

Disk drives overcome all of these problems. Essentially, they are just sophisticated tape recorders designed to record and replay at a much faster rate than normal, but instead of a tape there is a disk of the same material which is played rather like a gramophone record. To achieve this the recording head is no longer fixed but can scan rapidly across the disk surface, and since all parts of the disk are immediately accessible any particular program can be quickly located.

Like cassette recorders, disk drives can record not only programs but also lists of information such as tables of numbers or names and addresses, known in computer jargon as data files and text files. Immediate access to all parts of a disk means that such files can be searched repeatedly for

information, whereas similar files on tape can only be searched once before the tape must be rewound.

Since in computer jargon a program is a sort of file, disk drives are in general called disk filing systems, or DFS for short.

Tracks and sectors

There is an important difference from playing a record however. Records have a spiral groove, and if disks had a similar spiral magnetic track the recording head would still have to go through each file in turn to find the one required, albeit at a much faster rate than a tape recorder. Instead, disks have many circular tracks like a rifle target or bull's-eye, and when a file is transferred to a disk two things happen. Firstly, the file itself is stored in the first available space on the disk, and secondly the name of the file and its location on the disk are stored separately in a special place on the disk which the computer reserves for filenames. When a file is required later, the record/read head first goes to this special part of the disk, confirms that the file exists, and notes its location. The head then swings to the correct track and begins to transfer the file to the computer.

All of this requires synchronisation. For a start, a disk must have tracks on it in all the right places, but unlike the groove on a record there are no tracks to be seen on a new blank disk. In fact, you must lay down the tracks yourself magnetically, and the first task with a blank disk is to get the disk drive itself to do this for you. The process is known in computer jargon as *formatting*.

Even with circular tracks there is still a problem. Where, for example, does a file start and end on each track, particularly if a long file occupies several tracks? Without this information, the recording head would not know where to begin on each track and files could well be transferred back to the computer, jumbled up so that they made no sense. For this reason, tracks are actually laid down as sectors, with a small magnetic gap between each sector.

Even so, when the recording head swings to the correct track, how does it know which sector is which? The answer is

simple. Each disk has one or more holes punched in it at precise locations, and a corresponding photocell is built into the disk drive. With each revolution of the disk the photocell detects pulses of light, and these are used as markers when tracks and sectors are being laid down during formatting. Later, when the record/read head is hunting for a particular sector, the same markers again provide synchronisation. Such holes are sometimes called index holes.

Types of disk drive

Disk drives are available as either 40-track or 80-track models, meaning that they lay down either 40 tracks or 80 tracks over the same area of disk surface, with a corresponding change in track width. They may be single sided with only one recording head, or double sided with a recording head on each side of the disk, both heads normally being driven by the same motor. They may also be obtained as dual models with two separate disk drives in the same housing, either side by side or stacked one above the other, and again each drive can be single or double sided. It is even possible to obtain a switchable 40-track/80-track drive.

Types of disk

A typical disk is $5\frac{1}{4}$ in. in diameter, although other sizes are now becoming popular. Because it is thin and floppy it is known in computer jargon as a floppy disk, but there is nothing special about its floppiness other than low cost, and expensive professional computers often use rigid disks. To protect a floppy disk, it is permanently sealed into a stiff protective sleeve, with suitable cutouts for the recording head and index hole. In one edge of the sleeve there is also a small cutout which acts in a similar way to the protective tabs on the back of a cassette tape. Uncovered, it allows all disk operations to take place, but covered with an adhesive strip it only allows the disk to be read, thus protecting it against accidental erasure.

Some disk drives require only one index hole in a disk, in

which case all the gaps between sectors are magnetic. Such disks are said to be 'soft sectored'. Other systems require a ring of index holes, one per set of sectors, in which case the disk is said to be 'hard sectored'.

Some microcomputers provide the choice of doubling the number of sectors in each track while still keeping the same amount of information in each sector, thus doubling the storage capacity of the disk. The computer is then said to be operating in *double density* as opposed to the normal *single density*, and the ability of a disk to keep all of the little bits of information separate in double density depends on the quality of the disk coating. Disks are therefore usually certified by the manufacturer as suitable for single or double density. In a similar way, the ability of a disk to keep 40 or 80 tracks separate also depends on the disk coating, so that disks are usually certified for this as well.

Finally, disks are available as either single or double sided to suit the various disk drive options.

Thus a typical floppy disk might be certified as single sided, double density, soft sectored and 40 tracks.

Setting up

A double-sided dual drive is then really four separate drives, and can be used with microcomputers which have been designed to work with up to four drives at a time. To distinguish them, the computer usually numbers them 0, 1, 2 and 3, and you can normally choose which is to be which by setting certain switches inside the drives in accordance with the manufacturer's instructions.

Disk drives in use can be a little noisy. When activated, they can suddenly 'click' as the motor starts up and the recording head is moved into contact with the disk surface. In cases where drives are clicking on and off in quick succession, the start-up noises can be a nuisance. Some manufacturers therefore provide other internal switches which can be so set that when one disk drive starts up, all the drives start up together. There is thus one main start-up noise and the computer can then use any drive that it wishes in relative quiet. Disk operation is then noticeably faster because of the

absence of further start-up times. These apparent advantages must be weighed, however, against the increased wear that both disks and drives will suffer, and it is for this reason that a choice is provided.

Where files are stored on a disk

Unlike cassette recorders, disk drives allow at least two ways of storing files, and a microcomputer manufacturer must pre-programme his machine to use one method or the other. Some microcomputers, for example, instruct disk drives to store files in sequence starting from the outer tracks and working inwards, so that each file sits as a complete block on the disk. The problem with this system is that erased files will leave gaps unless something is done to keep moving the remaining files closer together. Other machines overcome this problem by splitting files up and storing the bits in any convenient gaps, but of course many more locations also have to be remembered.

How files are stored and transferred

The memory in a microcomputer is made up of thousands of tiny electrical units which in computer jargon are called *bytes*, and each unit or byte is capable of remembering any one of the keyboard characters. The word HELLO, for example, would be stored in five bytes, one for each letter.

Actually the characters themselves are not remembered, because microcomputers are only designed to store numbers. Manufacturers must therefore pre-programme their machines with a special code whereby each keyboard character is given a unique number. In the case of machines such as the BBC Micro, each byte can store any number from 0 to 255, and not surprisingly, therefore, the code for the keyboard is in this range. Just to complicate matters there are at least two other codes also operating in most microcomputers, and they too use numbers in the range 0 to 255, albeit in different ways. Manufacturers must therefore provide their machines with at least three different decoders or interpreters capable of

understanding the various number codes operating inside their machines.

The keyboard, for example, is coded as numbers according to the American Standard Code for Information Exchange, usually known in computer jargon as ASCII. Everything typed in at the keyboard starts off life in this way. Files written in the computer language BASIC, however, undergo further coding so that they use less memory. To do this, the manufacturer provides a code which recognises each BASIC keyword by a unique number, and the operating of this code is controlled by something called the BASIC interpreter. For example, the word PRINT would normally occupy five bytes in memory, but is actually further coded so that it can be represented by a single number and reduced to one byte. All BASIC keywords are condensed in this way, so that programs written in BASIC are stored as a mixture of normal ASCII code and condensed BASIC code, and cannot be understood properly by an ASCII decoder alone.

When a BASIC program is run, the BASIC interpreter takes each line in turn and converts it into yet another code. When the command PRINT, for example, is obeyed, the final result is that something appears on the screen. What actually happens is that when the number representing PRINT is detected, the BASIC interpreter activates a part of the permanent memory of the computer which the manufacturer has pre-programmed to carry out the various steps leading to the screen display. This pre-programmed memory uses yet another code called machine code, again using numbers from 0 to 255. When a BASIC program is run, its mixture of ASCII code and condensed BASIC code is in effect translated into machine code, and it is the machine code which controls the actual workings of the computer. It is also possible to write files directly in machine code, the advantage being that without the need to translate BASIC they run very much faster.

Text or data files are usually stored entirely in ASCII code, whereas programs are stored as either ASCII/condensed BASIC or as machine code. In each case, the result is simply a sequence of code numbers stored in memory, and it is these which are transferred between microcomputer and disk drive. The important point as far as the BBC machine is concerned is that certain commands to the disk drives are designed to act

only on files of one code type, and cannot be used on the other codes. See, for example, *TYPE (ASCII code only) and *RUN (machine code only) later in this book.

The need for buffers

To compensate for the difference in rates at which the computer and disk drive can each receive or transmit information, some form of intermediate memory is used. Thus if a file is being transferred to disk, the sequence of numbers is first copied into the temporary memory at a rate which suits the computer. When the temporary store is full, it is 'emptied' or copied on to disk at a rate which suits the disk drive. The temporary memory can then be refilled and the process repeated until the whole file has been transferred. In computer jargon, the temporary memory is called a buffer, and it is normally built into the computer by the manufacturer. If you recall that each number is stored in a byte, the buffer normally holds one sector's worth of bytes, and this is another reason why it is convenient to divide tracks into sectors.

Getting instructions

Disk drives are really just glorified tape recorders, and most of them receive all their instructions from their microcomputer. Being capable of many more operations, however, they require a much larger set of command words. The instructions themselves are stored in the computer's memory and are activated by typing in the appropriate command word, just like normal BASIC keywords.

However, the disk drive instruction set is not always provided with a standard microcomputer, but may have to be purchased as an accessory in one of two forms. Firstly, some microcomputers use a system where the set of instructions is provided on a separate floppy disk. When it is desired to use the disk drive, the instructions must first be transferred to the computer like a normal file. In computer jargon these are known as disk operated systems, and their main disadvantage is that they take up some of the available memory in the

computer. The alternative system is to provide the set of instructions in the form of pre-recorded extra memory which is simply plugged into a suitable socket inside the computer. The silicon chip which is used is often referred to as a disk interface. This has the advantage that the original computer memory is still free, but it may be difficult to alter the instruction set if required. In computer jargon, the instruction program is in this case said to be contained in read-only-memory or ROM, and the disk system is ROM-based.

Summary

Disk drives should therefore be regarded simply as rather good tape recorders with some of the features of record players, able to transfer files rapidly and to provide immediate access to any part of any file. It is important to remember, however, that in most cases the instructions controlling the disk drive operations are coming from the computer, and are not a part of the disk drive itself.

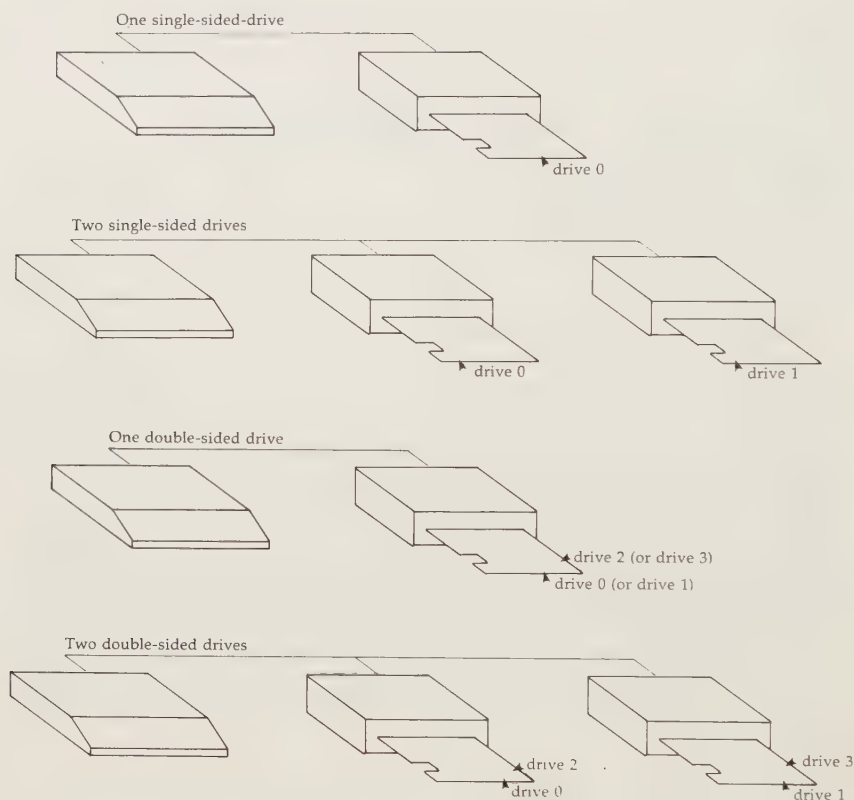
Disk Drives and the BBC Micro

The BBC Micro can handle up to four disk drives, and the additional commands required are stored in a plug-in disk interface within the computer. The necessary buffer is not part of the interface, but is created from available memory when the interface is fitted, reducing slightly the amount of memory for writing files. Both 40-track and 80-track drives can be used, and files are stored consecutively in complete blocks, starting from the outermost track and working inwards. There are ten sectors per track, and the machine only operates in single density.

Numbering the drives

Although the BBC Micro can deal with up to four disk drives, it can only control two drive motors. This means that if you want four drives, you must use two double-sided drives rather than four separate single-sided drives. The drives themselves are recognised by the computer as drives 0, 1, 2 and 3 according to the following simple rules.

1. If you have only one single-sided drive, the computer will recognise it as either drive 0 or drive 1, depending on how the internal switches in the drive have been set. Normally, a single drive should be set to be drive 0, for the simple reason that when the computer is first switched on, or when BREAK is pressed, the computer automatically connects up to drive 0. To use other drives, a drive number must be included with any commands, and setting a single drive to be drive 0 avoids the problem.



2. Two single-sided drives are always drives 0 and 1.
3. When two double-sided drives are used, the reverse side of drive 0 is drive 2, and the reverse side of drive 1 is drive 3.
4. Even if only one double-sided drive is used, the rule for numbering double-sided drives still applies. If the internal switches are set for drive 0, the reverse side will be drive 2, and not drive 1 as you might have expected.

