The Advanced Disk User Guide

for the BBC Microcomputer

Colin Pharo B.Sc, MBCS

published by The Cambridge Microcomputer Centre

Published in the United Kingdom by: The
Cambridge Microcomputer Centre, 153-154
East Road, Cambridge,
England

Telephone (0223) 355404
Tlx 817445
ISBN 0 946827 70 2

The author would like to thank his wife, jenny, for the long sessions at proof-reading and the endless cups of coffee which made this book possible. Thanks also to Dan Nanayakkara and his Wife, Java, and Peter Wederell for their assistance in the production ot this book.

The combined 40/80 track technique was first described by D.M. Barnett in 'Acorn User', February 1984.

This book was computer typeset by Compulcrset (MFK) Ltd, lily, Cambridgeshire and printed by The Burlington Press (Cambridge) Ltd, Foxton, Cambridge.

Contents

Introduction

Of all the peripheral devices that you can attach to a BBC Microcomputer, probably none enhances its use and scope more than the disk drive. With fast access to both programs and data alike, you can at last put the microcomputer to work - or at least that is how it should be.

In reality, there is a dearth of really detailed information available for BBC disk systems and this tends to limit the use to which they are put. This book is just one in a series published by the Cambridge Microcomputer Centre which attempts to supply missing information for the BBC Microcomputer. The series currently comprises:

The Advanced User Guide'

The Advanced BASIC Rom User Guide'

The Advanced Disk User Guide'

It is intended that the book be read from start to finish as concepts introduced in one chapter are expanded upon in subsequent chapters. It is a good idea to skim through the book to get the general drift, followed by a more serious read afterwards. Once the fundamental principles have been grasped the user should be in a strong position to develop programs and to solve problems, aided by the several programs published within this book. All of the programs published in the book contain REM statements. Since none of the programs use GOSUB, GOTO or RESTORE the line numbers in the programs have no significance and comments need not be entered at the keyboard.

Most of this book is written in the style of an instructional textbook. In places, however, the information is presented more in the style of a manual and it is hoped that, by this means, the book will serve a dual purpose. In places (such as the Chapter on OSWORD), this has led to some repetition. It is hoped that by repeating information, the user is spared the burden of looking up information at several different places in the book.

PART I
ACORN DFS - THEORY

1 The Diskette

Various names are commonly used in place of the term, 'diskette'. You may hear them simply called 'disks', 'flexible disks', 'floppy disks' or even just 'floppies'.

They come in various sizes (3 inch, 3.25 inch, 3.5 inch, 5.25 inch and 8 inch) and in a bewildering array of formats. Many of these permutations can be used with the BBC Microcomputer. However, this chapter discusses only the standard BBC disk system. Other systems, which will be covered later in the book, may differ in detail but are based on the same principles.

Figure 1 shows a diagram of a standard 5.25 inch diskette. It consists of a thin, mylar, circular disk with a diameter of 5.25 inches, looking rather like a small gramophone record. It is coated in a material sensitive to magnetic fields and in this respect it more closely resembles the cassette tape. This magnetic coating is fragile. It can easily be destroyed by dirt and dust, finger-prints, smoke particles and by wear and tear. For this reason the diskette is protected inside a square, cardboard envelope. At one place, the read/write window, the surface of the diskette is exposed giving access to recorded data. In the diagram, the disk itself is drawn as a dotted circle.

The recording surface of a diskette is divided into a number of tracks just like a gramophone record. There is an important difference, however. The track on a gramophone record is really one, long, continuous, spiral groove from the outside to the centre. The tracks on a diskette are equally-spaced, concentric circles and are not physically cut into the diskette. These tracks do not cover the whole surface of the disk. A track near the centre of the disk would be very short and not capable of storing much data, whilst the outside edge of a diskette is prone to warp and is unreliable for the storage of data. The actual recording surface is confined to a band starting about 1.5 inches from the centre and finishing about 2.5 inches from the centre. This band coincides with, but is a little less wide than, the read/write window. When the diskette is mounted in a disk drive, it is gripped at the hub and spun within its cardboard envelope. In this way, each part of the band falls beneath the read/write window once per revolution of the disk. Thus each part of the band is accessible. The better makes of diskette incorporate an oxide lubricant in the recording surface to reduce friction between the disk and the envelope. Some diskettes are also equipped with plastic, re-inforcing rings at the hub to help withstand the forces applied by the spindle and to prolong diskette life. Plastic hub re-inforcers can be bought separately.

At the edge of the square diskette envelope, there is a write-protect notch. On some disk drives, the absence of a notch trips a switch which then sets the drive to write-protected mode. Other drives employ a light source/detector system. Within the disk drive a small LED (light emitting diode) shines through this notch and the light strikes a photo-detector on the other side. So long as this notch is present, the photo-detector receives light and signals the disk controller that the diskette is unprotected. To protect the diskette from being accidentally over-written, we simply cover this notch (with opaque, sticky paper, for example). Now when the diskette is mounted in the drive, light from the LED cannot reach the photo-detector. The signal to the disk controller drops and the controller traps all attempts at writing to the diskette. This is a useful way of protecting valuable programs/ data once they have been recorded on the diskette. Some commercial software diskettes are packaged in an envelope which has no notch. This of course gives automatic write-protection and prevents accidental erasure.

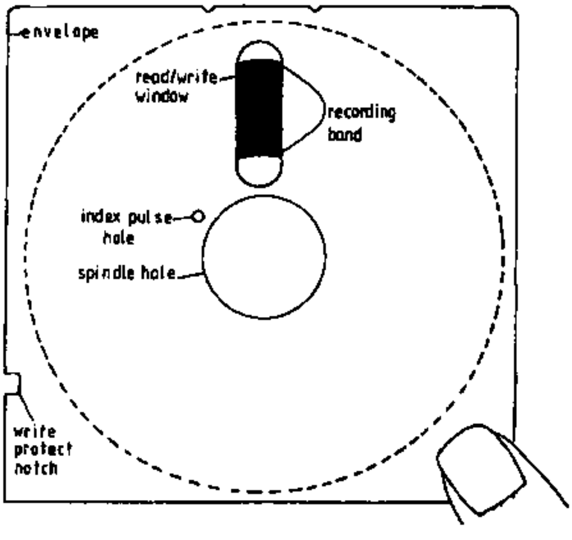


Figure 1. The Diskette.

A similar light-source/detector system is employed at the index pulse hole. This hole identifies the start of each track. The hole occurs on both sides of the envelope with a matching hole in the disk, itself. Every complete revolution of the disk, the three holes align perfectly, allowing light to strike the photo-detector which then signals the disk controller. The signal is known as the 'index pulse'. At the precise instant of this pulse, every track will be positioned at its start under the read/write window.

1.1 The Physical Track

We have already mentioned 'tracks' and described them as equally-spaced, concentric rings within a one inch recording band. Disk drives that can be used with the BBC Microcomputer normally operate in one of two formats. In one of these formats there are 80 such tracks on the diskette. The other has 40 tracks. Clearly 80 track diskettes have their tracks closer together than do 40 track diskettes. In both formats, the tracks are physically numbered from zero at the outermost track. Numbers then step sequentially towards the innermost track which is physical track 79 (80 track format) or physical track 39 (40 track format). Note especially that in most modern disk drives, the width of a track is identical in both formats. It is only the separation distance between tracks that varies (the 80 track separation is half that of the 40 track). This has important consequences as will be seen later. For the moment I just ask you to study Figure 2 and observe the following:

> Track 0 is common to both formats and is always accessible.
>
> Every track of a 40 track diskette is available to an 80 track drive. It is just that alternate tracks expected by an 80 track drive are missing.
>
> Every other track of an 80 track diskette is accessible to a 40 track drive, but odd-numbered tracks are not.
>
> Track 0 is longer than the innermost track. It is about 16 inches long, whilst the innermost track is about 9 inches long.

For clarity, only a few tracks are drawn in the diagram and these are drawn very much further apart than they really are.

It should be emphasised that some early 40 track drives (such as the Olivetti FDS01) used full width heads, such that data recorded on the surface of the diskette actually occupied two 80 track widths (i.e. the track was twice as wide). Once the 80 track head became established, it was incorporated into 40 track drives as well, since it is no less reliable than the full width head.

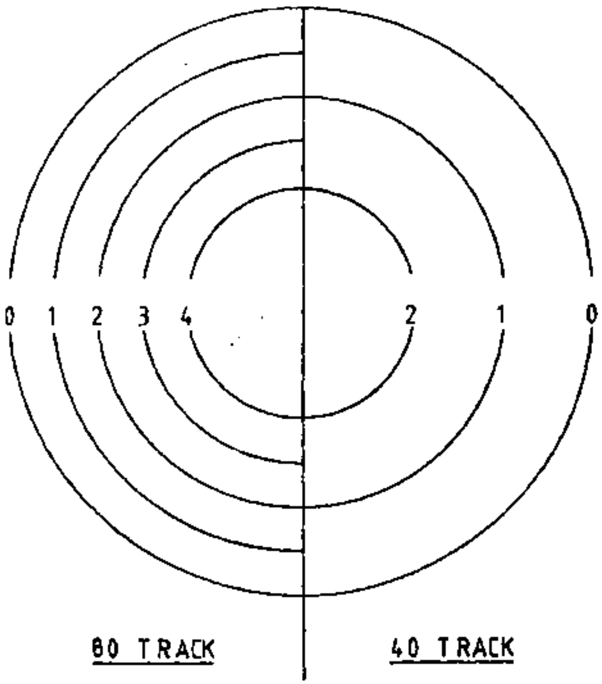


Figure 2. Physical Tracks.