You can check that this is indeed the way that drives are numbered, even if you have only one single-sided drive 0. When you have become more familiar with handling disk drives later in this book, try SAVEing a file to drive 2 rather than drive 0. You will find that the computer will start up the motor for drive 0, but since you have only a single-sided drive it will regard drives 0 and 2 as being on the same side of the disk rather than on opposite sides, and will SAVE the file on

what appears to be drive 0. Try SAVEing a file to drives 1 or 3 however, and the computer will report a drive fault.

Disk capacity

The BBC Micro will accept either 40-track or 80-track drives, and each track is divided into ten sectors. Each sector stores 256 bytes, so that a 40-track disk has 400 sectors and can store 102400 bytes. In computer jargon this figure is rounded down slightly to 100000 and is usually written as 100K. An 80-track disk would have double this capacity, and four 80-track drives would provide a massive storage capacity of 800K, enough to remember the whole of a fair length novel when you consider that a byte is equivalent to a keyboard character!

File storage

Files are stored in consecutive blocks on each disk, from the outermost track inwards, starting at the first free sector after the previous file. The outermost track is called track 0, and its first two sectors are called sector 0 and sector 1. It is here that the computer stores all the information on file names and their locations on the disk. In fact, sector 0 stores the actual names, and sector 1 stores all the other information such as file location, file length, and where the file is to go when transferred back to the computer (in computer jargon, the load address). There are therefore only the 256 bytes of sector 0 in which to store file names, and this limits the number of files that can be stored on one disk. File names can be up to seven characters long, and if all seven are not used the computer will make the number up to seven with blank spaces when storing the file name. Each file name should therefore use up seven bytes of sector 0, but a further byte is also required to indicate something called the directory of each file (see later). Thus each file name takes up eight bytes, so that a maximum of 32 names can be stored in sector 0. The first of these is reserved for the title of the disk itself, which the user can choose through the command `*TITLE`, so that each disk (or rather each drive) can only store 31 files, even if this leaves a lot of empty capacity

still on the disk. The first file will then be stored from sector 2 onwards, and in calculating file location the computer ignores track numbers and simply regards the disk as a sequence of numbered sectors. Sector 9, for example, is the last sector of track 0, and sector 10 is the first sector of track 1. The computer will in fact display the contents of sectors 0 and 1 on the command `*INFO`, including the sector at which each file starts, but if you think that sounds easy then beware! The sector numbers and other information such as file length (i.e. number of sectors used) are displayed in hexadecimal notation rather than decimal, and if you are not familiar with hexadecimal numbers it will seem a little confusing.

Finding room

When files are transferred to disk the number of sectors required is determined by the file length, and if there is insufficient space left on the disk a suitable error message will be displayed. When data files or text files are created with the BASIC command `OPENOUT` however, the length of the finished file is not known at the start. The computer overcomes this problem by creating an initial blank file 64 sectors long on a disk, then fills the file with the information as it is presented and finally ends the file at its actual length. If such a file were 10 sectors long for example, it would initially be 64 sectors long and then the computer would shorten the file to contain just the necessary information. Temporarily then, 64 sectors (or 6.4 tracks!) are required, and if these are not available a suitable error message will again appear.

All is not lost however! Since files are stored as complete blocks and erasure leaves gaps on the disk, the disk interface carries a command to compact the disk when required. An error message that a disk is full can often be overcome by simply compacting the disk and freeing more space.

When a file is `SAVED` on a disk, the BBC Micro will examine the locations of all the files already on the disk, starting with the outermost one and working inwards, and will store the new file in the first available space large enough to take it as a block. In this way the computer does at least partially remove any gaps left by deleted files.

The disk interface

The necessary set of instructions for controlling a disk drive is not provided with the standard BBC Micro, but must be purchased as an extra in the form of a disk interface which is simply plugged into the correct socket within the computer. In computer jargon the DFS in the BBC Micro is ROM-based.

As pointed out previously, the problem with this system is that it is not usually possible to change the instructions without buying a new interface. The BBC Micro interface, however, uses a special form of silicon chip in which the read-only-memory can be erased by ultraviolet light if reprogramming is required. In computer jargon the interface is really an 'eraseable/programmable read-only-memory' or EPROM, so that the BBC Micro DFS is actually said to be EPROM-based.

The interface is simply an add-on electronic component containing a program similar to the one already in the standard machine for recognising BASIC commands. It will recognise about 19 extra commands, and will control the disk drive accordingly. It is the purpose of the remainder of this book to explain these additional commands in detail.

Using your disk drive

This chapter assumes that you have a BBC Micro fitted with an Acorn disk interface, and that one or more disk drives have been connected in accordance with the manufacturer's instructions. To make the best use of the information, it is intended that you sit down at your computer and work through the chapter step-by-step until all command words have been covered. If you have a different interface fitted, the commands will be largely the same but the display may vary slightly.

Now switch everything on. In the top left-hand corner of the screen you should see the following display:

```
BBC Computer 32K
```

```
Acorn DFS
```

```
BASIC
```

```
:
```

The part of the display which says 'Acorn DFS' only appears when a disk interface has been fitted, and indicates that the computer is set up to work with disk drives.

Tape or disk?

When the BBC Micro is switched on, it automatically assumes that it is connected to a disk drive rather than a cassette tape recorder. In computer jargon, it is said to default to a disk filing system. If you wish to use a tape recorder, you must first type in the command `*TAPE` as explained in the *BBC User Guide*. If you then wish to return to a disk system you can either press `BREAK` or type in the command `*DISK` (or `*DISC` — either spelling will do).

Getting a list of command words

Now type `*HELP` and press RETURN. This command is not part of the disk interface but is in the main operating system of the standard machine. What it does is to display on the screen a list of all the add-on ROMs and EPROMs which have been fitted to the machine (there are many more besides a disk interface that can be purchased). If the only extra in your machine is a disk interface, the screen display should be similar to

```
*HELP
DISK 0.90
DFS
UTILS
DIS 1.20
```

This indicates that the disk interface is version 0.90, that it contains two programs called DFS and UTILS, and that the microcomputer operating system is version 1.20. If you have a different issue computer or interface, the version numbers may differ slightly from those above.

Now type `*HELP DFS` and press RETURN. You should see a display on the screen similar to

```
*HELP DFS
DFS 0.90
ACCESS <afsp> (L)
BACKUP <src drv> <dest drv>
COMPACT (<drv>)
COPY <src drv> <dest drv> <afsp>
DELETE <fsp>
DESTROY <afsp>
DIR (<dir>)
DRIVE (<drv>)
ENABLE
INFO <afsp>
LJB (<dir>)
RENAME <old fsp> <new fsp>
```

```
TITLE <title>
WIPF <afsn>
```

```
OS 1.20
>
```

These, in fact, are all the command words recognised by the program called DFS, together with what they should be followed by (in computer jargon, their syntax). Now type **HELP UTILS* and press RETURN to get a display similar to

```
>*HELP UTILS

DFS 0.90
  BUILD <fsp>
  DISC
  DUMP <fsp>
  LIST <fsp>
  TYPE <fsp>
```

```
OS 1.20
>
```

These are the command words recognised by the program called UTILS. Being able to remind yourself of the commands in this way is very useful once you are familiar with the workings of your disk drive.

All these commands must be prefixed by an ***, and the abbreviations after each command have the following meanings:

drv	the drive number (i.e. 0, 1, 2 or 3).
src drv	source drive (for copying drive-to-drive).
dest drv	destination drive for copying.
dir	directory (see later).
L	used with the command <i>*ACCESS</i> to protect individual files against erasure, just as an adhesive tab can protect a whole disk.

title	the title you can give to a disk using the command *TITLE.
fsp	the name that you give to each file. In computer jargon it is called the filespec, hence fsp.
afsp	means 'ambiguous filespec'. Given the speed of operation of disk drives, it is possible for certain commands to act on several files together rather than having to repeat the same command for each file. For example, it is possible to erase several files at once with the command *DESTROY, by replacing part of the file name with *. Thus *DESTROY will not distinguish between file names PROG and PR*, and if the computer is asked to destroy a file called P*, it will actually erase all files beginning with the letter P. Because 'P*' can represent more than one file, its meaning is ambiguous — hence 'afsp'. More will be said about this later, but for now bear in mind that it only applies to certain commands.

Other command words

Two further commands intended for disk drives, PTR# and EXT#, are actually in the standard microcomputer rather than the disk interface, and are used to extract individual items of information from data files and text files.

In addition, the following disk commands work equally well with cassette recorders, and are therefore also part of the standard computer rather than the disk interface.

*RUN
 *SPOOL
 *SAVE
 *EXEC
 *LOAD

These commands are described in the *BBC User Guide* for cassette recorders only. Disk usage is not covered.

Starting with a blank disk

The first task with a new blank disk is to lay down tracks and sectors, or in computer jargon to format it. The program for doing this is not part of the BBC Micro, but must be purchased separately, usually in the form of a floppy disk from either the disk drive manufacturer or the computer manufacturer. If your machine is fitted with a non-Acorn interface you may be lucky enough to have a formatting program built into the interface, in which case you will not need a separate floppy disk program. The different programs available will vary slightly in the way that they prompt the user, perhaps with differing screen displays or sound effects, but the essential procedure is as follows:

1. Place the formatting disk into drive 0 in accordance with your disk drive instruction manual.
2. Transfer the formatting program to the computer according to the manufacturer's instructions.
3. If you have more than one drive, put your new blank disk into one of the spare drives. If you have only one single-sided drive, remove the formatting disk and replace it with a blank disk.
4. Follow the prompts on the screen to run the formatting program. You will be asked, for example, which drive you wish to format.

If your disk interface has a built-in formatting program, you will not require steps 1 and 2 but will be able to run the program directly by following the interface manufacturer's instructions. You can't really go wrong since you will be prompted at all stages, unless you accidentally re-format a disk containing files, in which case the files will be erased. Unfortunately, step 2 above uses a little of the computer memory to store the formatting program, and any files still in the computer could be partially erased. It is therefore always good practice to carry out any formatting before beginning work, to avoid the possibility of losing precious data.

Now format *two* floppy disks before going on.

Viewing the contents of a disk

Place a freshly formatted disk in drive 0, type *CAT and press RETURN. The following display will appear on the screen.

```
*CAT
(00)
Drive 0                      Option 0 (off)
Directory :0.$              Library :0.$

>
```

This, in fact, is the heading for what is called the disk catalogue, and would normally be followed by a list of the names of files on the disk. Since there are no file names to be seen, the disk is obviously blank. To put some file names on to it, type in the following:

```
SAVE"PROG1" then press RETURN
SAVE"PROG2" etc.
SAVE"PROG3" etc.
SAVE"PROG4" etc.
SAVE"PROG5" etc.
```

Note that from now on the need for you to press RETURN will be assumed, and will not be included in any instructions.

Although there are no actual files in the computer to be SAVED, the correct procedure will be followed and each file name will be duly logged in sector 0. In fact each empty file will be allocated just one sector, starting with sector 2. These 'files' can be manipulated just like real files, and are a useful ploy when you are investigating the workings of your disk drive.

New repeat *CAT, and the display will become

```
*CAT
(05)
Drive 0                      Option 0 (off)
Directory :0.$              Library :0.$

    PROG1                     PROG2
    PROG3                     PROG4
    PROG5
```

You can see that the five files are indeed now on the disk. In fact, you will soon discover, when you use your disk drive for real, that the file names are listed alphabetically rather than in the order in which they lie on the disk.

But what do the various headings mean?

(00)

this is called the cycle number, and it increases by one every time the disk is altered in some way. In the example above five files have been added to the disk, and so the cycle number is now (05). After (99) the number returns to (00). Its usefulness is not immediately apparent, except perhaps as an indication that a disk has been tampered with. Unfortunately, the number itself can be tampered with easily without affecting the disk contents, for example by means of the command *TITLE (see later).

Drive

This simply indicates which drive is being catalogued. Since your disk is in drive 0, it is this which is shown. To see the contents of disks in other drives, you must use

*CAT 1
or *CAT 2
or *CAT 3

If you have more than one drive and you try this, you will see that the drive number changes accordingly in the catalogue heading.

Directory

A little more complicated this, but still quite straightforward. First, do you recall that we didn't need to type *CAT 0 originally, simply *CAT was sufficient? The computer is designed to think that it is connected to drive 0 when first switched on, and it assumed that we meant drive 0 when we typed in *CAT. In computer jargon drive 0 is called the default drive, and all commands will be directed to it if you do not indicate which drive

you want. The default drive can be changed if you wish by using

```
*DRIVE 1
or  *DRIVE 2
or  *DRIVE 3
```

If then you wish to catalogue drive 0, you must of course use *CAT 0, since *CAT will now catalogue the new default drive. You can experiment with this even if you have only a single drive attached to your computer, and you will soon discover that the number after the word 'Directory' in the catalogue heading indicates the current default drive.

To return to drive 0 as the default drive, simply press BREAK.

Now press BREAK to reset the default drive, and use *CAT to confirm that according to the heading 'Directory' the default drive is indeed drive 0. But what does the dollar sign (\$) mean? Well, let's find out by doing the following.

```
SAVE"A.PROG"
SAVE"B.PROG"
SAVE"C.PROG"
*CAT
```

You should now have the following screen display:

```
>*CAT
  (08)
Drive 0          Option 0 (off)
Directory :0.$   Library :0.$

  PROG1          PROG2
  PROG3          PROG4
  PROG5

A.PROG          B.PROG
C.PROG
```

Note first that the cycle number has increased by three because three files have been added to the disk. But note mainly that the new files have been listed separately from the old ones. This is because the computer does not regard "A.PROG" as a file of that name, but rather as a file called "PROG" which is in directory A. Similarly "B.PROG" is in directory B, and "C.PROG" is in directory C. All files must go into one directory or another, and just like the default drive there is a default directory ready waiting should you choose not to specify one. When the computer is first switched on, the default directory is automatically set to be directory \$, and this is the reason for the dollar sign after the word 'Directory' in the catalogue heading. When you perform *CAT, the contents of the default directory are displayed first, followed by a space and then the contents of the other directories.

Directories are a way of grouping the contents of a disk, and they have two main advantages. Firstly, they allow several files of the same name to be stored on a disk, whereas normally a file is erased when one of the same name is SAVED. Secondly, certain commands can be made to act on all the files in one particular directory in isolation, by means of ambiguous filespecs. Directories therefore are one way of being selective in manipulating files on a disk.

Almost any character on the keyboard can be chosen to indicate a directory, and the character is stored in the eighth byte reserved for the file name in sector 0 — remember?

Just as you can alter the default drive, so too you can alter the default directory using the command *DIR. Try, for example,

*DIR A

*CAT

You should now see two changes in the screen display. Firstly, the 'Directory' is seen to be ':0.A'

in the catalogue heading, meaning drive 0 directory A. Any files saved from now on without specifying drive or directory will go into drive 0 directory A. In all other drives, however, the default directory will still be directory \$. Secondly, "A.PROG" is now listed first, because it is in the default directory.

Try changing the default directory and/or the default drive as much as you like, then finally press BREAK to restore both to their original settings.

Library

When the computer searches a disk for a particular file, it searches only the one drive and directory and will give up if the file cannot be found. The use of a 'Library' enables one other directory and/or drive to be searched as well, rather like the reference section of a library. Again the library consists of a specified drive and directory, the normal default setting being drive 0 directory \$. The library can be changed readily by the command *LIB. For example,

*LIB :1.A will set the library to be drive 1 directory A

Try this and you will find the 'Library' changes accordingly in the catalogue heading. Suppose that you did this, and then asked the computer to search drive 0 for a particular file. If it could not be found on drive 0, the computer would go on to search drive 1 directory A before giving up. If you think that sounds useful for storing certain files without having to remember which disk they are on, I'm afraid the library only works with machine code files, so unless you are using these forget it. It really comes into its own with networked systems, where individual users can call up a file from a central source without having to know the drive number.

Try altering the library as much as you like, then press BREAK to reset the machine, and perform

*CAT to return to the original display.

Option

The BBC disk system provides the useful ability to load-and-run one special file on a disk when the SHIFT key is held down and BREAK is pressed. For the facility to work, this file must be given the special file name !BOOT, and the file itself must be created using the command *BUILD (see later). The autorun procedure is sometimes called auto-booting, and exactly what happens when SHIFT and BREAK are pressed at the same time is decided by the command *OPT 4 (see later). It is possible to have no action, to LOAD a file without running it, or to load-and-run it, depending on which option you choose. At the moment your catalogue heading shows Option 0 (off). This means 'no action' on SHIFT/BREAK. How the option setting is changed and how the special !BOOT file is created will be dealt with later.

Sending commands to different drives and directories

You should now understand default drives and directories, so that if you issue a command without specifying either, the computer will assume that you mean the default ones. If you specify a directory but not a drive, the computer will assume you mean the default drive. If, on the other hand, you specify a drive but not a directory, the computer will assume that you mean the default directory, and this can change from drive to drive. In the default drive, the default directory is set by the heading 'Directory' in the disk catalogue, but in all other drives it remains directory \$.

How do you do all this specifying? The answer is simply to use a colon for drive number and a full stop for directory just as in the heading 'Directory' in the disk catalogue.

Examples:

SAVE"PROG"

stores PROG in default drive and default directory.

SAVE"A.PROG"

stores PROG in default drive and directory A.

SAVE":2.A.PROG"

stores PROG in drive 2 directory A.

SAVE":2.PROG"

stores PROG in the default directory of drive 2.

You can try this even if you have only one drive, just to confirm that the computer is indeed trying to go to another drive (but remember, drive 0 and drive 2 are the same on a single drive). All you will get is an error message reporting a drive fault, but don't worry. Just press BREAK and carry on.

Although previously we have talked about names such as PROG being called the filespec, you should now be able to appreciate that the proper filespec should also include the drive and directory.

Use of inverted commas

As you will already know, BASIC commands such as LOAD, SAVE and CHAIN require inverted commas around the file name. Disk commands beginning with * on the other hand, work equally well with or without them, so forget them unless the computer puts up an error message telling you to include them. For example, *DELETE PROG is the same as *DELETE "PROG".

Ambiguous filespecs

A further word on the ability of the BBC Micro to handle several files with one command is necessary before we see just what the disk drive can do. Certain commands such as *DESTROY will recognise the symbol * in a file name as meaning 'any group of letters'. For these commands,

PROG

PRO*

PR*

P*

*

can all mean the same, and the number of fixed letters included is just a way of being selective. P*, for example, will apply to every file beginning with P, whereas PRO* will only apply to those beginning with the letters PRO. The symbol * on its own applies to every file! The only real rule is that the * must come at the end of a word, i.e. must not be followed by any letters. For example, *G would not mean 'all files ending in the letter G'. Instead, an error message would be displayed.

In the same way that * means a group of letters or characters, so # can be used to mean 'any single character or letter'. It is a little more selective than *, and unlike * it can be followed by one or more letters or characters. For most purposes though, * is all you will ever need to use.

The actual commands which will accept an ambiguous filespec will be explained as you work through this book. For now though, you should be able to appreciate that the symbols : . # * have special meanings to the computer as far as disk drives are concerned, and should not be used as part of the full file name otherwise great confusion could result!

Giving the disk a title

This is done by the command *TITLE. To give the disk in drive 0 the title PROGS for example, press BREAK to ensure that the computer is reset, i.e. that drive 0 is the default drive, then type

```
*TITLE PROGS
```

```
*CAT
```

You will now see the title PROGS at the top of the catalogue next to the cycle number. The command will only work with the default drive however, so that if you try

```
*TITLE :1.PROGS
```

you will give the disk in drive 0 the title ':1.PROGS' rather than the disk in drive 1 the title PROGS. It is the only exception to the general rules for directing commands, and you must change the default drive to title other disks. Titles may be up to twelve characters long, and since there is only room to store eight of them in sector 0 along with the file names, the

other four spill over into sector 1. If you don't use all twelve characters, the number will be made up to twelve by blank spaces when the title is stored on the disk.

To erase the title altogether, try

```
*TITLE ""
```

This is the one case of disk commands where inverted commas are unfortunately necessary, but if you forget don't worry. The computer will soon remind you!

You may have noticed that this command increases the cycle number by 1, so if you want a simple way to increase the cycle number by 50, say, try this simple program.

```
10 FOR X=1 to 50
20 *TITLE "" (or whatever title you want)
30 NEXT
```

In this way you can adjust the cycle number to any desired value and use it to check that a disk has not been tampered with (or to cover your tracks if you have been doing the tampering!).

Copying a disk

This is almost essential if you don't want to risk damaging or losing your only copy of precious files. The two commands provided for this are

***BACKUP**

makes copy of whole disk.

***COPY**

makes copy of selected files only.

(1) *BACKUP

This command must first be enabled with the command ***ENABLE**, and must be followed by the numbers of the source and destination drives. For example, to copy your PROGS disk, place it in drive 0. If you have more than one drive, place your second newly formatted disk in drive 1. Then type

```
*ENABLE
*BACKUP 0 1
```


The computer will do the rest, copying every track automatically. If you have only a single drive, type

```
*ENABLE
*BACKUP 0 0
```

The computer will then prompt you with suitable screen messages to alternately insert the two disks. It's a bit tedious since it has to be done a few tracks at a time, but at least it works.

The only minor irritation is that any mistake, such as wrongly spelling BACKUP, means that you have to *ENABLE all over again rather than just re-doing the *BACKUP command.

(2) *COPY

This command works like *BACKUP, except that it does not need to be enabled and file names must be given. For example,

```
*COPY 0 1 PROG
```

will copy the file PROG from the default directory in drive 0 to the same directory in drive 1. Similarly,

```
*COPY 0 1 A.PROG
```

will do the same thing in the A directory of both drives. The command *COPY will accept an ambiguous filespec, so that,

```
*COPY 0 1 A.*
```

will copy everything from the A directory in drive 0 to the A directory in drive 1. Even the directory character can be replaced by # or *, so that,

```
*COPY 0 1 *.*
```

will copy everything from drive 0 to drive 1, just like *BACKUP except that only the files are copied whereas *BACKUP copies every track, including any gaps due to files which have been erased.

Since *COPY only copies actual files, it can be used as a way of copying a disk and at the same time compacting the copy. Again, if you have only one drive you must use *COPY 0 0 and follow the screen prompts to insert the source disk and

destination disk alternately.

You should at this stage have made a copy of your PROGS disk with *BACKUP. If you have a third blank disk available and you wish to see *COPY in action, it would be a good idea to make a copy of your formatting disk at this stage.

One slight problem with both commands — they work by transferring the disk contents back into the computer temporarily, then out again to the destination disk. Any files already in the computer could be erased — so beware!

Erasing files

The three commands provided for this purpose are:

*DELETE

erases individual files.

*DESTROY

erases whole groups of files.

*WIPE

erases whole groups of files, but with ability to sort through group just before deletion.

To see these in action, place one of your PROGS disks in drive 0, and type

```
*DELETE A.PROG
```

```
*CAT
```

You should see that A.PROG has been deleted, and the cycle number has increased by 1. Now return A.PROG to the disk, and type

```
*ENABLE
```

```
*DESTROY P*
```

Note that *DESTROY must be enabled, and can be used with ambiguous filespecs. You should see on the screen a list of the files in the default directory of drive 0, together with a request for confirmation of your wish to delete them. If you answer Y they will all be deleted, but for now answer N. Then type

```
*ENABLE
```

```
*DESTROY *.*
```

You should now see a list of all files in every directory of drive 0. Again answer N. Note that to delete files in other drives you must simply add the drive number. For example

```
*ENABLE
*DESTROY :1.*.*
```

would allow you to delete all files on the disk in drive 1 at a stroke. Now type

```
*WIPE P*
```

You will be presented with all the files beginning with P in the default directory of drive 0, just like `*DESTROY P*`, except that you will see them one at a time and can decide about erasure for each one separately. Answer N to them all, then type

```
*WIPE *.*.*
```

All the files on the disk in drive 0 will now be presented for deletion one at a time. Answer N to them all again. Note that this command, like `*DESTROY`, will accept ambiguous file-specs. Unfortunately, the use of ambiguous filespecs cannot be extended to drive numbers, so that, for example

```
*WIPE :*.*.*
```

will not allow you to wipe out all files on all disks in one go. Instead you will get an error message saying that the computer does not recognise the drive number.

Protecting files

Accidental erasure of a file can be very annoying, and can happen if, for example, you `SAVE` a file and give it the name of one already on the disk, in which case the existing file will be erased in the process. Incidentally, the new file will be stored on the disk in the first large enough gap, starting from the outermost track and working inwards. This may or may not be the gap left by the old file, depending on the length of the new file and whether or not other files on the disk have previously been deleted.

Fortunately, the BBC Micro allows you to protect individual files by 'locking' them, just as the protective tab on the disk

sleeve protects the whole disk against erasure. The command used is **ACCESS*, and to see it in action type

```
*ACCESS PROG1 L
*CAT
```

You should see that PROG1 in the default directory of drive 0 has a letter L after the filename, indicating that it is locked. Any attempt to use the filename again or **DELETE* it will result in an error message that the file is locked. Any attempt to **DESTROY* it or **WIPE* it will not even present the file for deletion. The use of **ACCESS* will not, however, protect a file against re-formatting or **BACKUP*. Only a protective tab on the disk sleeve can do this.

In a similar way

```
*ACCESS :1.A.PROG L
```

will lock PROG in the A directory of drive 1. The command accepts ambiguous filespecs, so try

```
*ACCESS * L
*CAT
```

and confirm that all files in the default directory of the default drive are now locked. Then type

```
*ACCESS *.* L
*CAT
```

and confirm that all files in all directories are now locked. Try erasing them and observe the effect of the locking procedure.

To unlock each file, simply type in the same commands without the letter L. For example, type

```
*ACCESS B.PROG
*CAT
```

and confirm that B.PROG is now unlocked. Then try

```
*ACCESS *.*
*CAT
```

and observe that all files are now unlocked.

Looking at each file

With all this saving and deleting, it would be nice to know just where each file is located on a disk, and perhaps even what each file contains. Obviously we can *CAT the disk to see a list of filenames, and if a BASIC program is present we can LOAD it and LIST it. But is there anything else which we can find out? Well, the following six commands will all provide us with further information:

*INFO

gives all the information in sectors 0 and 1, i.e. filenames, locations, lengths, load addresses, etc.

*COMPACT

compacts a disk then performs *INFO.

*OPT 1

sets the computer so that *INFO is performed on files when other commands such as LOAD are given.

*DUMP

displays file contents byte by byte, i.e. the code numbers themselves, together with an attempted ASCII translation of the code.

*TYPE

lists any file coded entirely in ASCII, e.g. any file created with *BUILD or *SPOOL, or any text or data file. Does not work with BASIC or machine code files.

*LIST

like *TYPE, but with line numbers in steps of one added.

To see all these in actions, the disk must include at least one real file. Therefore type

```
10 PRINT"THIS IS *DUMP"
SAVE"DEMO"
*CAT
```

You should see that the simple BASIC program called DEMO is now on the disk. Then type

```
*INFO PROG1
```

and you should see the following display:

```
*INFO PROG1
#.PROG1          FF1900 FF8023 000002 002
^
```

Reading from left to right, this indicates that PROG1 will LOAD back into the computer starting at memory address 1900 hexadecimal, that the execution address, i.e. the place where execution of the file starts, is 8023 hexadecimal, that the file is two bytes long, and that it is stored on the disk starting at sector 2.