Some microcomputers, such as the ACT Sirius and the Apple Macintosh, make use of the greater length of the outer tracks to store extra data. Most microcomputers, including the BBC Microcomputer, do not do this. They store exactly the same amount of data on each track, as this simplifies the drive electronics. The Acorn DFS stores 2560 bytes (characters) of data on each and every track. Thus an 80 track diskette can hold 204,800 bytes on one side. This is commonly abbreviated to 200K (where K represents 1024 bytes). Clearly a 40 track diskette can hold half this amount of data, 102,400 bytes or 100K. In addition, some diskettes have recording surfaces on both sides, thereby doubling their capacity.

The manufacturers of diskettes set out to make top-of-the-range diskettes. They subject each diskette to rigorous tests and on the basis of how well it fares in the tests, they then decide whether to sell the diskette as 80 or 40 track: double-sided or single-sided. It is most unwise to buy a 40 track diskette for 80 track use, or a single-sided diskette for double-sided use. You may get away with it for a while, but you are asking for trouble.

1.2 Soft Sectoring

To make the processing of this data more convenient, each track is subdivided into 10 equal sectors as depicted in Figure 3. Some computers use diskettes which have a ring of holes, each similar to the index pulse hole, to mark the start of each sector. This is known as hard sectoring since the sector starts are identified by hardware means. The Acorn DFS does not use this technique. Special identifiers are recorded onto the disk at the start of each sector. These identifiers are recorded there by a purpose-built program called a 'formatter' and the technique is referred to as soft sectoring, since the sector starts are placed on the diskette by software. The sectors are physically numbered from 0 to 9 and each sector holds 256 bytes of data.

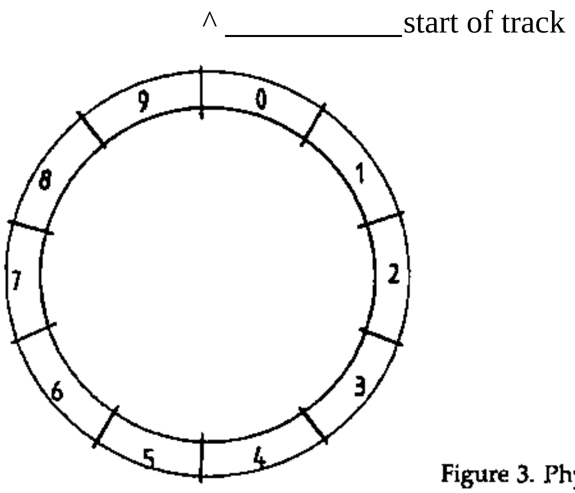


Figure 3. Physical Sectors.

The hardware components that make up the BBC Microcomputer disk system can accommodate a wide variety of formats in terms of tracks per side, sectors per track and data-bytes per sector. Here we are only concerned with the Acorn DFS which stipulates 10 sectors per track and 256 bytes of data per sector. In fact each sector is a bit more complicated than this as we shall now see. Figure 4 shows a whole sector of data.

```
---------------------
 sync bytes               I F
---------------------     D I
 sector id mark           E E
---------------------     N L
 logical track no.        T D
---------------------     I
 head no.                 F
---------------------     I
 logical sector no.       C
---------------------     A
 data size code           T
---------------------     I
 sector id CRC            O
---------------------     N

gap2

---------------------
 sync bytes               D F
---------------------     A I
 data mark                T E
---------------------     A L
 256 bytes of data          D
---------------------
 data CRC
---------------------

gap3
```

Figure 4. The Sector.

Each sector starts with six 'sync bytes', which are bytes of &00. A sync byte is the first example of several different 'special marks' that are recorded onto the diskette. Sync bytes synchronise the controller to the rotational speed of the diskette.

This is followed by another, special mark, the 'sector identification mark', which identifies the start of the sector.

Next comes the logical track number. For most purposes the logical track number will be identical to the physical track number since most formatter programs supplied with disk drives ensure this. The physical track number identifies the position of the track in relation to the outside track. The logical track number, which is actually recorded at the start of each sector on the diskette, can be any single-byte number that we choose. However, if we choose to make it differ from the physical track number, we may impose upon ourselves the burden of coding a special read routine. This ability to deal with both physical and logical tracks is nevertheless most useful. It will be seen later that it enables us to devise a combined 40/80 track format. Moreover, it is exactly the sort of jiggery-pokery that appeals to Software Houses in their endless search for software protection.

Next comes the head number. The term, 'head', will be explained later. For the moment, we shall simply say that this number is either 0 to identify the top surface of a diskette, or 1 to identify the bottom surface. Clearly it will always be zero for a single-sided diskette.

Next comes the logical sector number. The physical sector number identifies the position of the sector with respect to the start of track. The first physical sector on the track is always zero and the last is always 9. Each sector can, however, be given any single-byte, logical sector number that we wish, though again the choice can make the diskette more difficult to read. One particular advantage of logical sectors is that it enables us to stagger sector numbers from one track to the next. It takes a certain amount of time to move from one track to the next. If we get the stagger just right, in the time that elapses between tracks, the disk will rotate to logical sector 0. The track can now be read from its logical beginning, without waiting for logical sector 0 to appear under the read/write window. This technique is employed in many of the available formatter programs. Naturally, if anybody is going to exploit logical sectoring to the point of absurdity, it is going to be the Software Houses.

Next comes the size of data to be found in the sector (a code denoting the number of bytes of data).

This is then followed by a two byte CRC (cyclic redundancy check). The CRC is a number derived from the previous bytes which serves as a check that those bytes have been recorded properly. See pages 347-348 of The Advanced User Guide for a detailed description of the algorithm used. The items described so far are called the 'Identification Field', since they serve to identify the sector.

After the Identification Field there is a gap, called 'gap 2', which consists of a number of bytes of &FF.

Then comes a series of items which are collectively known as the 'Data Field'. The Data Field starts with six sync bytes, just like the Identification Field. This is followed by another, special mark, called the 'data mark' which identifies the start of data. This is followed by the data itself, which is 256 bytes long*. The Data Field is terminated by its own 2 byte CRC.

Another gap, called 'gap 3', separates each sector from one another.

In Figure 5 we see a complete track. Essentially this is just 10 sectors repeated one after another. To avoid cluttering up the diagram, sectors 2 to 7 have not been completely entered. Note that each track starts with a special mark called the 'Index Mark'. Note also the variety of gaps to be found in each track.

gap 1 occurs once per track between the Index Mark and physical sector 0.

gap 2 occurs 10 times per track between each Sector Identification Field and Data Field.

gap 3 occurs 10 times per track following each Data Field.

gap 4 occurs once per track at the end of physical sector 9.

gap 5 pads out the rest of the track from gap 4 up to the Index Mark.

For the Acorn DFS, which specifies ten sectors per track, the numbers of &FF bytes in each type of gap are:

gap 1 = 16 bytes of &FF

gap 2 = 11 bytes of &FF

gap 3 = 21 bytes of &FF

gap 4 = 30 bytes of &FF

gap 5 = 0 bytes of &FF

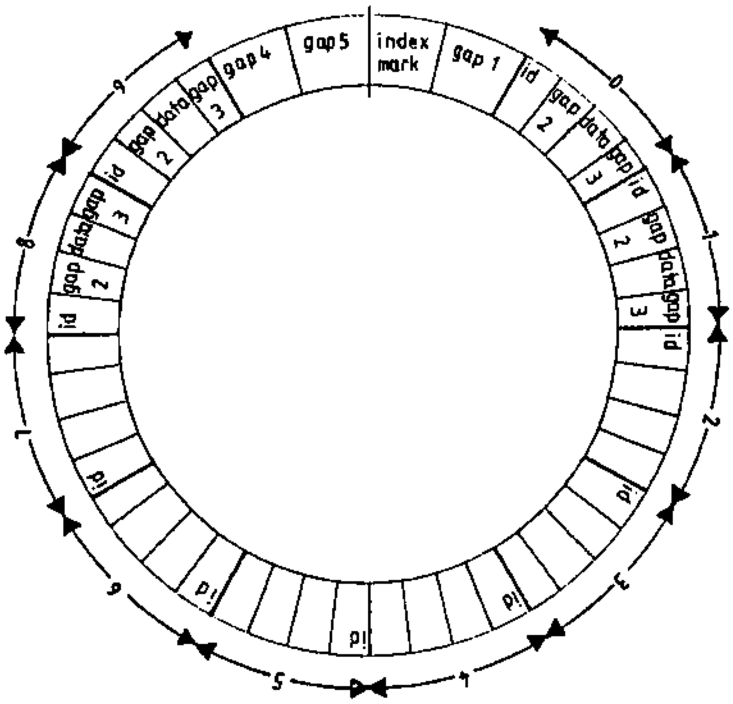


Figure 5. A Complete Track.

1.3 FM Single-Density Recording Mode

The system to be described here is the one which is employed by the Intel 8271 disk controller, the Acorn DFS standard. The method of recording data is known as 'Frequency Modulation' or FM for short.

Data is recorded onto, or read from, the recording surface of the diskette by a part of the disk drive, known as the read/ write head. This has the ability to convert an electrical pulse into a magnetic field and vice-versa. Each bit of data is stored serially as a magnetic field on the recording surface of the disk. The read/write head converts magnetism to electricity when reading and vice-versa when writing. But how does the system identify the start of each bit? If the 8271 had to read a long string of zero bits, it would require a precise spin-rate of the diskette and precise internal clocks. In practice, this would be prohibitively expensive. The answer is that immediately before each bit of data, a special '1' pulse is recorded on the surface of the disk. This is called a 'clock-bit' and it synchronises the read operation. Figure 6 shows a typical 8-bit byte of data which just happens to have a value of &55. Each bit is preceded by a clock bit. It is convenient to think of the clock-bit pattern as &FF. When this is interleaved with the data-bits, the result is a bit pattern of &BBBB. Clock bits are marked 'C and data bits 'D'. Each bit has a total period of 2 microseconds.