Several points are worth noting. Firstly, memory address 1900 hexadecimal is the normal start of computer memory for BASIC programs when a disk interface is fitted. The address at which BASIC programs are executed is 8023 hexadecimal, because this is where the BASIC interpreter starts. BASIC files always start with an 'end-of-line' marker byte and finish with an 'end-of-file' marker byte. Since PROG1 was actually empty, the entire stored 'file' contains only these two bytes. Finally, since PROG1 was the first file to be stored on the disk, it is bound to start at sector 2. Since it is only two bytes long and each sector can hold 256 bytes, it will obviously not go beyond sector 2. In a similar way, PROG2 will be in sector 3, PROG3 will be in sector 4, etc.

To confirm this, note that *INFO accepts an ambiguous filespec, so type

```
*INFO *.*
```

and to see how a disk can be compacted try

```
*COMPACT
```

Note that when *INFO is performed as part of *COMPACT, files are listed in reverse order. To compact a different drive, simply type *COMPACT 1, etc.

Now type

```
*OPT 1 1 (or *OPT 1,1)
LOAD"A.PROG"
```

Note that the file details for A.PROG are displayed as the command LOAD is obeyed. This will also happen when other

commands are given. To turn it off type *OPT 1 0, and to turn it on use *OPT 1 n, where n can be any number from 1 to 255. It is also turned off when BREAK is pressed.

Unfortunately, as you perform *INFO on longer files you will soon discover that in addition to the memory locations, both file length and file location are actually being displayed in hexadecimal notation, so that you really need to understand hexadecimal numbers to make proper use of the command.

Incidentally, if you have been wondering what FF means before the memory locations, it indicates that the file will be dealt with by the main computer processor rather than a second processor purchased as an accessory, in which case 00 would replace FF.

Now type

*DUMP PROG1

and you should see the following display:

```
*DUMP PROG1
0000 0D FF ** ** ** ** ** ** ** ** ** 
>
```

Believe it or not, this is a display of the actual bytes making up the file PROG1, which we know from *INFO is two bytes long. Reading from left to right, the left-hand column showing 0000 indicates that the line starts at byte 0 of the file. With a long file, the line would then show the code numbers in the first eight bytes of the file, in hexadecimal form. In this case however, the file is only two bytes long, and the computer has simply filled up the other six places on the line with double asterisks. The first byte 0D appears at the end of every BASIC program line as a sort of 'end-of-line' marker. Similarly, it appears at the beginning of every BASIC program to indicate that a new line is about to start. The second byte FF is an 'end-of-file' marker for BASIC programs, and always appears at the end of a BASIC file. Since PROG1 is actually empty, the file simply consists of the starting byte 0D and the ending byte FF, which in the decimal system are the numbers 13 and 255.

And the row of eight dots on the right of the display? This is where the computer tries to translate each code number in the line according to the ASCII code. Whenever there is no

character corresponding to a particular number, the computer puts a dot for that number instead. Here you can see complete failure of translation.

To make it all a bit clearer, now type

```
*DUMP DEMO
```

and you should see the following screen display:

```
*DUMP DEMO
0000 0D 00 0A 15 20 F1 22 54 ..... ."T
0008 48 49 53 20 49 53 20 2A HIS IS *
0010 44 55 4D 50 22 0D FF ** DUMP"...
>
```

Looking at the left-hand column first, you can again see that the first row starts at byte 0 (actually shown as 0000), so the eight bytes in the first row are bytes 0 to 7. The second row therefore starts at byte 8 (shown as 0008). As usual these are hexadecimal numbers however, so that the third row starts at byte 16, which in the hexadecimal system looks like our number '10' and so is shown as 0010. Again you need to understand hexadecimal numbers to make full use of the display, but some insight can be gained by looking at the translation on the right so long as you remember that only ASCII code is being translated. Thus, for example, the hexadecimal number 53 appears twice in the second row. It is equivalent to the number 83 in the decimal system, and if you look up a table of ASCII codes you will find that the number 83 corresponds to the letter S. Now look again at the translation of row 2, and you will see that the number 53 has indeed been translated as the letter S. Almost all the program has in fact been translated as characters, and since it is a BASIC program you can see that it starts with the byte 0D and ends with the byte FF as expected.

Finally, go back to row 1 of the display and compare it with the first part of the original program. Where is the word PRINT? Working from left to right along the eight bytes, 0D is the expected starting byte for a BASIC file. Then 00 and 0A together contain the next line number, which is calculated by taking the decimal equivalent of the first number 00, multiplying it by 256, and adding the result to the decimal

equivalent of the second number 0A. Since 0A is the number 10 in the decimal system, the line is line 10 as expected. The next byte contains the hexadecimal number 15, which is the same as the decimal number 21. This indicates that the line is 21 bytes long. The next byte contains the hexadecimal number 20, equivalent to 32 in the decimal system, and reference to the ASCII code shows that this is a space. Look at the translation and you can see that it is indeed a space. Now skip a byte to the seventh byte containing the hexadecimal number 22. This is 34 in the decimal system, and is the ASCII code for inverted commas. So the previous byte containing the hexadecimal number F1 must be the compressed BASIC code for the word PRINT. This is the decimal number 241, and you can use *DUMP in this way if you wish to investigate the BASIC code being used by the computer.

Merging files

If you recall, BASIC programs are stored in computer memory as a special code which is a mixture of normal ASCII code and a code for each basic keyword. This is then transferred to disk by the command SAVE. The command *SPOOL, on the other hand, will transfer a program to disk as though it were entirely in ASCII code. For example, the word PRINT will be regarded as five separate bytes, each with its own ASCII code number, whereas SAVE will transfer the program to disk with the word PRINT condensed to the single code number 241, as we have just seen.

So what, you may ask? Well, every character is stored on the disk by *SPOOL just as it was typed in at the keyboard, and when the program is transferred back to the computer later using the special command *EXEC, it will seem to the computer as though it is again coming from the keyboard. If you already have a BASIC program in the computer, the SPOOLED program will add lines to it just as though you had typed them in at the keyboard, and in this way two programs, one in the computer and one on the disk, can be merged. This is the intention of the command *SPOOL.

To see it in action, let's create a program which we shall call MERGE, as follows:

```
10 PRINT"THIS DEMONSTRATES SPOOLING"  
*SPOOL MERGE  
LIST  
*SPOOL  
*CAT (to confirm that the file is on the disk)
```

Note that *SPOOL is given twice, once to open a special text file and once to close it. The command LIST in effect means 'send the program to the special file'.

Now press BREAK and type

```
*EXEC MERGE
```

This command transfers the program back to the computer just as though it were being typed in for the first time at the keyboard. You should see the following display:

```
*EXEC MERGE  
>>LIST  
  
Syntax error  
> 10 PRINT"THIS DEMONSTRATES SPOOLING"  
>>*SPOOL  
  
Syntax error  
>
```

It looks a little strange, with double spacing and error messages, but don't worry. The program will now RUN as normal, just as though it had been freshly typed in. Provided line numbers don't clash, the program being spooled in this way will add lines to any program already in the computer, just as though you were typing them in directly. Where line numbers do clash, the spooled lines will unfortunately replace the ones already in memory, just as would happen if you were actually typing them in.

To LIST the file without LOADING it, simply type

```
*TYPE MERGE
```

Note that when you do this, the words LIST and *SPOOL are seen to be part of the file. The command *TYPE will not work with BASIC files. Try it with the program DEMO and you will see what I mean. To confirm that the file is indeed entirely in ASCII code, try

***DUMP MERGE**

and you will obtain a complete translation of the code character by character.

Renaming a file

Suppose we wish to give the program DEMO the new name DISPLAY. The command for doing this is

***RENAME DEMO DISPLAY**

Try this and use ***CAT** to confirm the change. It is also possible to put the renamed file into a new directory. To see this, try

***RENAME DISPLAY A.DEMO**

Then use ***CAT** to confirm that DISPLAY has been changed back to DEMO and is now in the A directory.

Machine code files

Just as ***TYPE** and ***LIST** are intended for ASCII files, so there are certain commands intended only for machine code files. For example, machine code files are transferred to disk using the command ***SAVE**, which must also include the location of the machine code in the computer's memory. What the command does is to SAVE a specific section of computer memory, and its full form is

***SAVE PROG SSSS FFFF EEEE RRRR**

where PROG could equally well be A.PROG or :1.A.PROG, for example. Although it looks complicated, it is really quite simple. Do you remember that computer memory consists of units called bytes? Well, the bytes are numbered in sequence so that the computer can find its way around its own memory. Files therefore start at one particular byte (the start address) and finish at another (the finish address).

SSSS

is the start address. It is written in this way

because you must give the number in hexadecimal form rather than decimal, and the highest numbered byte in the BBC Micro is a four-digit number in the hexadecimal system.

FFFF

is the finish address plus one.

Machine code programs will LOAD and RUN automatically using the command *RUN, somewhat like the BASIC command CHAIN. Unlike BASIC however, which CHAINS from the first line of a program, machine code programs can be made to run from any particular point in the program provided that an 'execution' address is given during *SAVE.

EEEE

is the execution address. If left out, the computer will assume that the program starts at the beginning, i.e. the start address.

Normally, when a machine code file is transferred back to the computer, it will go into its original place in memory, so that SSSS and FFFF will once again be the start and finish addresses. However, it is also possible to transfer the file back to a different section of memory, and one way to do this is to include a relocation address with *SAVE.

RRRR

is the relocation address. If left out the computer will assume that the file is to go back into its original place in memory.

Unfortunately, as pointed out these addresses are all in hexadecimal notation, so that you will need to understand hexadecimal numbers to *SAVE machine code files.

Want to try an example? Since all it is doing is SAVEing a section of memory, you don't actually need a machine code file although that is what the command is designed for. If you knew the start and finish addresses of a BASIC file it too could be stored using *SAVE, but there is something much more interesting that can be done. Since the screen display is also contained in the computer's memory you can use *SAVE to save any screen display that you wish, provided only that you know the memory addresses involved. So, create any screen

display that you like, in a mode of your choosing, then type

`*SAVE SCREEN SSSS FFFF`

where SSSS and FFFF have the following hexadecimal values.

Model A

	SSSS	FFFF
Mode 4	1800	4000
Mode 5	1800	4000
Mode 6	2000	4000
Mode 7	3000	4000

Model B

Mode 0	3000	8000
Mode 1	3000	8000
Mode 2	3000	8000
Mode 3	4000	8000
Mode 4	5800	8000
Mode 5	5800	8000
Mode 6	6000	8000
Mode 7	7C00	8000

If you then `*CAT`, you will see that a file called SCREEN has been saved. Machine code files may be loaded-and-run by the single command `*RUN`. For example

`*RUN :1.A.PROG`

would load-and-run PROG from directory A of drive 1. For files in the default directory of the default drive, the command may be simplified to `*PROG`. To illustrate this, try typing

`*SCREEN`

O.K., it's not strictly a machine code file, but it's a great way of recalling a dazzling display! Don't forget to be in the right mode when you issue the command though! Also, do not allow the screen to scroll at all while you are creating your display, or you will get some odd split-screen effects on recalling it.

Incidentally, `*RUN` also works with BASIC files, since when they are SAVED the necessary memory addresses are automatically stored in sector 1 of the disk, and all `*RUN` does is to put

the chunk of memory back into its original place in the computer. BASIC programs won't run though, until you give the normal command RUN. To see what I mean, try

```
*RUN A.DEMO (assuming that you have renamed the file properly)
RUN
```

To load a machine code file without running it, the command *LOAD is used. If you want to relocate a file at the same time, one way is to arrange this during *SAVE, as already explained. Alternatively, you can arrange it during *LOAD, for example

```
*LOAD PROG
```

will load PROG into its original place in computer memory, whereas

```
*LOAD PROG 3000
```

will load PROG starting at memory address 3000 hexadecimal. This command will override any relocation address given during *SAVE. It can also be used to load and/or relocate BASIC files, but you may need to press BREAK before running them.

Autorunning programs

As you should know by now, programs can be loaded-and-run by the single commands

CHAIN (basic programs)

*RUN (machine code programs)

*PROG (machine code programs in default drive and default directory)

However, the BBC Micro provides two other ways, both involving the command *BUILD. This command allows you to create directly on a disk a special program which is executed line-by-line by the command *EXEC. To see it in action let's create a program called SHOW. First type

```
*BUILD SHOW
```

The disk drive will start up, and you will be offered line numbers in steps of 1. Now type

```
1 PRINT"LINE ONE"  
2 PRINT"LINE TWO"
```

When line 3 appears, press ESCAPE to transfer the file to the disk, and use *CAT to confirm the transfer. Next type

```
*EXEC SHOW
```

You will see that each line appears on the screen and is executed separately as though it were a separate program. For this reason, FOR . . . NEXT and other loops must be on one line only. To see what I mean try

```
*BUILD LOOP  
1 FOR X=1 TO 10  
2 PRINT"LOOP"  
3 NEXT  
4 (PRESS ESCAPE)  
*EXEC LOOP
```

Files created with *BUILD are stored in ASCII code, and you can list them by using the command *LIST. To see it in action, type

```
*LIST LOOP
```

The line numbers are not stored in the file LOOP, but are provided by the command *LIST. To confirm this, try

```
*LIST MERGE
```

The file MERGE is also entirely in ASCII code, but was created by *SPOOL and has its own built-in line numbers. The command *LIST adds a second set of line numbers, whereas *TYPE did the same thing without adding extra numbers. If you can put up with the odd-looking double set of line numbers, *LIST and *TYPE can be used interchangeably.

Why is *BUILD useful, when a simple BASIC program could have done the same thing? Well, the command comes into its own if you give the program created with *BUILD the special name !BOOT, which is treated by the computer in a unique way. When the SHIFT key is held down and BREAK is pressed, the computer will load-and-run the file called !BOOT auto-

matically, and this saves having to type in any commands. Again though, this may appear to you to be a very limited facility, since files created by *BUILD are executed as separate lines. However, suppose your !BOOT file consisted solely of the line

```
1 CHAIN"PROG" (or *RUN PROG)
```

In effect, you have autostarted the file PROG. What *BUILD provides then is the ability to autostart any one of the programmes on a disk simply by pressing SHIFT/BREAK. Unfortunately, pressing BREAK resets the default drive to drive 0 and the default directory to directory \$, so the !BOOT file must be placed here, and it is not possible to autostart other drives.

To see it in action, type

```
*BUILD !BOOT
1 CHAIN "A.DEMO"
2 (press ESCAPE)
*CAT (to confirm presence of !BOOT)
```

Now a slight complication before we run the !BOOT program. Look at the top of the catalogue. See that word 'Option'? Well, it's connected with the autostart facility and is set by the command *OPT 4. What this command does is to give you a little bit of choice over what happens when you press SHIFT/BREAK, according to the following:

```
*OPT 4 0      no action — the current status of your machine.
*OPT 4 1      *LOAD a machine code !BOOT file, but without
               running it.
*OPT 4 2      *RUN a machine code !BOOT file.
*OPT 4 3      *EXEC a BASIC !BOOT file.
```

Since our !BOOT file is in BASIC, type

```
*OPT 4 3
*CAT
```

Notice what has happened to 'Option' in the catalogue

heading. Now try SHIFT/BREAK and watch the program A.DEMO autorun. Note that unlike 'Directory' and 'Library' the 'Option' is not reset by pressing BREAK, but is actually a facility recorded on the disk itself.

You could of course find other uses for *BUILD !BOOT. One favourite alternative to autorunning a program is to use the !BOOT file to automatically define all the function keys on SHIFT/BREAK.

Abbreviated commands

To save you having to type in full commands, the BBC Micro is programmed to accept abbreviated command words, where a full stop can replace one or more letters. For example, 'P.' is the abbreviated form of the BASIC word PRINT. To confirm this, type

P."IT WORKS"

However, it is not easy to remember all the abbreviated forms of command. Fortunately, 'P.' is not the only acceptable abbreviation, it is simply the shortest. If you care to add any of the other letters the command will still work, i.e. P., PR., PRI., PRIN. and PRINT all do the same thing. You can therefore make a reasonable guess at most abbreviations.

All of the disk commands can be abbreviated in a similar way. For example, the shortest abbreviation for *TITLE is *TI., but *TIT. and *TITL. are also acceptable.

A full list of the disk command abbreviations is given in the reference section.

Random Access Files

In addition to storing programs, both tapes and disks can store lists of information such as names and addresses or tables of numbers. In computer jargon these are called text files and data files, and they can be searched later for selected items of information, either directly by the user or through a suitable program. Unfortunately, they can only be searched once on tape before the tape must be rewound to the start of the file again, and this limits the use of tapes for file handling. Disk drives, on the other hand, provide immediate access to all parts of a file, don't need rewinding, and are ideally suited to repeated searching of information files.

The making and using of such files on disk is quite straightforward. When a disk interface is fitted to the BBC Micro, a little of the computer's memory is assigned to it, one purpose being to provide buffers linking the computer to the disk drive. If you remember, these are used to compensate for the different rates at which the computer and disk drive can receive and transmit information. One of these buffers is used to carry out the various commands that have been covered so far. Five other buffers are also provided, their purpose being to link the computer to information files. Each file being used requires its own buffer, and it is therefore possible to have up to five files in use at any one time.

To keep track of things, each file being used must be allocated to a specific buffer. The buffers therefore need some sort of label, and when a new file is started the first command given amounts to 'start a new file, assign a buffer to it, and give the buffer a name'. The command used is OPENOUT, and to illustrate its use let's create a text file called "ENTRIES".

Now type

```
X=OPENOUT"ENTRIES"
```


When you press RETURN you will hear your disk drive start up. What has happened is that the computer has created on the disk a file called "ENTRIES" (check it with *CAT if you like), has assigned a buffer to it, and has given the buffer the variable name 'X'. In fact any variable name could be chosen, but single characters are the most convenient. From now on, information can be sent to the disk simply by sending it to 'X'.

But first a problem! How long is the file? After all, it will grow as you put in more entries, and you may wish at some stage to SAVE a program while still keeping the file open for further entries. How does the computer know where the file ends, so that it knows where to start SAVEing the program on the disk? The answer is that it doesn't, as long as the file is still growing. One solution might be to keep shifting all the other files on a disk as your information file grows, but this is not what the BBC Micro does. Instead, an empty file 64 sectors long is created at the start, ample room for most purposes, and when you tell the computer that the file is complete it is ended at its proper length. So, OPENOUT has not only started the file called "ENTRIES", but has already made it nominally 64 sectors long. You can confirm this easily by typing

```
*INFO ENTRIES
```

You should see that the file is 4000 bytes long in hexadecimal, which is $16 \times 16 \times 16 \times 4$ bytes in decimal. Since there are 256 bytes in a sector, and $256 = 16 \times 16$, this means that there are 16×4 , or 64 sectors in the file.

Now the file requires some information to store, so type

```
A$="ENTRY 1"
```

```
B$="ENTRY 2"
```

```
C$="ENTRY 3"
```

```
D$="ENTRY 4"
```

```
E$="ENTRY 5"
```

Next a line is required which amounts to 'send the information to the buffer allocated to 'ENTRIES' '. The command for doing this is PRINT#, so type

```
PRINT#X,A$,B$,C$,D$,E$
```

Should the buffer become full (remember, buffers hold one sector of bytes) it will empty into the file and you will hear the

disk drive start up. This will not happen in this case because we are only dealing with a few short entries.

Finally, the file must be properly closed to release the buffer for use with other files. To do this, simply type

```
CLOSE#X
```

The effect is to cancel the buffer and empty whatever remains in it on to the disk. Try *CAT if you wish to confirm the presence of the file on the disk. Also try *INFO again and note that the actual finished file is now much shorter.

Reading back from the file is just as straightforward. First a buffer must again be assigned to the task, this time using the command OPENIN, so type

```
X=OPENIN"ENTRIES"
```

Now a line is required which amounts to 'put the required entries from the file into the buffer'. The command for this is INPUT#, so type

```
INPUT#X,Y$,Z$
```

This will put the first two entries into the buffer. Next a line is required which says 'do something with these entries'. Since the easiest thing is simply to display them on the screen, type

```
PRINT Y$,Z$
```

Finally, free the buffer by closing the file with

```
CLOSE#X
```

Files such as these would normally be written or read from a program, in which case a simple loop is all that you would have to add so that the computer would keep going back in the program to get more information.

For example, a simple program to write a file would be

```
10 X=OPENOUT"FILE"
20 REPEAT
30 INPUT A$
40 IF AS="" THEN 60
50 PRINT#X,A$
60 UNTIL A$=""
70 CLOSE#X
```

If you run this program, all you have to do to close the file is to press RETURN without typing another entry in. Line 40 then does the rest.

Reading back from a file is equally simple, particularly if you use the command EOF# which detects the end of a file. For example, a typical program to read back all the entries from a file would be

```
10 X=OPENIN"FILE"  
20 REPEAT  
30 INPUT#X,Y$  
40 PRINT Y$  
50 UNTIL EOF#X  
60 CLOSE#X
```

Now a small problem! Suppose that the finished file is going to occupy more than 64 sectors. What does the computer do when it comes to the end of the 64 sectors that were originally allocated? If no other files have been added to the disk since the information file was opened, the answer is simply that the computer automatically keeps extending the file.

But what if this is not the case? Suppose that you have SAVED a program, or perhaps opened a second information file, so that the first file is now limited to 64 sectors. When this happens and the first file is full, the screen will display an error message that the file cannot be extended.

The problem can be avoided if you know in advance that the file will be longer than 64 sectors, by getting the computer to allocate more space from the start. This can be done quite simply when you remember that storing a file with the same name as one already on the disk causes the old file to be erased. Provided that the new file is no longer than the old one it then goes into the same location on the disk.

All you have to do then to create extra space is first to put on the disk a file of the same name as your proposed information file, and make sure that it is of the desired length. Then when you OPENOUT your information file the earlier file will be erased. Since the earlier file will be longer than 64 sectors, the new information file will go into the same location and you will have the full length of the old file at your disposal. To ensure that there are no other gaps on the disk to which your new file might go by mistake, you might also like to use

***COMPACT.**

How then is the 'dummy' file created? One simple way is to *SAVE a section of computer memory. Suppose, for example, that you want a file 128 sectors long. Since each sector has 256 bytes, the file will actually be 32768 bytes long. Now all you have to do is *SAVE that many bytes, and since you are not really interested in which bytes are involved, why not make things simple and start from byte 0. That way you will only have one hexadecimal number to calculate. The number 32768 in decimal is the number 8000 in hexadecimal, so in this case the command would be

```
*SAVE FILE 0000 8000
```

If hexadecimal numbers bother you, try the following short program:

```
10 X=OPENOUT"FILE"
20 FOR Y=1 TO 4096
30 PRINT#X,"SPACES"
40 NEXT
50 CLOSE#X
```

This sends the six bytes required for the word 'SPACES' to the dummy file 4096 times. But, of course, 4096×6 is only 24576, whereas we require 32768 bytes to reserve 128 sectors. In fact, the above program does reserve 128 sectors, because when each entry is put into the file it is preceded by two bytes indicating the type of entry. Thus in the above case, eight bytes are stored per entry, not six, giving us the desired 32768 bytes. Now all you need to do is to OPENOUT your real file, give it the same name, and all this space is at your disposal.

You can confirm that two extra bytes precede each entry if you like by performing *DUMP on the file ENTRIES, In which case you should get the display

```
*DUMP ENTRIES
0000 00 07 31 20 59 52 54 4E ..1 YRTN
0008 45 00 07 32 20 59 52 54 E..2 YRT
0010 4E 45 00 07 33 20 59 52 NE..3 YR
0018 54 4E 45 00 07 34 20 59 TNE..4 Y
0020 52 54 4E 45 00 07 35 20 RTNE..5
0028 59 52 54 4E 45 ** ** ** YRTNE...
```

Note first that the information is stored in reverse. If you *CAT a disk there is no indication as to which files are programs, which are information files, which are spooled files, etc. Using *DUMP in this way indicates which type of file you are dealing with.

The two bytes preceding each entry cannot be translated as ASCII code, and are therefore shown as two dots in the translation of the file. By reference to the actual bytes you should be able to see that for each entry the two extra bytes are 00 and 07. The first of these indicates that the entry is a string, and the second gives the length of the string in bytes. Remember though that these are hexadecimal numbers. If an entry had been an integer number, it would have been preceded by a single byte 40 followed by four bytes containing the number. If an entry were a real number, it would be preceded by the single byte FF followed by five bytes containing the number. Note that numbers are not stored in ASCII code and will not translate on *DUMP.

Searching files

Now suppose that you wanted to search through an information file for a particular entry. One way would be to take each entry in turn, put it into the buffer, examine it to see if it were the one required, move on to the next entry, and so on, until the desired entry was found. For example, if you had a file containing names and addresses and you wanted a printout of all those living in a particular town, you would examine each entry for the name of the town, printing or rejecting each entry in turn until the end of the file was reached. This can be done on either tape or disk.

However, disks by their nature allow random access to all parts of a file without having to go through every entry, provided only that you can tell the computer whereabouts in the file to find the information, and in this way the computer can dodge about in a file collecting entries at will.

How then do you tell the computer the place in the file where an entry is to be found? The answer is to use the command PTR# which is provided for this very purpose.

PTR# and EXT#

These commands work only with disk drives. What they do is:

PTR#

indicates the current place in a file by pointing to the actual byte, counting from the first byte as byte 0.

EXT#

indicates the length of a file in bytes.

When you OPENOUT a new file, it is empty at the start. PTR# and EXT# therefore both indicate the first byte, which the computer counts as byte 0. To confirm this type

```
A=OPENOUT"POINTER"
PRINT PTR#A,EXT#A
```

You will see that both commands indeed point to byte 0. Now type

```
A$="POINT"
PRINT#A,A$
PRINT PTR#A, EXT#A
```

You should see that the pointers have now moved to byte 7. What has happened is the the word 'POINT' has been preceded by two extra bytes as before, so that seven bytes have been used to store the entry. The computer numbers these 0 to 6, and the pointers indicate the next available byte, i.e. byte 7. Now close the file with CLOSE#A, and type

```
B=OPENIN"POINTER"
PRINT PTR#B, EXT#B
```

Note that EXT# still points to the end of the file, but because the file has just been opened for reading, PTR# points to byte 0, and is ready to move through the file byte by byte as the computer searches for entries. In effect, PTR# indicates where the computer has got to in the file.

Now, the important and useful features of PTR# is that you can set it yourself to any particular place in a file and tell the computer to go straight there. All you need to know is where every entry starts! This could be quite tricky with entries of variable length, since it is really impractical to keep a separate

record of the starting point for each entry. One way round this is simply to ensure that every entry is of the same length, perhaps by padding out shorter entries with blank spaces. The starting point for each entry can then be calculated very easily.

As an example, take the file 'ENTRIES' which you should have on your disk. Each entry occupies nine bytes, seven for the entry itself and two extra ones preceding it. You can confirm this from the *DUMP display given earlier if you wish. Suppose then that you want the computer to retrieve the third entry. The first two entries occupy 18 bytes, which the computer understands as bytes 0 to 17. So you must set the pointer to the 19th byte, i.e. byte 18. Therefore type

```
A=OPENIN"ENTRIES"
PTR#A=18
INPUT#A,ENTRY$
PRINT ENTRY$
```

You should indeed have retrieved 'ENTRY 3' by this process. Practise retrieving other entries in this way, simply setting PTR# to multiples of nine, and then, finally, don't forget to close the file properly when you have finished.