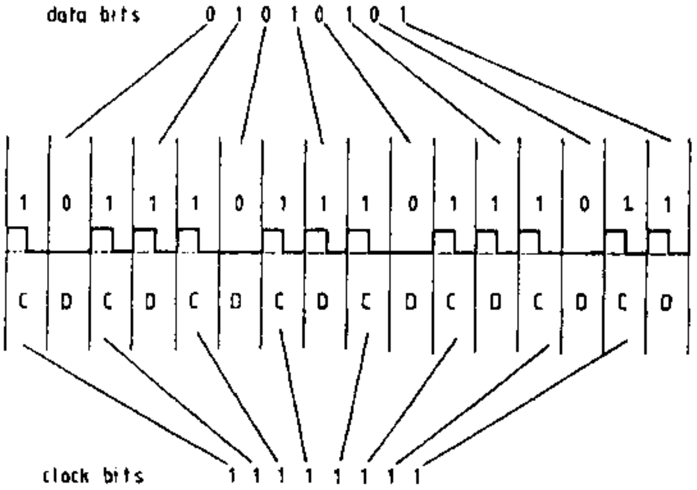


Figure 6. Single-Density Data Byte.

All of the special marks on a track are made up of combinations of specified patterns of clock-bits and data-bits. They are tabulated in Figure 7. In each case, the clock-bits are interleaved with the data-bits, just as they were for a data byte.

Type of Mark	Clock Bits	Data Bits
Sync byte	&FF	&00
Sector id Mark	&C7	&FE
Data Mark	&C7	&FB
Deleted Data Mark	&C7	&F8
Index Mark	&D7	&FC

Figure 7. Standard Disk Marks.

One of these marks we have yet to dicuss. The Deleted Data Mark replaces the Data Mark when data is deleted. It will be seen later that the 8271 has a set of commands specifically directed towards deleted data.

1.4 Care of Diskettes

Diskettes are easily damaged but if you obey the following rules you should save yourself much inconvenience.

Never touch the recording surface of a diskette.

Only handle them at the corners as shown in Figure 1.

After use, always replace diskettes in their antistatic wrappers. They should be placed in these wrappers window-first so that no part of the window is exposed.

It is advisable to keep diskettes in a special storage box, of which there are many on the market. This box should have the lid closed to give further protection from dirt and dust, not to mention coffee spills etc.

Keep diskettes away from strong magnetic fields which can corrupt data. Likely sources of magnetic fields within the home are televisions/monitors, loudspeakers and telephones.

Do not fold, bend or roll a diskette.

Keep diskettes away from heat and sunlight which can warp them.

Do not write on diskettes with a biro or other sharp object. It is best to write the label first and then stick it to the diskette. If you have to write on a diskette, use only a felt tip pen and even then apply the least possible pressure.

Do not use 40 track diskettes as 80 track diskettes, or single-sided diskettes as double-sided. They will probably fail sooner or later.

Do use hub reinforcers.

Use a proprietary cleaning disk and fluid regularly. This removes dirt from the head and prolongs the life of both the head and the diskette.

Do not switch power from the drive whilst it is spinning, or whilst the head is moving.

Buy good quality diskettes.

Never force a diskette into the drive. If it does not slide in easily, remove it and try again.

2 The Disk Drive

We already know quite a lot about disk drives. We know about the write-protect facility and also about the index pulse and we have mentioned the read/write head in passing. There are a number of other parts that we need to know about.

The first of these is the drive motor. This consists of a motor which rotates a spindle. The spindle grips the spindle hole at the centre of the diskette and by this means the disk is rotated at speeds normally around 300 r.p.m.

Located over the read/write window is the read/write head of the disk drive. It is this device which is capable of magnetising (writing to) and sensing magnetism on (reading from) the diskette's magnetic surface. The head must be positioned precisely over the track to be read qr written. It must also be capable of being moved up and down the read/write window to any track position. To achieve this, the head is attached to a stepper motor. A stepper motor is a motor which can rotate a spindle through some fixed angle, commonly 7.5 degrees, whenever it is instructed to do so. This rotational movement is converted to lateral motion by worm gear which moves the head exactly one track, each time the stepper motor is pulsed. Thus the stepper motor serves to position the head exactly over any track that we specify. Clearly, in an 80 track disk drive, the lateral movement achieved by a single step will be half that of a 40 track drive. You have probably heard the stepper motor in action. Single steps click whilst the rapid steps that can occur at the end of a SAVE, when the head is stepped back to track 0, tend to purr. The time that the stepper motor takes to move the head by one track is known as the 'step time' and is the first of several important parameters that we shall encounter.

The rotation of the diskette under the head then makes all the data on that track accessible to the head. Some disk drives can access double-sided diskettes. These drives have only one drive motor, since only one motor is needed to spin the diskette. However, they have two heads each driven by an independent stepper motor. The heads are located either side of the diskette.

Thirdly, there is a special detector for identifying when the head is located over physical track 0. Whenever the DFS is uncertain about the position of the head, it is likely to instruct the controller to step the head outwards until physical track 0 is located, and then step the head inwards again to the required track. The track 0 detector is usually a microswitch which is tripped by the head/stepper mechanism as the head moves over track 0.

Lastly, there is a device known as the load actuator. For the head to perform a read or write operation it needs to be in physical contact with the surface of the diskette. If the head was permanently in this position it would cause a lot of unnecessary wear to both the head and the diskette, aggravated by dust particles sticking to the head itself. Thus the surface of the diskette is only brought into contact with the head just before a read/write operation and only for the duration of that operation. The device which achieves this is called a load actuator and frequently consists of a felt pad which pushes the diskette towards the head. The time taken to load the head is another important parameter and is known as the 'head load time'. On many disk drives, the load actuator is the noisiest component, often making a loud clunk when activated. When the load actuator is activated, the head tends to bounce a little. It takes a little while to settle. This is referred to as 'settlement time' and is our third important parameter. The final parameter of interest is the time taken to unload the head after a disk input/output. This is expressed in terms of the number of revolutions of the disk that occur at the end of the input/output before the unload, and normally has a value of 12.

The BBC Microcomputer has a set of eight switches, located inside the casing on the keyboard immediately below the DELETE key. These switches determine various initialisation options at power up, such as the graphics mode to be used. On older models of the BBC Microcomputer, the switches are implemented in a DIL package. More recent models use north/south links. A pair of these switches (3 and 4) determine the disk drive timings at power up. In Figure 8, a 0 indicates that the switch is closed (i.e. the north/south link is present) whilst a 1 indicates an open switch. Most BBC Microcomputers are delivered with switches 3 and 4 open, corresponding to the slowest disk access times. Many of the more modern disk drives are capable of much quicker performance. Some specify these parameters in their operating manuals, allowing you to configure your microcomputer exactly. For the others, it is worth experimenting to see what is best for your system. All times are given in milliseconds. Optimising these parameters can significantly improve disk performance. They can also make the difference between having a disk system that works and one that does not.

Link 3	link 4	step time	settle-ment time	head load time
0	0	4	16	0
0	1	6	16	0
1	0	6	50	32
1	1	24	20	64

Figure 8. Disk Access Timings.

Fortunately, the settings can also be changed by software enabling you to experiment more easily. To test the impact of changes to these settings, set up a diskette with some data that you do not mind losing. You can experiment with various settings using OSBYTE, as described in Figure 9. The settings that you specify via OSBYTE do not take effect until a soft BREAK and will remain in force until a hard BREAK. To send these settings to the disk drive, you can simply enter *DISK. When you are sure that you have found the settings which are best for your disk system, you can set the links permanently. Remember to disconnect from the power source before you tinker inside your computer. If the innards of your computer terrify you, I am sure that most dealers would undertake it happily in return for a very small charge.

link-3	link-4	OSBYTE call
0	0	*FX255,0,207
0	1	*FX255,0,223
1	0	*FX255,0,239
1	1	*FX255,0,255

Figure 9. *FX calls.

We are jumping ahead a bit, but you should know that you can experiment with other settings by invoking OSWORD, which also allows you to change the head unload time. This may be necessary if you are trying to use a particularly esoteric species of disk drive. Alas, you will not be able to translate these settings into hardware links.

A wide choice of disk drives exists for the BBC Microcomputer. Essentially a disk drive can be:

> 40 track or 80 track. This specifies the amount by which the head will move each time the stepper motor is pulsed. Some disk drives are switchable, allowing you to choose either 40 track or 80 track format. This is a very useful facility as it enables you to work in 80 track format yourself, but also allows you to buy software that may only be available on 40 track diskettes.
>
> Single-sided or double-sided. This specifies the number of read/write heads in the drive.
>
> Single or dual drives. Some manufacturers sell two drives in a single box. If they are double-sided they can access four diskette surfaces.
>
> With or without independent power supply.

Clearly you get what you pay for. Ignoring the matter of independent power supply and 40/80 track switching, each of which increases the cost, Figure 10 tabulates the options in order of increasing expense.

No- of drives	No. of heads	tracks	capacity
1	1	40	100K
1	2	40	200K
1	1	80	200K
1	2	80	400K
2	2	40	200K
2	2	80	400K
2	4	40	400K
2	4	80	800K

Figure 10. Disk Configurations.