Updating and extending files

So far, you have seen how OPENOUT is used to create a file, and how OPENIN is then used to read information from it. Sooner or later, however, it will occur to you that it would be very nice to be able to alter an already existing file, either by changing the entries in it or by extending it with further entries. The command for doing this is OPENUP, and it is used in a similar way to OPENIN except that when an entry to be altered is found, PTR# will have moved past it and will now be pointing to the beginning of the next entry. Before putting in your new entry, you will have to move PTR# back to the start of the entry that you are altering.

To see it in action, try the following on the file ENTRIES:

1. Altering an entry

```
10 X=OPENUP"ENTRIES"
20 REPEAT
```

```

30 pointer=PTR#X
40 INPUT#X,A$
50 IF A$="ENTRY 3" THEN PTR#X=pointer:PRINT#X,
  "CHANGED"
60 UNTIL EOF#X
70 CLOSE#X

```

If you now use OPENIN to read the entries in the file, or for faster results use *DUMP, you will see that ENTRY 3 has indeed been replaced by the word CHANGED. Notice how the variable 'pointer' in line 30 keeps track of where each entry starts, and how it is used in line 50 to reset PTR# once the offending entry has been found.

Note also that your new entry must be the same length as the old one. If you put in a longer entry you will start to overwrite the next entry in the file as well, and will obliterate the markers separating each entry in the process. A shorter entry will alter only the end of the old one, and the marker between entries will become corrupted, i.e. changed so that the computer will not accept it. If necessary, therefore, use spaces to pad out a shorter entry until it is the right length.

2. Extending a file

```

10 X=OPENUP"ENTRIES"
20 REPEAT
30 INPUT#X,A$
40 UNTIL EOF#X
50
60 REPEAT
70 INPUT A$
80 IF A$="" THEN 100
90 PRINT#X,A$
100 UNTIL A$=""
110 CLOSE#X

```

The first part of this program finds the end of the file at line 40, then from line 60 onwards the program simply writes entries just as you have already seen with OPENOUT. Try putting in ENTRY 6 and ENTRY 7, then use OPENIN or *DUMP to confirm that they have indeed been added to the file.

Note that the last file to be stored on your disk was called `POINTER`, and unless you have created some space by deleting one or more of the old files, it will have been stored on the disk after `ENTRIES`, thereby preventing `ENTRIES` from being extended beyond filling up the last sector already allocated. Fortunately, `ENTRIES` is a short file occupying only one sector, and there is sufficient space left in this sector for several further entries. However, if you try putting in a long entry which would take the length of the file beyond 256 bytes, the computer will report that the file cannot be extended. You will then have to delete `POINTER` if you want to extend the file further.

Problems

Screen scrolling

Some commands such as *DUMP, *LIST and *TYPE may cause the screen to scroll with longer files, just as LIST does with BASIC files. Fortunately, pressing CTRL and then 'N' together will allow you to see the information page by page, just as should already be familiar to you in connection with LIST. Pressing CTRL and then 'O' together cancels the effect.

Tapes and discs

On the standard tape version of the BBC Micro the memory available for your program starts at location 3584 (or byte number 3584, if you prefer), but when a disk interface is fitted some memory is allocated to it for buffers, etc. In the case of the Acorn interface, this memory is automatically reserved when the computer is switched on, and the remainder available to you starts at location 6400. You can confirm this by asking your computer to print PAGE. The missing 2816 bytes usually cause no problem, and most tape programs will load-and-run happily on a disk-based machine. You can then normally SAVE each program on to disk so that you can run it more conveniently in the future.

Longer tape programs may however cause you one of three problems. Firstly, you may find that a program that runs on a tape-only machine will no longer run on a disk-based machine because at least some of the missing 2816 bytes are required for it to do so. The answer here is simple. Before loading the program from tape, reset PAGE to 3584. Loading will then start at this location rather than 6400, thus restoring the missing bytes, and the program will run normally. Beware though!

Pressing BREAK will cause PAGE to be reset to 6400. Also, any attempt to use your disk drive while the program is in the machine will cause the disk system to 'take over' the bytes that it needs, thus erasing part of your program.

The other two problems arise when you try to transfer long programs from tape to disk. Firstly, some programs may be so long that you will have to reset PAGE simply to make enough room to store them. When you try to SAVE them to disk, the disk system will again take over the bytes that it needs, erasing part of your program in the process. Unfortunately, there is no simple solution to this.

Secondly, you may find that a program which runs on a tape-only machine but not on a disk-based machine, is at least short enough to be SAVED to disk without the problem of resetting PAGE. In these cases a simple LOAD from tape and SAVE to disk will produce a file on disk which can at least subsequently be LOADED and LISTed. Unfortunately, loading from disk starts at memory location 6400, so the program will not run without the extra memory. The answer is simply to LOAD the program from disk as normal, then move it 2816 bytes down in memory. Unfortunately, the computer keeps track of where BASIC programs are by means of special memory locations called 'pointers', and these must also be reset so that they follow the program down in memory. To do this, PAGE must be reset and the command OLD must be given.

A typical set of commands for the purpose would be

```
FOR A%=PAGE TO TOP:?(A%-2816)=?A%:NEXT
PAGE=3584
OLD
*TAPE
```

The program will now run properly. Beware of pressing BREAK however, because it resets all pointers and you will have to start from PAGE = 3584 again. Why A% and not A? Well, 'real' variables such as A are handled in user-available memory, in the space above a BASIC program, whereas 'integer' variables are handled well below user-available memory. It's safer to use integer variables when shifting a program around in memory, and in any case integer variables are handled much more quickly. And why *TAPE? Well, since

you are operating from disk, you might just forget that you have moved the program into the working area of the disk drive, and you might use the disk drive to, say, *DUMP a file. A part of the program would be lost in the process. Issuing *TAPE inactivates the disk system until you either use *DISK or press BREAK.

Further Work with Disk Drives

As you become familiar with your disk drive through this book, you will eventually wonder if there are further facilities for disk drives which you can use. Since your disk drive is simply taking its instructions from the computer, all you have to do is write your own alternative instructions and you can get your disk drive to carry out many more functions. The list of possibilities is endless, and a description of how such instructions are written is beyond the scope of this book. Fortunately however, it is possible to purchase commercial sets of instructions either on disk or in a plug-in ROM which will carry out many additional functions for you, and in this chapter some of these possibilities are briefly described.

First however, provided that your BBC Micro is fitted with BASIC II, there is one further standard command that you can make use of.

OSCLI

OSCLI stands for 'Operating System Command Line Interpreter'. It is a special command which must be followed by a string or string variable (i.e. a word rather than a number), and what it does is to send the string to the operating system as though it were a standard command of the type that you would normally precede with an *. For example

```
OSCLI"LOAD SCREEN"
```

is the same as

```
*LOAD SCREEN
```

Try it if you like to confirm that it does indeed work. You can

even replace the string by a string variable if you wish. For example

```
A$="LOAD SCREEN":OSCLI A$
```

will do the same thing.

This may seem to be of very limited benefit, but it has a particular use in joining BASIC programs together without using *SPOOL. To see it in action first type the following:

```
PRINT PAGE,TOP
```

You should find that with your disk drive fitted PAGE has the value 6400 and TOP has the value 6402. What this means is that the start of memory for your BASIC programs is the decimal address 6400, and that the top of the current program in memory is the decimal address 6402. This makes sense if you recall that even an empty program still contains two bytes, one indicating the end of a line and the other indicating the end of the program. Now place a freshly formatted disk in your disk drive and type the following:

```
10 PRINT"THIS IS PROG1"  
SAVE"PROG1"
```

```
20 PRINT"THIS IS PROG2"  
SAVE"PROG2"
```

Suppose now that you wanted to join these programs together. Normally you would reload PROG2, then send it back to the disk with *SPOOL. You would then load PROG1 and *EXEC PROG2 to complete the join. Alternatively, of course, you could *SPOOL PROG1 and join it to PROG2 instead, and the result would be the same.

There is however another way of doing the same thing without using *SPOOL. To see it in action do the following:

```
press BREAK and LOAD PROG1
```

If now you were to load PROG2 in the normal way, PROG1 would be lost in the process. However, do you remember that the command *LOAD can be used with BASIC programs, and that they can be relocated in the process? All you have to do now is to load and relocate PROG2 so that it sits in the right place at the end of PROG1, and it will appear to the computer

to be part of PROG1. Finding the correct place is easy, because it is always two bytes below TOP just as you found above for an empty program. What you have to do then is to load PROG2 starting two bytes below TOP for PROG1, and the end-of-file marker for PROG1 will be erased by the start of PROG2. The computer will then see one continuous long program rather than two separate programs.

First of all then, type

```
PRINT ^TOP
```

Note that the hexadecimal value of TOP is required because *LOAD take hexadecimal relocation addresses, not decimal. You should obtain the value 1917 for TOP of PROG1. PROG2 must now be loaded starting two bytes lower at hexadecimal address 1915. To do this type

```
*LOAD PROG2 1915
```

If you now LIST the program, you will find that PROG1 and PROG2 have been joined together, and may be RUN as one program.

So far everything is like *SPOOL, but now for the unusual part! Press BREAK and then try loading PROG2 first and adding PROG1 to the end of it. When you LIST this, you will find that line 20 comes before line 10, and when you RUN it line 20 will be executed before line 10! The computer in fact simply takes the lines in sequence and ignores the unusual line numbering. Using RENUMBER will correct matters if it really bothers you! Notice also that if you continue stitching programs together like this, it would be quite possible to end up with a long program in which all the lines are line 10!

How then is all this related to OSCLI? Well, even subtracting 2 from hexadecimal numbers can be tricky sometimes, and OSCLI lets you avoid the problem. What it lets you do is to convert the hexadecimal value for (TOP-2) into a string using STR\$, so that it can be added to the rest of the OSCLI command line. Then

```
LOAD PROG1
```

```
OSCLI"LOAD PROG2 " + STR$^ (TOP-2)
```

will complete the join for you without you ever having to know the value of TOP.

Adding extra commands

Extra commands providing additional facilities for your disk drive are available on disk or in plug-in ROM, through either your local dealer or the specialist computer magazines. The different products vary in the facilities offered, but in general the following are typical of what can be done.

1. Direct disk-tape transfer

Normally, if you want to copy a file from tape to disk or vice versa, you must first transfer the file to the computer before sending it to the desired device. It is possible, however, to get the computer to do the whole operation for you, not only for one file at a time, but for whole groups of files using ambiguous filespecs.

2. Loading a program then moving it in memory

Do you remember the problem mentioned in the last chapter where long tape programs can sometimes only be run on a disk-based microcomputer if they are first LOADED normally then moved down 2816 bytes in memory? It is possible to get the computer to do the whole thing automatically for you with a single command.

3. Editing a disk

You have already seen how data and text files can be altered with OPENUP. However, it is also possible to directly edit a disk with a single command, in which case the disk contents are displayed one sector at a time on screen rather like *DUMP, and can be edited byte by byte on screen before being sent back to the disk.

4. Storing more than 31 files

Thirty one files may seem like a lot to store on any one disk, but you can soon get up to that number with short programs, and it can be irritating to realise that there is still a considerable amount of empty space on a disk which cannot be used. It is possible, however, to double this number of files by formatting the disk in a special way which treats it as two separate parts, each with only half the original storage capacity but each able to store 31 files. Obviously a special formatting routine is required, and there must be an extra command to allow you to switch between the two sets of files. Alternatively, it is also

possible to join several short files together to make one long file requiring only one filename, and then later to separate them by loading only a specific part of the joined file.

5. Restoring a faulty disk

After long use, parts of a disk may become unreadable, and your computer will report a disk error. Sometimes it is possible to reclaim at least some of the files on the disk by LOAD/SAVE or *BACKUP, but it is also possible to carry out a partial reformatting of only those tracks which are causing a problem. The rest of the disk, including the catalogue in sectors 0 and 1 of track 0, are untouched and the disk can then be used as before.

6. Restoring a deleted file

Inevitably, the day will arrive when you inadvertently *DELETE, *DESTROY or *WIPE a precious file and regret it afterwards. Provided that you have not stored any further files on the disk which might have gone into the space occupied by the deleted file, all is not lost! Removing the file only deletes it from the catalogue in sectors 0 and 1 of track 0, and the actual file is still in place on the disk. Commercial programs are available which will allow you to search the disk for the file, then transfer it back to the computer as a block of sectors before returning it to the disk as a new file.

Writing your own formatting routine

It is also possible for you to write your own formatting routine if you wish, and an article about how it can be done was published in the magazine *Acorn User* in September 1984. One reason for doing so is to protect your files against illegal copying. For example, it is possible to store part of a file in what the computer expects to be a gap between sectors, so that the file cannot be LOADED in the normal way, nor will *BACKUP make a copy of the file. To use the file, you must first RUN a program which is stored in a part of the disk which has been formatted normally, and this program tells the computer how to gain access to the protected file.

The list of facilities is potentially endless, and the above are

just a few of those on offer commercially at present. As you gain more experience in the use of your disk drive, you will undoubtedly begin to need at least some of these, and hopefully this chapter has given you some initial insight into the possibilities available.

Reference Section

Error Messages

Bad attribute

A letter other than 'L' has been used with *ACCESS.

Bad command

Part of a command could not be understood.

Bad directory

More than one character given with *DIR.

Bad drive

Drive number greater than three in filespec.

Bad filename

Filename has wrong syntax or too many characters.

Can't extend

An information file cannot be further extended.

Catalogue full

There are already 31 files on the disk.

Checksum error

Something is wrong with the information in an open file.

Disk changed

The disk has been changed for another while a file was open.

Disk fault AA at BB, CC

There is a fault with trying to use the disk. The fault number is AA, and the problem occurred on track BB sector CC. Some useful values of AA are

- 10 drive not recognised
- 14 could not find track 0
- 16 write fault
- 18 disk corrupted.

Disk full

There is not enough room left on the disk for the task requested.