Figure 11 shows the most common permutations of disk drives, together with the *DRIVE parameters used to address each surface. For some people, these parameters are a source of confusion. You should regard the *DRIVE parameter as a 2 bit field. The most significant bit is the head-number and the least significant bit is the drive-number. Thus:

0 = head 0 drive 0
1 = head 0 drive 1
2 = head 1 drive 0
3 = head 1 drive 1

Figure 11. "DRIVE Parameters.

Sadly, the computer industry has managed to introduce some bewilderment into the disk scene by using the terms single-density and double-density to convey quite different meanings.

For some suppliers, single-density means 40 track and double-density means 80 track. This is really the track density.

For other suppliers it refers to the modulation technique used to record data. The standard technique for the BBC Microcomputer is FM (Frequency Modulation). A disk system based on the 8272 or other double-density controller uses MFM (Modified Frequency Modulation). This technique reduces the number of clock bits required but it requires higher quality diskettes to be safe.

Still other manufacturers use terms like 'quad-density' to convey a mixture of these two ideas.

Alas the only way around these vague terms is to avoid them altogether. To be absolutely sure that you are purchasing the correct diskette, you need to ascertain:

> the number of sides (1 or 2)
>
> the density, specified in BPI (bits per inch). This should be approaching 5,000 BPI for standard BBC disk systems and approaching 10,000 BPI if you have a double-density controller.
>
> the recording method. FM for single-density controllers. MFM for double-density controllers.
>
> the track density, specified in terms of TPI (tracks per inch). It needs to be about 48 TPI for a 40 track system and 96 TPI for an 80 track system.

As a piece of advice this turns out to be rather unhelpful, since few suppliers quote these unambiguous statistics. The best advice I can offer is to tell your dealer exactly what hardware you own, including the fact that you have a BBC Microcomputer. Then if he sells you the wrong diskettes, the fault is clearly his.

3 The 8271 FDC

The floppy disk controller used in the standard BBC disk system is the Intel 8271. The 8271 controls data transfer between the processor and the disk drive, in both directions. This highly complex device is almost a microprocessor in its own right. It has its own instruction set with which it accomplishes a wide variety of functions, though for some of these it is assisted by much simpler disk interface chips.

3,1 8271 signals

Figure 12 shows the signals in and out of each pin of the 8271. It is worth covering these signals, at least superficially, because they give insight into how the 8271 communicates. A complete description of these signals would include timing diagrams etc., but that is outside the scope of this book.

Some of the signal names are prefaced by 'not'. This simply means that these signals are true at low voltage (logic level 0) rather than at high voltage (logic level 1). You may see elsewhere such signals shown with a line above them. This is simply a different convention for expressing the same idea.

The 8271 is powered by a 5 volt source at pin 40, designated Vcc. It also has a ground or earth reference at pin 20, designated GND.

Each other signal has an accompanying arrow which indicates the direction of the signal i.e. whether the signal is supplied to or provided by the 8271, or both. The 8271 is supplied with a clock signal at pin 3 and can be reset at pin 4.

Let us look first at the way that the 8271 reads data from the disk drive. Data exchanged between the 8271 and the drive is in bit-serial form with the added complication of interleaved clock-bits, as we have already seen. The first signal to consider is the PLO/SS signal at pin 25. This signal determines the method that the 8271 will use to separate clock-bits from data-bits, when reading data from the disk drive. One method is the phase-locked oscillator (PLO). A cheaper alternative, which the BBC Microcomputer actually uses, is the single-shot (SS). Don't worry if you do not understand the terms PLO and SS. Such knowledge is not a pre-requisite to understanding this book. The inquisitive may find descriptions of these circuits in any good book on Electronics. When reading data from the disk drive, the 8271 is given the entire bit stream of unseparated data (UNSEP DATA) at pin 27. The disk interface chips also sample UNSEP DATA and provide an extra signal at pin 26, called the DATA WINDOW, to help the 8271 separate the clock-bits from the data-bits. The 8271 uses UNSEP DATA simply to determine when to sample DATA WINDOW, and the signal on DATA WINDOW tells it whether the data bit was 0 or 1. With this assistance the 8271 is able to separate out the data-bits. In a PLO system the IN SYNC signal at pin 23 would be used as a feedback into the phase-locked loop, but this signal has no use in the BBC Microcomputer.

FLT RESET/OPO	<–	1	40	.–	Vcc
SELECT0	<–	2	39	–>	LOW CURRENT
CLK	~>	3	38	–>	LOAD HEAD
RESET	–>	4	37	–>	DIRECTION
not READY1	–>	5	36	–>	SEEK/STEP
SELECT1	<–	6	35	–>	WR ENABLE
not DACK	–>	7	34	<–	not INDEX
not DRQ	<–	8	33	<~	not WR PROTECT
not RD	–>	9	32	<~	not READYO
not WR	–>	10	31	<–	not TRKO
INT	<–	11	30	<–	not C0UNT/0P1
DO	<->	12	29	–>	WR DATA
D1	<->	13	28	<–	FAULT
D2	<->	14	27	<–	not UNSEP DATA
D3	<->	15	26	<–	not DATA WINDOW
D4	<->	16	25	<–	PLO/SS
D5	<->	17	24	<–	CS
D6	<->	18	23	–>	IN SYNC
D7	<->	19	22	<–	A1
GND	-	20	21	<--	A0

Figure 12. 8271 FDC.

Let us now look at writing to the disk drive. The first signal of interest is at pin 33, designated WR PROTECT. As the name suggests, this signal emanates from the disk drive and reflects the status of the write-protect notch on the diskette. The 8271 uses this signal to avoid writing data to a write-protected diskette. At pin 28, a signal designated FAULT is used by the disk drive to tell the 8271 that it has encountered a condition which could lead to loss of data integrity. The 8271 will abort any write attempt if this happens. It can reset the fault by signalling at pin 1, designated FLT RESET/OPO. If all is well, the 8271 notifies the disk drive that it intends to write data, by asserting WR ENABLE at pin 35. The interleaved clock-bits and data-bits are then supplied at pin 29, designated WR DATA. If the track to be written on is greater than 43, the 8271 also sends the LOW CURRENT signal at pin 39, which causes the disk drive to reduce the current used to magnetise the surface of the diskette. This is because data on the inner tracks is packed more closely together. If the normal high current was used, the magnetic fields would be sufficiently strong for each bit of data to blur with its neighbour. This would, of course, render the data useless.

Of course, we know that read and write operations to the disk drive are rather more complicated than this. The 8271 must choose the disk drive and head, position the head on the correct track and load the head onto the diskette.

Drive and head selection is achieved by the SELECT1 and SELECTO signals at pins 6 and 2 respectively. These two signals collectively constitute a two-bit signal which reflects the drive number (in *DRIVE parameter format). As we shall see later, the 8271 actually uses the two most significant bits of its Command Register to determine the voltages on SELECT1 and SELECTO. In practice, OSWORD uses the drive number to generate these two bits in the command field and in this sense the drive number can be regarded as the source of these two signals. Throughout all operations, the 8271 monitors the availablity of the appropriate drive, either at pin 0 (READYO) or at pin 5 (READY1). It also monitors TRKO at pin 31 to determine whether or not the head is at track 0, and the Index Pulse at pin 34 to determine whether or not the head is at start of track. To move the head, the 8271 sends two signals to the disk drive. The first, named DIRECTION, at pin 37, tells the drive whether to move the head towards the centre or towards the outside. For each track to be moved, the 8271 sends a STEP pulse at pin 36, allowing the stepper motor time to accomplish each step before sending the next. Finally, when the head is on the correct track, the 8271 brings the head into physical contact with the diskette by signalling at pin 38, designated LOAD HEAD. The 8271 has an alternative scheme for moving the head, whereby the drive provides its own step pulses and signals the 8271 with a pulse for each step at pin 30, designated COUNT/OP1. In this event, the 8271 holds the STEP signal high until it sees the correct COUNT. This scheme can be configured by switch 7 on the BBC Microcomputer, should your drive work that way.

Next we have to consider the way in which the 8271 interfaces with the processor. Data is transferred to and from the processor in 8-bit parallel form via the data bus (pins 12 to 19). However, it is a little more complicated than this. Firstly the processor signals the 8271 to read or write data via pins 9 or 10, designated RD or WD. How does the processor know when to inspect the data bus for data? The 8271 has two different schemes when sending data back to the 6502. The first is called DMA (Direct Memory Access) in which the 8271 sends data directly to specified memory locations. DMA requires that the microcomputer has a complete DMA interface set of chips, with which the 8271 can communicate via the DRQ and DACK signals at pins 7 and 8. The BBC Microcomputer has no DMA support. This is a pity, because several of the nicer features of the 8271 are really only useful in a DMA environment. In place of DMA, the BBC Microcomputer uses the interrupt system. The 8271 sends a signal at pin Undesignated INT, to notify the 6502 that data is available for it on the data bus. Thus the 6502 is interrupted for each and every byte read from the diskette. In fact this is true for every input/output interface on the BBC Microcomputer. The disk system, however, is different from most other I/O interfaces. In common with the ECONET system, the disk system uses NMI (Non-Maskable Interrupts). This simply means that the 6502 cannot switch these interrupts off. Much more will be said of NMI later in the book.

There are just three signals left to cover, all of which have to do with addressing. The disk interface chips recognise certain addresses as representing the 8271. They intervene whenever they see one of these addresses on the address bus. They supply three signals to the 8271. The first, the Chip Select signal (designated CS) at pin 24 simply notifies the 8271 that it has been selected. Additionally, the 8271 has several different internal, addressable registers and the signals AO (pin 21) and Al (pin 22) identify to the 8271 which register is being invoked. Figure 13 lists them. Note also that data to be read or written is accessed at &FE84.

read/ write	AO	A1	internal register	add- ress
read	0	0	status	&FE80
read	0	1	result	&FE81
write	0	0	command	&FE80
write	0	1	parameter	&FE81
write	1	1	reset	&FE82
read and write data				&FE84

Figure 13. 8271 Addressing.

3.2 8271 registers

Before we proceed further with this description of the 8271, it is worth pointing out that the Acorn OS / DFS supports all of the usable 8271 commands via OSWORD. There is little point in peeking and poking the 8271 yourself. Nevertheless, if we are to make sensible use of the OSWORD facilities, we need to understand what is actually going on.

The 8271 has five internal registers with which we may communicate in order to get the 8271 to carry out our wishes. They are tabulated in Figure 13. Three of these registers can only be written into (not read from) whilst the other two can only be read from (not written into). Thus the three addresses &FE80 to &FE82 suffice to address these registers. In addition the 8271 has 12 Special Registers, with which we cannot communicate directly. We can however use the Read Special Register and Write Special Register commands of the 8271 to access them and alter their contents.

The execution of a task on the 8271 generally requires four separate phases.

Firstly, there is the Command Phase. In this phase we simply check the Status Register to ensure that the 8271 is ready to receive a command, and then transmit the command code to the Command Register.

The second phase is the Parameter Phase. Many commands have associated parameters which further define what the 8271 is to do. In this phase we send each of the required parameters in turn to the Parameter Register, first checking the Status Register each time.

The third phase, which applies only to those commands that read or write data, is the Read/Write Phase. Since all bytes which are read or written are controlled by the NMI interrupt scheme, the Read/Write Phase is actually incorporated into the NMI interrupt handler routine. In this phase, we check to see if the 8271 is ready to send or receive a byte, and then read it or write it at &FE84. This process is repeated until all the bytes to be read or written have been handled.

Lastly there is the Result Phase. If the command generates an interrupt, it also places a result in the Result Register. Some commands update the Result Register without generating an interrupt. When the processing of results is interrupt driven, the Result Phase is part of the NMI interrupt handler. We examine the Status Register to ensure that the result is available and finally read the Result Register.

Let us begin our discussion of the 8271 registers by examining the Status Register. We shall start at the least significant (rightmost) bits and examine each one in turn.

Status Register bits 0 and 1

These are not used by the 8271. They are always zero.

Status Register bit 2

This status bit is only applicable to non-DMA access, i.e. it applies to the BBC Microcomputer.

1 = the 8271 is ready to receive a byte of data (disk write) or send a byte of data (disk read).
0 = No data request pending.

Status Register bit 3

This status bit is the interrupt bit and reflects the INT signal. It is reset every time the Result Register is read.

1 = interrupt pending.

0 = no outstanding interrupt.

Status Register bit 4

This status bit denotes the Result Register Status.

1 = Result Register contains a result.
0 = Result Register is empty.

Status Register bit 5

This status bit is used to determine when it is safe to write another parameter to the Parameter Register.

1 = Parameter Register in use. Do not write.
0 = Safe to write to Parameter Register.

Status Register bit 6

This status bit is set when the 8271 first receives a command and cleared immediately afterwards. It is therefore no use at all to us and we should look to bit 7 to check if the Command Register is ready.

1 = command just received.
0 = command not just received but it may not be safe to send another command yet.

Status Register bit 7

This status bit determines the accessibility of the Command Register.

1 = Command Register in use. Do not write.
0 = Safe to write to Command Register.

At the end of most of its commands the 8271 indicates the outcome in the Result Register, which the program should then examine. Reading the Result Register clears any interrupt. We will continue our examination of the 8271 by looking more closely at the Result Register. This time we shall start at the most significant (leftmost) bit.

Result Register bits 7 **and** 6

Not used. Always zero.

Result Register bit 5

This bit is set if deleted data was found during the command. You will remember from Chapter 1 that deleted data is preceded by a Special Mark called the Deleted Data Mark in place of the normal Data Mark. This is how the 8271 recognises deleted data.

1 = deleted data found.

0 = no deleted data found.

Result Register bits 4 **and** 3

These two bits are the completion type.

0	= no error.
1	= recoverable system error.
10	= unrecoverable error - intervention required.
11	= bad command or faulty drive.

Result Register bits 2 and 1

These two bits are the completion code. When combined with the completion type
above they give more information.

type	code	hex	meaning
00	00	0	good or scan not met
00	01	1	scan met equal
00	10	2	scan met not equal
00	11	3	---
01	00	4	clock error
01	01	5	late DMA (not BBC)
01	10	6	Sector Id CRC error
01	11	7	DATA CRC error
10	00	8	drive not ready
10	01	9	write protect error
10	10	A	track 0 not found
10	11	B	write fault
11	00	C	sector not found
11	01	D	---
11	10	E	---
11	11	F	---

Figure 14. Completion Type/Code.

Result Register bit 0

Not used. Always zero.

The contents of the entire Result Register are displayed by the Acorn DFS in its 'Disk fault' and 'Drive fault' messages. In most cases, this has the effect of multiplying the above error codes by two.

We have said that we send commands to the Command Register and parameters to the Parameter Register, checking the Status Register en route, and where appropriate data is transferred at FE84 and the Result Register is read at the end. Figures 15 to 17 are flowcharts which give a general description of the Command Phase, Parameter Phase and Result Phase. The Read/Write Phase is more command-specific and is not flow-charted.

```
              |
 ┌--->--------|
 |            |
 |    read status register
 |            |
 |            |          no
 |    does bit 5 = 0?-----ABORT
 |            |
 |            |  yes
 | no         |
 └---does bit 7 = 0?
              |
              |  yes
              |
    write command register
```

Figure 15. 8271 Command Phase Flowchart.

```
    for each parameter
              |
 ┌--->--------|
 |            |
 |    read status register
 | no
 └---does bit 5 = 0?
              |
              |  yes
              |
      write parameter register
              |
            next
```

Figure 16. 8271 Parameter Phase Flowchart.

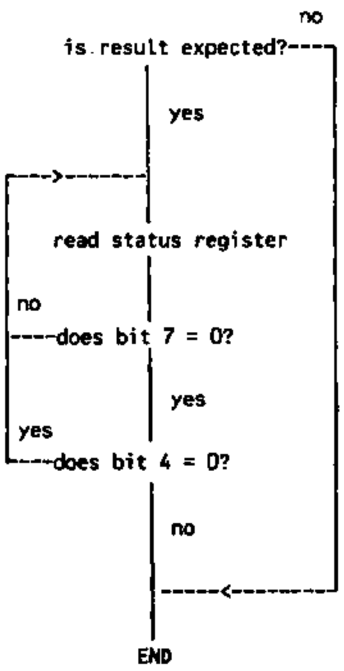


Figure 17. 8271 Result Phase Flowchart.

Figure 18 lists the Special Registers, which we shall describe later when we deal with the commands that use them. These registers are primarily used internally by the 8271, but in some cases we can manipulate their contents lor special purposes. We can also read their contents, especially useful as a diagnostic aid.

hex add- ress	description of register
06	Scan Sector Register
10	Bad Track Reg 1 - drive 0/2
11	Bad Track Reg 2 - drive 0/2
12	Track Register - drive 0/2
13	Scan Count Register (LSB)
14	Scan Count Register (MSB)
17	DMA Mode Register
38	Bad Track Reg 1 - drive 1/3
19	Bad Track Reg 2 - drive 1/3
1A	Track Register - drive 1/3
22	Drive Control Input Register
23	Drive Control Output Register

Figure 18. 8271 Special Registers.

3.3 8271 commands

As previously stated, all the 8271 commands can be executed much more conveniently via OSWORD and indeed the use of OSWORD is fully described later in this book. However, it is highly instructional to delve a bit more deeply into the workings of the 8271.

The 8271 Command Register uses bits 7 and 6 to specify to which drive the command applies. This complication, together with all checking of the Status and Result Registers, is handled by OSWORD for you. Figure 19 summarises all the available 8271 commands, specifying for each whether or not an interrupt is generated, and the number of parameters.

Those commands which are drive-specific are only shown in their drive 0 form. The only commands that are independent of the drive are:

initialise 8271

load bad tracks

reset 8271

All other commands are dependent on the drive number. For drive 1, add &40 to the op-code. For drive 2, add &80. For drive 3, add &C0.

hex op-code	int	no. parms	command
00	Y	5	scan data
04	Y	5	scan data and deleted data multi-sector
0A	Y	2	write data 128 bytes
OB	Y	3	write data multi-sector
0E	Y	2	write deleted data 128 bytes
OF	Y	3	write deleted data multi-sector
12	Y	2	read data 128 bytes
13	Y	3	read data multi-sector
16	Y	2	read data and deleted data 128 bytes
17	Y	3	read data and deleted data multi-sector
IB	Y	3	read sector id(s)
1E	Y	2	verify data and deleted data 128 bytes
1F	Y	3	verify data and deleted data multi-sector
23	Y	5	format track
29	Y	1	seek
2C	N	0	read drive status
35	N	4	initialise 8271
35	N	4	load bad tracks
3A	N	2	write special register
3D	N	1	read special register
01/00	N	0	reset 8271 (written to Reset Register)

Figure 19. 8271 Command Set.

Notice that the &35 op-code has duplicate functions. The first parameter resolves which function is performed.