Disk read only

A write protect tab is fitted to the disk and an attempt has been made to write to it.

Drive fault AA at BB, CC

There is a fault in the disk drive itself which was detected at track BB, sector CC. Some values of AA are given above.

File exists

An attempt was made to rename a file using an existing filename.

File locked

An attempt has been made to write to a file which has been protected by *ACCESS.

File not found

The required file is not on the disk.

File open

An attempt has been made to open an already open file.

File read only

An attempt has been made to write to a file opened with OPENIN.

Not enabled

An attempt has been made to use *BACKUP or *DESTROY without first using *ENABLE.

Command reference guide

Command	Abbreviation	Purpose
*ACCESS	*A	locks a file against most erasures, or unlocks it
*BACKUP	*BAC	makes copy of whole disk
*BUILD	*BU.	writes files which run by *EXEC or SHIFT/BREAK
*CAT	*	displays disk catalogue
CLOSE#	CLO.#	closes a text or data file
*COMPACT	*COM.	compacts files into outer tracks of disk
*COPY	*COP.	makes copy of selected files
*DELETE	*DE.	deletes a specified file
*DESTROY	*DES.	deletes a group of files
*DIR	*DI.	sets default directory
*DRIVE	*DR.	sets default drive
*DUMP	*DU.	lists file byte by byte, with attempted translation
*ENABLE	*EN.	enables *BACKUP and *DESTROY
EOF#	EOF#	detects end of file
*EXEC	*E.	runs any file created with *BUILD or *SPOOL
EXT#	EX.#	gives length of random access file
*HELP	*H.	shows extra ROMs fitted to computer
*INFO	*I.	gives file details such as length and location
INPUT#	I.#	gets information from data or text file
*LIB	*LIB	sets library
*LIST	*LIST	lists any file created with *BUILD
*LOAD	*L.	loads a machine code or BASIC file
OPENIN	OP.	opens a text or data file for reading only
OPENOUT	OPENO.	creates a new text or data file
OPENUP	OPENU.	opens a text or data file for alteration
*OPT 1	*O.1	performs *INFO when other commands are given
*OPT 4	*O.4	determines autoload result of SHIFT/BREAK
OSCLI	OS.	treats strings as though they were *commands
PRINT#	P.#	sends information to text or data files
PTR#	PT.#	specifies any point in random access file
*RENAME	*RE.	gives a file a new filename
*RUN	*R.	loads and runs a machine code file
*SAVE	*S.	saves a specified section of computer memory
*SPOOL	*SP.	saves BASIC program entirely in ASCII code
*TITLE	*TI.	gives disk a title
*TYPE	*TY.	lists any file created with *SPOOL
*WIPE	*W.	deletes group of files with final sorting

***ACCESS**

This command will protect or 'lock' a file against accidental deletion if the letter L is also given. It will not protect against formatting or *BACKUP.

Examples

*ACCESS PROG L

protects PROG in the default directory of the default drive.

*ACCESS A.PROG L

protects PROG in the A directory of the default drive.

*ACCESS :1.A.PROG L

protects PROG in the A directory of drive 1.

The command also accepts ambiguous filespecs. For example,

*ACCESS * L

protects all files in the default directory of the default drive.

*ACCESS A.*L

protects all files in the A directory of the default drive.

*ACCESS *.* L

protects all files in all directories of the default drive.

ACCESS :1..* L

protects all files in all directories of drive 1.

To unlock a protected file, simply use the same command without the final letter L.

***BACKUP**

This command is used to make an exact copy of a disk, including any gaps between files. It must first be enabled with *ENABLE.

Examples

*ENABLE

*BACKUP 0 0

will allow a copy to be made using drive 0 only. The screen will prompt the user to insert the source disk and destination disk alternately.

*ENABLE

*BACKUP 0 1

will copy automatically from drive 0 to drive 1.

Notes

Any mistake in the command will require a complete restart from *ENABLE.

This command uses a little computer memory, and may partially erase any program still in the computer.

***BUILD**

This command creates a special program file which is executed directly from disk by *EXEC, just as CHAIN executes BASIC programs.

Examples

***BUILD PROG**

creates PROG in the default directory of the default drive.

***BUILD A.PROG**

creates PROG in the A directory of the default drive.

***BUILD :1.A.PROG**

creates PROG in the A directory of drive 1.

When the command is given, the user will be offered line numbers in steps of 1, and the writing of the program ends when ESCAPE is pressed. The file is stored entirely in ASCII and can be listed with *LIST. Each line is regarded as a separate program and is executed separately from the other lines on *EXEC. Care must therefore be taken to limit any loops to single lines.

Special use

If the created file is given the filename !BOOT, it will not only be executed by *EXEC but can also be loaded-and-run simply by holding SHIFT down and pressing BREAK. Exactly what happens when this is done is determined by the command *OPT 4.

***CAT**

This command will display an alphabetical list of all files on a disk, with those in the default directory being shown separately before the rest. Also shown are the disk title, the default drive and its default directory, the library, and the 'Option' set by the command *OPT 4.

Examples

*CAT

catalogues the default drive.

*CAT 1

catalogues drive 1.

CLOSE#

This command must be given each time a text or data file is finished with. What it does is to close the buffer linking the computer to the file, and free it for use with other files. It must be followed by the name of the buffer being closed.

Example

CLOSE#X

will close buffer X.

***COMPACT**

This command is used to remove all gaps between files and put the freed space at the end of the disk. It then performs *INFO on the compacted files.

Examples

***COMPACT**

compacts the default drive.

***COMPACT 1**

compacts drive 1.

Note

The command uses some computer memory, and may partially erase any program still in the computer.

***COPY**

This command will copy one or more files on to a second disk.

Examples

***COPY 0 1 PROG**

copies PROG from the default directory of drive 0 to the same directory in drive 1.

***COPY 0 1 A.PROG**

copies PROG from the A directory of drive 0 to the A directory of drive 1.

The command will also accept an ambiguous filespec. For example,

COPY 0 1 A.

copies every file in the A directory of drive 0 to the A directory of drive 1.

COPY 0 1 *.

copies all files in all directories of drive 0 to the same directories in drive 1.

Note

This command uses a little computer memory, and may partially erase any program still in the computer.

***DELETE**

This command will delete any specified file, provided that the file has not been locked with *ACCESS.

Examples

***DELETE PROG**

deletes PROG from default directory of default drive.

***DELETE A.PROG**

deletes PROG from the A directory of the default drive.

***DELETE :1.A.PROG**

deletes PROG from the A directory of drive 1.

***DESTROY**

This command will erase one or more files as a group, provided that they have not been locked by *ACCESS. The command must first be enabled with *ENABLE, and will accept an ambiguous filespec.

Examples

*ENABLE

*DESTROY PROG

will present PROG from the default directory of the default drive and ask if you want to delete it.

*ENABLE

*DESTROY :1.A.PROG

will present PROG from the A directory of drive 1 and ask if you want to delete it.

*ENABLE

DESTROY A.

will present all unlocked files in directory A of the default drive and ask if you want to delete them all.

*ENABLE

*DESTROY *.*

will present all unlocked files on a disk and ask if you want to delete them.

***DIR**

This command sets the default directory in the default drive only. In all other drives the default directory remains directory \$. The directory must be a single character.

Example

*DIR A

sets the default directory in the default drive to be directory A.

***DRIVE**

This command sets the default drive, which must be given as a number between 0 and 3.

Example

***DRIVE 2**
sets the default drive as drive 2.

***DUMP**

This command produces a list of the contents of each byte in a file as hexadecimal numbers, together with an attempted translation of each number assuming it to be ASCII code. Where there is no corresponding ASCII character, a full stop is inserted in the translation.

Example

```
10 PRINT"7"
SAVE"PROG"
*DUMP PROG
```

will produce the following display

```
0000 0D 00 0A 09 20 F1 22 37 . . . . . "7"
000B 22 0D FF ** ** ** ** ** ** " . . . . .
```

Reading the top row from left to right, the first number 0000 indicates that the row starts at byte 0 in the file. The first byte in the file, 0D, appears at the start of every BASIC file and is used by the computer as an end-of-line marker. The next two bytes, 00 and 0A, contain the next line number which is calculated by multiplying the decimal equivalent of the first (00) by 256, and adding the result to the decimal equivalent of the second (0A). The next byte 09 indicates the length of the line in bytes, and byte 20 after this is the ASCII code for the space after the line number in the program. The next byte F1 is the BASIC code for PRINT, byte 22 is the ASCII code for inverted commas, and byte 37 is the ASCII code for the number 7. There then follows on the same line an attempted translation of each byte as ASCII code, and a full stop indicates that no translation was possible. The second line may be similarly interpreted, the byte 0D again being an end-of-line marker and the byte FF being an end-of-file marker. Double asterisks are used simply to fill up the rest of the line.

***ENABLE**

This command precedes, or 'enables', *BACKUP and *DESTROY.

EOF#

This command detects the end of a random access file, and must include the name of the buffer assigned to the file.

Example

```
10 X=OPENIN"FILE"  
20 REPEAT  
30 INPUT#X,A$  
40 UNTIL EOF#X
```

will find the end of the file currently linked to buffer X.

***EXEC**

This command will do one of two things.

1. It will load-and-run any program created with *BUILD. Unlike normal programs, each line is executed separately as though it were a separate program, so that FOR . . . NEXT and other loops must be restricted to one line when the program is written.
2. It will load into the computer any program created with *SPOOL, just as though the program were being typed in at the keyboard for the first time. Any program already in the computer is not affected, and the result is to add lines to the existing program. Thus two programs can be merged in this way, provided that line numbers do not clash, in which case the new line would replace the old.

Examples

*EXEC PROG

executes PROG from the default directory of the default drive.

*EXEC A.PROG

executes PROG from the A directory of the default drive.

*EXEC :1.A.PROG

executes PROG from the A directory of drive 1.

EXT#

This command indicates the length of a text or data file in bytes.

Example

```
X=OPENIN"-names"  
PRINT EXT#X
```

***HELP**

This command is used to obtain information about the extra ROM's fitted to a BBC Micro.

Examples

***HELP**

gives list of all extra ROM's fitted.

***HELP PROG**

gives a list of all the command words and their syntax which are associated with the ROM called PROG.

***INFO**

This command provides information on the length and location of a file on disk, together with the address at which LOADING will start and the address from which execution will start. The command will accept an ambiguous filespec.

Examples***INFO PROG**

gives information on PROG in default directory of default drive.

***INFO A.PROG**

gives information on PROG in directory A of default drive.

INFO A.

gives information on all files in directory A of default drive.

INFO *.

gives information on all files in default drive.

***INFO :1.A.PROG**

gives information on PROG in directory A of drive 1.

Typical display

```
$ .PROG      L  FF1900 FF8023 000002 002
```

Reading from left to right, this indicates that the file is PROG in the \$ directory, that it has been locked by *ACCESS, that it will be loaded back into the computer starting at location 1900 hexadecimal, that execution of the file will start at location 8023, that the file is two bytes long and that it starts at sector 2. The letters FF indicate that the file will be dealt with by the main computer processor. They would be replaced by 00 if the file were to be dealt with by a second processor purchased as an accessory.

INPUT#

This command is used to send information from a text or data file to the computer. It must be followed by the name of the buffer which is currently linked to the file.

Example

```
INPUT#X,A$
```

```
PRINT A$
```

will send an entry from a text file into the computer through buffer X, and display it on the screen.

***LIB**

This command is used to make one directory in one drive a reference section or 'library'. Normally, if a file cannot be found in a particular drive and directory, the computer will look no further but will report that the file cannot be found. The use of *LIB will cause the computer to search the 'library' if necessary before giving up.

Example

*LIB :1.A

will set the library to be directory A of drive 1.

*PROG

will now search for the machine code file called PROG in the default directory of the default drive, and if it cannot be found will then also search the A directory of drive 1 before giving up.

Note

Unfortunately, *LIB only works for machine code files.

***LIST**

This command will list any file stored entirely in ASCII code, but is designed to be used in particular with files created by *BUILD because it adds line numbers in steps of 1 to the listing.

Examples

***LIST PROG**

will list PROG in the default directory of the default drive.

***LIST A.PROG**

will list PROG in the A directory of the default drive.

***LIST :1.A.PROG**

will list PROG in the A directory of drive 1.

Note

The command will not work with BASIC or machine code files.

***LOAD**

This command will load any file into the computer, starting normally at the load address given by *INFO. However, this can be overridden if a relocation address is given with the command.

Examples

***LOAD PROG**

loads PROG from the default directory of the default drive, starting at the load address given by *INFO.

***LOAD PROG 3000**

loads PROG from the default directory of the default drive, starting at location 3000 hexadecimal.

Note

Although *LOAD will work with any file, it is normally used with machine code files. BASIC files may need BREAK to be pressed before they will run. It is also simpler to relocate BASIC files by resetting PAGE before a normal LOAD.

OPENIN

This command is used to read information from text or data files. What it does is to open a buffer connecting the file with the computer, and give the buffer a name so that it can be distinguished from other buffers.

Example

```
X=OPENIN"-names"
```

will create a buffer X linking the file NAMES with the computer, and instructions can then be given to read information from the file by first placing it in 'X' and then using it as desired.

```
INPUT#X,A$  
PRINT A$
```

will take an entry from a file, place it in buffer X, and then print it on the screen.

When the file is no longer required, the buffer must be closed with

```
CLOSE#X
```

so that it can be assigned to another file when required.

OPENOUT

This command is used to create a new text or data file and to write entries to it. What it does is to create an initially blank file 64 sectors long, open a buffer connecting it to the computer, and give the buffer a name so that it can be distinguished from other buffers.

Example

```
X=OPENOUT"-names"
```

will create an empty file called NAMES and link it to the computer through buffer X. Entries can then be made in the file by simply sending them to buffer X.

```
INPUT A$  
PRINT#X,A$
```

will place an entry in the file through buffer X. When no more entries are required, the buffer must be closed with

```
CLOSE#X
```

so that it can be assigned to another file when required. The file NAMES will then be stopped its actual length rather than the 64 sectors allocated originally.