3.3.1 Reset

The Reset command differs from all the others. It has a two-byte op-code, 01 followed by 00, which are sent in succession to the Reset Register rather than the Command Register. Thus:

```
100 ?&FE82 = 1   :  7&FE82 = 0
```

This code would not work across the Tube, of course, and the following code which achieves the same object is preferred:

```
100 *FX151,130,1
110 *FX151/130,0
```

The Reset command is exactly equivalent to the RESET input signal on the 8271. Note that the 8271 needs time to process the first byte (01) before it receives the second byte (00). In fact a minimum of 10 clock cycles is required. In practice, the time spent by BASIC and OS processing the *FX call will be more than adequate. If you do not have a Tube and are writing non-Tube-compatible machine code, you should insert five NOP instructions.

```
100 LDX #1
110  LDY #0
120  STX &FE82
130  NOP
140  NOP
150  NOP
160  NOP
170  NOP
180  STY &FE82
```

The Reset command causes the following to happen: all output

signals to the disk drive are forced low any command in

progress is halted the Status Register is zeroised the 8271

goes idle the head is unloaded

3.3.2 Initialise

This command initialises certain key parameters used by the 8271. It is used internally by the Acorn DFS to initialise the 8271 according to link-3 and link-4- settings (see Figure 8) unless modified by OSBYTE (see Figure 9).

command   : &35
parameter 1: &0D
parameter 2: step time
parameter 3: settling time
parameter 4: unload time/load time

The first parameter merely distinguishes this Command from the load bad tracks command, which shares the same op-code.

The second parameter controls the rate at which step pulses are sent to the disk drive by the 8271. This rate is specified in two millisecond intervals and is a number in the range 1 to 255, representing times of 2 to 510 milliseconds. You can set this parameter to zero, in which case you are telling the 8271 that the disk drive will provide its own step pulses and the 8271 should examine the COUNT/OPl signal to determine the position of the head.

The third parameter specifies the head settling time delay in two millisecond intervals. It is a number in the range 0 to 255 specifying a delay of 0 to 510 milliseconds.

The last parameter contains two specifications. The four most significant bits represent the amount of time that the head will remain loaded after completion of a command. This is expressed as a number between 0 and 14 representing the number of complete revolutions of the diskette before the head is unloaded. The value 15 may also be used to specify that the head should not be unloaded at all. The least significant four bits specify the time it takes to load the head in 8 millisecond intervals. It is a number between 0 and 15 representing head load times of 0 to 120 milliseconds.

3.3.3 **Load Bad Tracks**

This command informs the 8271 that zero, one or two tracks on the diskette are bad and any bad tracks should not be used. The Acorn DFS does not mark bad tracks, presumably because modern diskettes are sufficiently reliable to render this command obsolete. The effect of the command is to set the two Bad Tracks Registers for the specified pair of drives. The 827l accesses these registers before each I/O operation to avoid the use of bad tracks.

command : &35
parameter 1: &10 for drive 0/2; &18 for drive 1/3
parameter 2: bad track number 1
parameter 3: bad track number 2
parameter 4: current track

The first parameter distinguishes this command from the Initialise command which has the same op-code. It also specifies which pair of drives have bad tracks. &10 specifies drives 0 and 2 whilst &18 specifies drives 1 and 3.

Parameter 2 specifies the first physical track number which is bad. It is a number in the range 0 to 39 for 40-track diskettes or 0 to 79 for 80 track diskettes.

The third parameter specifies the second physical track number which is bad.

The fourth parameter specifies the current track number.

Parameters 2 and 3 may be set to &FF, in which case the 8271 assumes that there are no bad tracks on the diskette.

Initialisation routines should normally do this. Remember that whatever values you send to the 8271 remain in force until either a power-up or a new Load Bad Tracks command is sent. This means that if you mount a different diskette in the drive, the bad tracks will still be marked if you have used this command.

3.3.4 Seek

This command instructs the 8271 to 'seek' a specified physical track. It does not load the head and therefore it does not read Sector Ids to check that it is positioned over the correct track. The 8271 maintains two Track Registers which identify the current track. One is shared by drives 0 and 2, whilst the other is shared by drives 1 and 3. The Seek command uses the appropriate one of these to determine in which direction and by how many steps the head must be moved. In fact Seek is covertly embodied in most of the read/write instructions and it is not usually necessary to issue a Seek command. The 8271 operates differently when commanded to seek physical track 0. In this event, the Seek command will step the head outwards until the TRKO signal indicates that track 0 has been located. The 8271 will step up to 255 times looking for track 0. This makes it a useful command in programs which are uncertain about the current track position. The Seek command performs the following tasks:

checks that the drive is ready

SEEKs the specified track

sets the Result Register

sends an interrupt to the processor

command   : &29
parameter 1: physical track number

The parameter specifies the physical track over which the head is to be positioned.

3.3.5 Read Drive Status

This command interrogates those control signals which are input to the 8271. The information is passed via the Result Register, but an interrupt is not generated. Each bit of the Result Register, except bit 7 which is not used, reflects the status of one of these control signals. All of the signals are presented in positive logic form (0=false, l=true). The information required is maintained by the 8271 in the Drive Control Input Register and the command simply transmits the contents of this register. Note that this command is the only way of clearing a 'not ready' signal. It is used by the DFS for just this purpose.

bit 7 =	unused
bit 6 =	READY1
bit 5 =	FAULT
bit 4 =	INDEX
bit 3 =	WR PROTECT
bit 2 =	READYO
bitl =	TRKO
bit 0 =	COUNT/OP1

command :&2C

This command has no parameters. In the normal way it will not be necessary for application programs to use this command. However, it can be used as a diagnostic aid.

3.3.6 **Read Special Register**

This command instructs the 8271 to send the contents of one of the Special Registers listed in Figure 18. We have already learned about the two Track Registers, the four Bad Track Registers and the Drive Control Input Register. The three Scan Registers are mainly used by the Scan Commands and will be dealt with when we describe those commands.

The Mode Register has bits 7 and 6 set to 1 and bits 5 to 2 set to 0. Bit 1 will normally be set to zero also. It defines how the heads move in double-sided drives. A zero value specifies that the heads move independently, being driven by two independent stepper motors. When bit 1 is set to 1, it indicates that the two heads share a single stepper motor (some drives work this way) and therefore the heads on either side of the diskette will always be positioned over the same physical track number. Bit 0 of the Mode Register will normally be set to 1, specifying non-DMA mode. A microcomputer which uses DMA for disk transfers zeroises this bit.

The Drive Control Output Register reflects the status of those control signals which are output by the 8271.

bit 7	SELECT 1
bit 6	SELECT 0
bit 5	FAULT RESET/OP0
bit 4	LOW CURRENT
bit 3	LOAD HEAD
bit 2	DIRECTION
bit l	SEEK/STEP
bit 0	WR ENABLE

The Read Special Register command transfers the contents of the specified Special Register into the Result Register, but does not generate an interrupt. Regardless of which Special Register is to be read, the command has the format:

command   : &3D
parameter 1: hex address of register

The only parameter specifies which Special Register is to be read. See Figure 18 for a list of addresses. The reply is placed in the Result Register.

3.3.7 **Write Special Register**

This command allows you to manipulate the contents of Special Registers. You must remember that the Special Registers are used internally by the 8271 in its normal operation, so tinkering with their contents is a practice which is not normally to be encouraged. One exception is coping with logical track numbers which differ from physical track numbers. Here it is useful to be able to position the head over the required physical track and then change the appropriate Track Register so that it now equates to the logical track number. This is one way in which you can read the sectors on that track. It is advisable to send a SEEK 0 command when you have read the track, to reset the Track Register and to re-establish a known physical position on the diskette, or alternatively to restore the Track Register to its former value.

command   : &3A
parameter 1: hex address of register
parameter 2: data to put in register

The first parameter identifies the address of the Special Register to be over-written (see Figure 18).

The second parameter contains the value to be written to the Special Register.

3.3.8 **Verify Data and Deleted Data 128 bytes**

This command verifies that the first 128 bytes of any specified logical sector on the diskette are readable. It performs the following tasks:

checks that the drive is ready

SEEKs the specified track

loads the head

locates the required logical sector

checks the Sector Id for correct track number

checks the Sector Id CRC

locates the Data Field, regardless of whether it starts with a Data Mark or a Deleted Data Mark

reads the first 128 bytes of the data

checks the data CRC

sets the Result Register appropriately

sends an interrupt to the processor

command   : &1E
parameter 1: logical track address
parameter 2: logical sector address

The first parameter specifies the logical track number. If this is different from the physical track number, then special steps must be taken.

The second parameter specifies the logical sector number. It does not matter if this is different from the physical sector number, because the 8271 identifies the sector number from the Sector Identification Field which contains the logical sector number.

3.3.9 Verify Data and Deleted Data Multi-sector

This command verifies that a specified number of sectors are readable, starting at any logical sector on the diskette. It performs the following tasks:

> checks that the drive is ready
>
> SEEKs the specified track
>
> loads the head
>
> locates the first required logical sector
>
> checks the Sector Id for correct track number
>
> checks the Sector Id CRC
>
> locates the Data Field, regardless of whether it starts with a Data Mark or a Deleted Data Mark
>
> reads the sector or half-sector as required
>
> checks the data CRC
>
> if any more sectors are to be verified, it repeats the steps above ensuring that each successive sector number is one higher than the previous one.

sets the Result Register appropriately

sends an interrupt to the processor

command : &1F
parameter 1: logical track address
parameter 2: logical sector address
parameter 3: sector size/number of sectors

The first parameter specifies the logical track number. If this is different from the physical track number, then special steps must be taken.

The second parameter specifies the logical sector number. It does not matter if this is different from the physical sector number, because the 8271 identifies the sector number from the Sector Identification Field which contains the logical sector number.

The third parameter contains two fields. The most significant three bits represent the sector size:

0	=	128 bytes per sector
1	=	256 bytes per sector
10	=	512 bytes per sector
11	=	1024 bytes per sector
100	=	2048 bytes per sector
101	=	4096 bytes per sector
110	=	8192 bytes per sector
111	=	16384 bytes per sector

The least significant five bits represent the number of sectors per track and can have any value between zero and &1F.

In practice, the Acorn DFS works with 256 bytes per sector. The allowable values for this whole field are thus:

&21	=	1 sector of 256 bytes
&22	=	2 sectors of 256 bytes
&23	=	3 sectors of 256 bytes
&24	=	4 sectors of 256 bytes
&25	=	5 sectors of 256 bytes
&26	=	6 sectors of 256 bytes

&27 =	7 sectors of 256 bytes
&28 =	8 sectors of 256 bytes
&29 =	9 sectors of 256 bytes
&2A =	10 sectors of 256 bytes

Note that in multi-sector operations, the 8271 expects each successive logical sector number accessed to be one higher than the previous one.

3.3.10 Format Track

This command formats a specified physical track, using the information supplied in the parameters. Note that some books tell you that you must subtract 6 from each of the gap parameters to leave room for the sync bytes. The gap sizes printed in 1.2 of this book already allow for the sync bytes and so this is unnecessary.

command	: &23
parameter 1:	physical track address
parameter 2:	gap 3 size
parameter 3:	sector size/number of sectors
parameter 4:	gap 5 size
parameter 5:	gap 1 size

The first parameter specifies the physical track number.

The second parameter specifies the size of gap 3 in bytes exclusive of sync bytes.

The third parameter specifies two fields. The most significant three bits represent the sector size:

000 =	128 bytes per sector
001 =	256 bytes per sector
010 =	512 bytes per sector
011 =	1024 bytes per sector
100 =	2048 bytes per sector
101 =	4096 bytes per sector
110 =	8192 bytes per sector
111 =	16384 bytes per sector

The least significant five bits represent the number of sectors per track and can have any value between zero and &1F.

In practice, the Acorn DFS works with 256 bytes per sector. The allowable values for this whole field are thus:

&21 =	1 sector of 256 bytes
&22 =	2 sectors of 256 bytes
&23 =	3 sectors of 256 bytes
&24 =	4 sectors of 256 bytes
&25 =	5 sectors of 256 bytes
&26 =	6 sectors of 256 bytes
&27 =	7 sectors of 256 bytes
&28 =	8 sectors of 256 bytes
&29 =	9 sectors of 256 bytes
&2A =	10 sectors of 256 bytes

Note that in multi-sector operations, the 8271 expects each successive logical sector number accessed to be one higher than the previous one.

The fourth parameter specifies the size of gap 5 in bytes exclusive of sync bytes.

The last parameter specifies the size of gap 1 in bytes exclusive of sync bytes.

Note that the 8271 calculates appropriate values for gap 2 and gap 4, based on the values supplied in these five parameters.

3.3.11 Read Sector Id(s)

This command reads one or more Sector Ids starting at physical sector zero of a specified physical track. The ids are accessible at &FE84, byte by byte. Each id transferred consists of 4 bytes:

logical track number
head number (normally = 0 i.e. not used)
logical sector number
data size

The logical track numbers contained within each Sector Id are not checked against the physical track number by this command. This enables you to use it to get round logical sectoring problems, since you can ascertain both the logical track numbers and the logical sector numbers used in any physical track. Note, however, that Sector Id CRCs are checked by this command.

command   : &1B
parameter 1: physical track address
parameter 2: 0
parameter 3: number of id fields to be read

The first parameter specifies the physical track number.

The second parameter must be set to zero.

The last parameter specifies the number of Sector Id fields to be read. It must be within the range 0 to &A.

3.3.12 Read Data 128 Bytes

This command reads the first 128 bytes of any specified logical sector on the diskette. It performs the following tasks:

checks that the drive is ready

SEEKs the specified track

loads the head

locates the required logical sector

checks the Sector Id for correct track number

checks the Sector Id CRC

locates the Data Field

skips out if the sector starts with a Deleted Data Mark

for each byte read, makes byte available on &FE84 and sends an interrupt to the processor

checks the data CRC

sets the Result Register appropriately

sends an interrupt to the processor

command   : &12
parameter 1: logical track address
parameter 2: logical sector address

The first parameter specifies the logical track number. If this is different from the physical track number, then special steps must be taken.

The second parameter specifies the logical sector number. It does not matter if this is different from the physical sector number, because the 8271 identifies the sector number from the Sector Identification Field which contains the logical sector number.

3.3.13 **Read Data and Deleted Data 128 Bytes**

This command reads the first 128 bytes of any logical sector on the diskette, including any data marked as deleted data. It performs the following tasks:

checks that the drive is ready

SEEKs the specified track

loads the head

locates the required logical sector

checks the Sector Id for correct track number

checks the Sector Id CRC

locates the Data Field, regardless of whether it starts with a Data Mark or a Deleted Data Mark

for each byte read, makes byte available on &FE84 and sends an interrupt to the processor

checks the data CRC

sets the Result Register appropriately

sends an interrupt to the processor

command   : &16
parameter 1: logical track address
parameter 2: logical sector address

The first parameter specifies the logical track number. If this is different from the physical track number, then special steps must be taken.

The second parameter specifies the logical sector number. It does not matter if this is different from the physical sector number, because the 8271 identifies the sector number from the Sector Identification Field which contains the logical sector number.

3.3.14 Read Data Multi-sector

This command reads a specified number of sectors starting at a specified logical sector on the diskette. It performs the following tasks:

checks that the drive is ready

SEEKs the specified track

loads the head

locates the required logical sector

checks the Sector Id for correct track number

checks the Sector Id CRC

locates the Data Field

skips over deleted data but counts the number of deleted bytes towards the total number of bytes to be transferred

for each byte read, makes byte available on &FE84 and sends an interrupt to the processor

checks the data CRC

repeats the steps above until all the data has been processed, checking that each logical sector number is one higher than the previous one.

sets the Result Register appropriately

sends an interrupt to the processor

command : &13
parameter 1: logical track address
parameter 2: logical sector address
parameter 3: sector size/number of sectors

The first parameter specifies the logical track number. If this is different from the physical track number, then special steps must be taken.

The second parameter specifies the logical sector number. It does not matter if this is different from the physical sector number, because the 8271 identifies the sector number from the Sector Identification Field which contains the logical sector number.

The third parameter contains two fields. The most significant three bits represent the sector size:

0	-	128 bytes per sector
1	=	256 bytes per sector
10	=	512 bytes per sector
11	=	1024 bytes per sector
100	=	2048 bytes per sector
101	=	4096 bytes per sector
110	=	8192 bytes per sector
111	=	16384 bytes per sector

The least significant five bits represent the number of sectors per track and can have any value between zero and &lF.

In practice, the Acorn DFS works with 256 bytes per sector. The allowable values for this whole field are thus:

&21	=	1 sector of 256 bytes
&22	=	2 sectors of 256 bytes
&23	=	3 sectors of 256 bytes
&24	=	4 sectors of 256 bytes
&25	=	5 sectors of 256 bytes
&26	=	6 sectors of 256 bytes

&27 =	7 sectors of 256 bytes
&28 =	8 sectors of 256 bytes
&29 =	9 sectors of 256 bytes
&2A =	10 sectors of 256 bytes

Note that in multi-sector operations, the 8271 expects each successive logical sector number accessed to be one higher than the previous one.

3.3.15 Read Data and Deleted Data Multi-sector

This command reads a specified number of sectors starting at a specied logical sector on the diskette, including any data marked as deleted data. It performs the following tasks:

checks that the drive is ready

SEEKs the specified track

loads the head

locates the required logical sector

checks the Sector Id for correct track number

checks the Sector Id CRC

locates the Data Field, regardless of whether it starts with a Data Mark or a Deleted Data Mark

for each byte read, makes byte available on &FE84 and sends an interrupt to the processor

checks the data CRC

repeats the steps above until all the data has been processed, checking that each logical sector number is one higher than the previous one.

sets the Result Register appropriately

sends an interrupt to the processor

command   : &17
parameter 1: logical track address
parameter 2: logical sector address
parameter 3: sector size/number of sectors

The first parameter specifies the logical track number. If this is different from the physical track number, then special steps must be taken.

The second parameter specifies the logical sector number. It does not matter if this is different from the physical sector number, because the 8271 identifies the sector number from the Sector Identification Field which contains the logical sector number.

The third parameter contains two fields. The most significant three bits represent the sector size:

0	=	128 bytes per sector
1	=	256 bytes per sector
10	=	512 bytes per sector
11	=	1024 bytes per sector
100	=	2048 bytes per sector
101	=	4096 bytes per sector
110	=	8192 bytes per sector
111	=	16384 bytes per sector

The least significant five bits represent the number of sectors per track and can have any value between zero and &1F.

In practice, the Acorn DFS works with 256 bytes per sector. The allowable values for this whole field are thus:

&21	=	1 sector of 256 bytes
&22	=	2 sectors of 256 bytes
&23	=	3 sectors of 256 bytes
&24	=	4 sectors of 256 bytes
&25	=	5 sectors of 256 bytes
&26	=	6 sectors of 256 bytes
&27	=	7 sectors of 256 bytes
&28	=	8 sectors of 256 bytes
&29	=	9 sectors of 256 bytes
&2A	=	10 sectors of 256 bytes

Note that in multi-sector operations, the 8271 expects each successive logical sector number accessed to be one higher than the previous one.