OPENUP

This command is used to update an existing text or data file by either altering previous entries or adding new ones. What it does is to open a buffer linking the file to the computer, give the buffer a name so that it can be distinguished from other buffers, and allow both reading the file and writing to it through the buffer.

Example

```
X=OPENUP"-names"
```

will create a buffer X linking the file NAMES with the computer, and will allow information to pass both to and from the file through the buffer.

```
INPUT A$  
PRINT#X,A$
```

will put an entry in the file starting at the current position of PTR#. Previous entries can be overwritten with new ones if PTR# is positioned properly, but the new entry must be of the same length as the old one. If PTR# is set at the end of the file, i.e. EOF#X, the file can be extended with further entries provided there is room on the disk.

```
INPUT#X,A$  
PRINT A$
```

will take an entry from the file, place it in buffer X, and then print it on the screen. When the file is finished with, the buffer must be closed with

```
CLOSE#X
```

so that it can be assigned to another file when required.

***OPT 1**

This command allows *INFO to be performed on a file when most other disk commands are given.

Examples

*OPT 1 n (or *OPT 1,n)

n=1 to 255

any of these values enables the facility.

n=0

disables the facility.

***OPT 4**

This command determines what will happen if a file called !BOOT has been created in the \$ directory of drive 0, and SHIFT is held down while BREAK is pressed. The various options are

- *OPT 4 0 no action.
- *OPT 4 1 *LOADs the !BOOT file.
- *OPT 4 2 *RUNs the !BOOT file.
- *OPT 4 3 *EXECs the !BOOT file.

The current setting is shown under 'Option' in the disk catalogue.

OSCLI

OSCLI stands for 'Operating System Command Line Interpreter'. What it does is to take a string or string variable, i.e. a word not a number, and treat it as though it were an '*' command such as *DUMP. It is particularly useful for joining BASIC programs together.

Examples

1. OSCLI"DELETE NAMES"

is the same as

*DELETE NAMES

2. LOAD PROGA

then

OSCLI"LOAD PROGB " + STR\$(TOP-2)

will join PROGA and PROGB together without regard for line numbers.

PRINT#

This command is used to send information to a text or data file. It must be followed by the name of the buffer which is currently linked to the file.

Example

```
A$="ENTRY 9"
```

```
PRINT#X,A$
```

will send ENTRY 9 to a text file through buffer X.

PTR#

This command points to a particular byte in a text or data file, and tells the computer where the next piece of information is to be found. The computer can then go directly to the indicated place rather than search the file from the start.

Example

X=OPENIN"-names"

PTR#X=199

This sets the pointer to 199, so that the computer will now go directly to byte 199 to retrieve the next piece of information. Since the computer counts from byte 0, byte 199 is actually the 200th byte of the file.

***RENAME**

This command is used to give a file on disk a new name. At the same time, the file can be moved to a new directory, but not to a new drive. If the proposed new name already exists on the disk, a suitable error message will be displayed.

Examples

***RENAME PROG FILE**

takes PROG in the default directory of the default drive, and renames it FILE.

***RENAME PROG A.FILE**

takes PROG in the default directory of the default drive, renames it FILE and moves it to the A directory.

***RENAME PROG A.PROG**

takes PROG in the default directory of the default drive, and moves it to the A directory.

***RENAME :1.A.PROG :1.B.FILE**

takes PROG in the A directory of drive 1, renames it FILE, and moves it to the B directory.

***RUN**

This command will load-and-run a machine code file.

Examples

***RUN PROG**

loads-and-runs PROG from the default directory of the default drive.

***PROG**

a shortened version which does the same as *RUN PROG.

***RUN A.PROG**

loads-and-runs PROG from the A directory of the default drive.

***RUN :1.A.PROG**

loads-and-runs PROG from the A directory of drive 1.

Note

The use of *LIB provides an alternative directory and/or drive which will also be searched for PROG if the file cannot be found where specified by *RUN.

***SAVE**

This command is used to SAVE a specified block of computer memory as a file. It is the way, for example, that machine code files are SAVED. The full command is

***SAVE PROG SSSS FFFF EEEE RRRR**

SSSS

is the hexadecimal memory address where the block starts.

FFFF

is the hexadecimal memory address where the block ends, plus one.

EEEE

is the hexadecimal memory address where execution of the file begins. If omitted, it is assumed to be SSSS.

RRRR

is the hexadecimal memory address where the file is to be relocated on *RUN or *LOAD. If omitted, it is assumed to be SSSS.

It is also possible to say

***SAVE SSSS + LLLL**

where LLLL is the hexadecimal length of the block.

***SPOOL**

This command is used to merge BASIC files. What it does is to transfer a BASIC program to disk entirely in ASCII code. When the file is later executed with *EXEC, it loads back into the computer as though it were being typed in at the keyboard. It will thus add lines to any program already in memory, provided that line numbers do not clash. Where they do, the new lines will replace the old, just as would happen if a line were retyped directly from the keyboard.

The correct steps are:

1. Create the BASIC program.
2. *SPOOL PROG — this will open a text file on the disk.
3. LIST.
4. *SPOOL — this will close the text file.

***TITLE**

This command is used to give a disk a title, which may be up to twelve characters long. The disk must be in the current default drive however. Any attempt to include a drive number in the command will result in the drive number being regarded as part of the title. Also, any spaces in a title may cause problems.

Examples

***TITLE PROGS**

will give the disk in the default drive the title PROGS.

***TITLE MY PROGS**

will give the disk in the default drive the title MY.

***TITLE "MY PROGS"**

will give the disk in the default drive the title MY PROGS.

***TITLE ""**

will erase the title of a disk.

***TYPE**

This command is similar to *LIST except that it does not insert line numbers in steps of 1. It will list any file stored entirely in ASCII code, but is designed to be used in particular with those created by *SPOOL, which already have line numbers.

Examples

*TYPE PROG

will list PROG in the default directory of the default drive.

*TYPE A.PROG

will list PROG in the A directory of the default drive.

*TYPE :1.A.PROG

will list PROG in the A directory of drive 1.

Note

The command will not work with BASIC or machine code files.

***WIPE**

This command is used to delete whole groups of files like *DESTROY, but allows each file in the group to be examined just before deletion, so that a separation decision about deletion can be made for each. The command will accept an ambiguous filespec, but the files must not have been locked by *ACCESS.

Examples

***WIPE PROG**

will present PROG from the default directory of the default drive and ask if you want to delete it.

***WIPE :1.A.PROG**

will present PROG from the A directory of drive 1 and ask if you want to delete it.

WIPE A.

will present each unlocked file in turn from the A directory of the default drive and ask if you want to delete it.

WIPE *.

will present every unlocked file from the default drive in turn, and ask if you want to delete it.

Index

Abbreviated commands 44, 66
*ACCESS 15, 16, 31, 67
Acorn DFS 14
afsp 17
Ambiguous filespec 17, 25
ASCII 6, 32, 42, 87, 102
Autorunning programs 41, 69, 81, 93, 98

*BACKUP 15, 27, 68
BASIC 6, 32, 34
!BOOT 24, 42
Buffers 7, 45
*BUILD 16, 24, 32, 44, 69
Byte 5

*CAT 19, 20, 70
Catalogue 11, 19, 70
CHAIN 39, 43
CLOSE# 47, 71
Command line interpreter 58, 94
Command summary 66
*COMPACT 15, 32, 33, 72
Compressed BASIC 36
*COPY 15, 27, 73
Copying a disk 15, 16, 27, 68, 73
Cycle number 20, 27

Data file – see Random access file
Default directory 22, 23, 24, 76
Default drive 20, 21, 23, 24, 77
*DELETE 15, 25, 29, 74
Deleting files 29, 62, 74, 75, 103
Density 4
dest drv 16
Destination drive 16
*DESTROY 15, 17, 25, 29, 75
DFS 2
dir 16
*DIR 15, 22, 76

Directory 16, 20, 76
*DISK 14, 16
Disk capacity 11
Disk interface 8, 13
Disk operated system 7
Disk-tape transfer 61
Double sided disks 4
Double density 4
Double drives 3
Drive 20
*DRIVE 15, 21, 77
drv 16
Dual drives 3
*DUMP 16, 32, 49, 78

Editing disks 61
Empty programs 19
*ENABLE 15, 27, 79
EOF# 48, 80
EPROM 13
Erasing files 29, 62, 74, 75, 103
Error messages 64
*EXEC 17, 36, 43, 59, 81
EXT# 17, 51, 82
Extra commands 61

Files – see Random access files
Filespec 17
File storage 5, 11
Floppy disks 3
Formatting 2, 18, 62
fsp 17

*HELP 15, 83
Hard sectoring 4

Index hole 3
*INFO 12, 15, 32, 33, 46, 84, 92
Information file – see Random access file

- INPUT# 47, 85
- Inverted commas 25
- Joining files 36, 58, 94, 100
- L 16, 31
- Length of file 46, 48
- *LIB 15, 23, 86
- Library 23, 86
- Limit on filenames 11, 61
- *LIST 16, 32, 38, 42, 87
- *LOAD 17, 41, 88
- Locking files 16, 31
- Machine code 6, 23, 32, 38, 98
- Making room on disk 48
- Merging files 36, 58, 94, 100
- Moving files down in memory 56, 61
- Numbering drives 4, 9, 16
- Numbering sectors 12
- OPENIN 47, 89
- OPENOUT 12, 45, 90
- OPENUP 52, 91
- *OPT 1 32, 33, 92
- *OPT 4 24, 43, 93
- Option 24, 43
- OSCLI 58, 94
- Problems 55
- Protecting against erasure 3, 30, 67
- PRINT# 46, 95
- PTR# 17, 51, 96
- Random access files 45
 - opening new 45, 90
 - opening to read 47, 89
 - opening to extend or update 52, 91
 - sending information to 46, 95
 - getting information from 47, 85
- finding end of 48
- length of 17, 51, 82
- pointing to place in 17, 51, 96
- searching 50
 - closing 47, 71
- Relocating files 39, 41, 56, 61, 88, 99
- *RENAME 15, 38, 97
- Renaming a file 38, 97
- Restore faulty disk 62
- Restore deleted file 62
- ROM accessories 15, 83
- ROM-based DFS 8
- *RUN 7, 17, 39, 43, 98
- Running tape programs 55
- *SAVE 17, 38, 99
- Save block of memory 39, 99
- Save screen 39
- Scrolling 40, 55
- Sectors 2, 4
- Single density 4
- Single drives 3
- Single sided disks 4
- Soft sectoring 4
- Source drive 16
- Specifying drive and directory 24
- *SPOOL 17, 32, 36, 59, 100
- src drv 16
- Switchable drives 3
- *TAPE 14
- Tape-disk problems 55
- Tape-disk transfer 61
- Text file – see Random access file
- *TITLE 11, 16, 17, 20, 26, 101
- Titling a disk 17, 26, 101
- Tracks 2
- *TYPE 7, 16, 32, 37, 42, 102
- Viewing files byte-by-byte 32, 34, 78
- *WIPE 16, 29, 103

15 GRAPHIC GAMES FOR THE SPECTRUM

Richard G. Hurley
0 7447 0002 7

GRAPHIC ADVENTURES FOR THE SPECTRUM 48K

Richard G. Hurley
0 7447 0013 2

SPECTRUM SUPERGAMES

Richard G. Hurley
0 7447 0017 5

MAKING THE MOST OF YOUR SPECTRUM MICRO DRIVES

Richard G. Hurley
0 7447 0005 1

THE SPECTRUM OPERATING SYSTEM

Steve Kramer
0 7447 0019 1

MASTERING THE TI-99

Peter Brooks
0 7447 0008 6

ADVANCING WITH THE ELECTRON

Peter Seal
0 7447 0012 4

QUALITY PROGRAMS FOR THE ELECTRON

Simon
0 7447 0004 3

THE ATMOS BOOK OF GAMES

Wynford James
0 7447 0018 3

QL SUPERBASIC: A PROGRAMMER'S GUIDE

John Wilson
0 7447 0020 5

THE QL BOOK OF GAMES

Richard G. Hurley
0 7447 0022 1

QUALITY PROGRAMS FOR THE BBC MICRO

Simon
0 7447 0001 9

EDUCATIONAL GAMES FOR THE BBC MICRO

Ian Soutar
0 7447 0016 7

INTERFACING AND ROBOTICS ON THE BBC MICRO

Ray Bradley
0 7447 0023 X

BASIC PROGRAMMING ON THE AMSTRAD

Wynford James
0 7447 0024 8

GRAPHICS PROGRAMMING TECHNIQUES ON THE AMSTRAD CPC 464

Wynford James
0 7447 0027 2

MACHINE CODE FOR BEGINNERS ON THE AMSTRAD

Steve Kramer
0 7447 0025 6

THE COMMODORE 64 BOOK OF SOUND AND GRAPHICS

Simon
0 7447 0015 9

BASIC PROGRAMMING ON THE COMMODORE 64

Gordon Davis & Fin Fahey
0 7447 0026 4

PLUS/4 MAGIC FOR BEGINNERS

Bill Bennett
0 7447 0031 0

BBC MICRO DISK DRIVES

This book describes the use of disk drives with the BBC Micro. It begins with exploring the basic features of disk drives and the way in which a computer stores information. The reader is then taken through a step-by-step guide to his own disk drive, working through a series of examples which illustrate all the key features of the system.

A reference section is included where each command is described on a separate page and this will be of particular interest to the experienced programmer. However, the book has been written with the newcomer in mind as well providing, as it does, a concise introduction to the subject.

The Author

Robert Bagnall lives in Scotland and is involved closely with the development of computerised teaching methods at a local university.

KP-777-197

GB £ NET +006.5

ISBN 0-7447-0028-0

00695



9 780744 700282