3.3.16 Write Data 128 Bytes

This command writes 128 bytes of data, transferred byte by byte at &FE84, to any, specified logical sector on the diskette. It performs the following tasks:

checks that the drive is ready

SEEKs the specified track

loads the head

locates the required logical sector

checks the Sector Id for correct track number

checks the Sector Id CRC

locates the Data Field and writes a Data Mark

for each byte to be written, obtains the byte at &FE84

calculates and writes a data CRC

sets the Result Register appropriately

sends an interrupt to the processor

command : &0A
parameter 1: logical track address
parameter 2: logical sector address

The first parameter specifies the logical track number. If this is different from the physical track number, then special steps must be taken.

The second parameter specifies the logical sector number. It does not matter if this is different from the physical sector number, because the 8271 identifies the sector number from the Sector Identification Field which contains the logical sector number.

3.3.17 Write Deleted Data 128 Bytes

This command writes 128 bytes of deleted data, transferred byte by byte at &FE84, to any, specified logical sector on the diskette. It performs the following tasks:

checks that the drive is ready

SEEKs the specified track

loads the head

locates the required logical sector

checks the Sector Id for correct track number

checks the Sector Id CRC

locates the Data Field and writes a Deleted Data Mark

for each byte to be written, obtains the byte at &FE84

calculates and writes a data CRC

sets the Result Register appropriately

sends an interrupt to the processor

command   : &0E
parameter 1: logical track address
parameter 2: logical sector address

The first parameter specifies the logical track number. If this is different from the physical track number, then special steps must be taken.

The second parameter specifies the logical sector number. It does not matter if this is different from the physical sector number, because the 8271 identifies the sector number from the Sector Identification Field which contains the logical sector number.

3.3.18 Write Data Multi-sector

This command writes a specified number of sectors starting at a specified logical sector on the diskette. It performs the following tasks:

checks that the drive is ready

SEEKs the specified track

loads the head

locates the required logical sector

checks the Sector Id for correct track number

checks the Sector Id CRC

locates the Data Field and writes a Data Mark

for each byte to be written, obtains the byte at &FE84

calculates and writes a data CRC

repeats the above steps until all bytes are processed

sets the Result Register appropriately

sends an interrupt to the processor

command : &0B
parameter 1: logical track address
parameter 2: logical sector address
parameter 3: sector size/number of sectors

The first parameter specifies the logical track number. If this is different from the physical track number, then special steps must be taken.

The second parameter specifies the logical sector number. It does not matter if this is different from the physical sector number, because the 8271 identifies the sector number from the Sector Identification Field which contains the logical sector number.

The third parameter contains two fields. The most significant three bits represent the sector size:

0	=	128 bytes per sector
1	=	256 bytes per sector
10	=	512 bytes per sector
11	=	1024 bytes per sector
100	=	2048 bytes per sector
101	=	4096 bytes per sector
110	=	8192 bytes per sector
111	=	16384 bytes per sector

The least significant five bits represent the number of sectors per track and can have any value between zero and &1F.

In practice, the Acorn DFS works with 256 bytes per sector. The allowable values for this whole field are thus:

&21	=	1 sector of 256 bytes
&22	=	2 sectors of 256 bytes
&23	=	3 sectors of 256 bytes
&24	=	4 sectors of 256 bytes
&25	=	5 sectors of 256 bytes
&26	=	6 sectors of 256 bytes
&27	=	7 sectors of 256 bytes
&28	=	8 sectors of 256 bytes
&29	=	9 sectors of 256 bytes
&2A	=	10 sectors of 256 bytes

Note that in multi-sector operations, the 8271 expects each successive logical sector number accessed to be one higher than the previous one.

3.3.19 Write Deleted Data Multi-sector

This command writes a specified number of sectors of deleted data starting at a specied logical sector on the diskette, using Deleted Data Marks. It performs the following tasks:

checks that the drive is ready

SEEKs the specified track loads

the head

locates the required logical sector

checks the Sector Id for correct track number

checks the Sector Id CRC

locates the Data Field and writes a Deleted Data Mark

for each byte to be written, obtains the byte at &FE84

calculates and writes a data CRC

repeats the above steps until all bytes are processed

sets the Result Register appropriately

sends an interrupt to the processor

command : &0F
parameter 1: logical track address
parameter 2: logical sector address
parameter 3: sector size/number of sectors

The first parameter specifies the logical track number. If this is different from the physical track number, then special steps must be taken.

The second parameter specifies the logical sector number. It does not matter if this is different from the physical sector number, because the 8271 identifies the sector number from the Sector Identification Field which contains the logical sector number.

The third parameter contains two fields. The most significant three bits represent the sector size:

0	=	128 bytes per sector
1	=	256 bytes per sector
10	=	512 bytes per sector
11	=	1024 bytes per sector
100	=	2048 bytes per sector
101	=	4096 bytes per sector
110	=	8192 bytes per sector
111	=	16384 bytes per sector

The least significant five bits represent the number of sectors per track and can have any value between zero and &1F.

In practice, the Acorn DFS works with 256 bytes per sector. The allowable values for this whole field are thus:

&21 =	1 sector of 256 bytes
&22 =	2 sectors of 256 bytes
&23 =	3 sectors of 256 bytes
&24 =	4 sectors of 256 bytes
&25 =	5 sectors of 256 bytes
&26 =	6 sectors of 256 bytes
&27 =	7 sectors of 256 bytes
&28 =	8 sectors of 256 bytes
&29 =	9 sectors of 256 bytes
&2A =	10 sectors of 256 bytes

Note that in multi-sector operations, the 8271 expects each successive logical sector number accessed to be one higher than the previous one.

3.3.20 Scan Data Commands

These commands assume that the disk system incorporates a DMA interface, which is certainly not the case with the BBC Microcomputer. They allow the data on a diskette to be matched against a key held in memory and can be used to locate specified data patterns on the diskette. To do this the 8271 must simultaneously read bytes from the diskette and the key from memory, not really feasible in a non-DMA environment. These two commands will not be explained in detail in this book since they cannot be used sensibly by the BBC Microcomputer. Note, however, that of the three Scan Registers, the Scan Sector Register is also used by the multi-sector read and write commands. If any of these commands errors on CRC checking, the sector number at which the error occurred will be found in the Scan Sector Register, which can be interrogated by the Read Special Register Command.

4 Acorn DFS Introduction

The Acorn DFS is supplied in ROM. There are many different versions of this ROM, but the main releases are:

DFS 0.90 first main release of DFS
DFS 1.20 current release of DFS
DNFS 3.0 supplied with second processors, equivalent to DFS 1.20

In addition, there have been other releases in the 0.9 series from 0.90 to 0.97 and also 0.9E and 0.9H. In this book, we deal only with the two main releases (0.90 and 1.20). You should enter *HELP to discover which you have. All remarks in this and subsequent chapters refer to Acorn DFS 0.90 unless the sentence specifically mentions a different version.

The Acorn DFS code provides:

a help facility invoked by the *HELP command.

a set of disk-specific star commands:

*ACCESS	*BACKUP	*BUILD
*COMPACT	*COPY	*DELETE
*DESTROY	*DIR	*DISC
*DRIVE	*DUMP	*ENABLE
*INFO	*LIB	*LIST
*RENAME	*TITLE	*TYPE
*WIPE		

(note: *BUILD, *DUMP, *LIST and *TYPE will also work on any other current filing system)

disk-specific filing system code which can be executed at low-level, but which also provides filing system support for star commands and BASIC.

OSFILE	(supports *BUILD, CHAIN, *EXEC, *LOAD, LOAD, *SAVE, SAVE, *SPOOL)
OSARGS	(supports EXT#, PTR#)
OSBGET	(supports BGET#)
OSBPUT	(supports BPUT#)

OSGBPB	(supports INPUT#, PR1NT#)
OSF1ND	(supports CLOSE#, OPENIN, OPENOUT, OPENUP)
OSFSC	(supports *CAT, EOF#, *OPT, *RUN)

low-level disk interface support in the form of disk-specific code for OSWORD (&7D to &7F)

a set of NMI interrupt handler routines

support for the Tube (when present)

In addition, since the DFS is a ROM with a service entry, it contains code to support the OS service ROM calls.

Each of these topics will be covered in this book. For the moment we need to concentrate on one of the fundamental design features of the Acorn DFS, the disk catalogue.

4.1 Catalogue

The disk catalogue contains information about all the disk files on one side of the diskette. Double-sided diskettes have separate catalogues on each side. The catalogue occupies logical sectors 0 and 1 of physical track 0. Physical track 0 is an ideal choice, because it is the same for both 40 track and 80 track diskettes. This means that if you insert a 40 track diskette into an 80 track drive (or vice-versa), you can still read the catalogue, even if you cannot read much else.

Since the catalogue occupies a fixed part of the diskette, this implies that there must be some constraints imposed on the number of files etc. that we can store per side of diskette. Indeed this is true. The constraints are:

a disk title of 12 or less characters

no more than 31 files per side

each file name is 7 or less characters long plus a single-character directory code

each filename within a given directory must be unique each file starts on a new sector - never in the middle of a sector

each file occupies a single, contiguous space on the diskette

each file is confined to one side of a diskette

The individual characters allowed in a file name or a directory are:

'a' to 'z'
'A' to 'Z'
'0' to '9'
any symbol except '*',':','!'/.' or space

Note, however, that the Acorn DFS does not distinguish between lower-case letters and upper-case letters in filenames. DFS 0.90 does distinguish lower-case letters in directories, but DFS 1.20 does not. In all, it is best to avoid lower-case letters.

Many people choose to contrive all sorts of weird effects by tinkering with file names and titles. Examples include teletext control codes at the start of the disk title, so that when the catalogue is printed on the screen the title appears in glorious colour. Some Software Publishers even put control code sequences at the end of file names so that you cannot print the catalogue at all. More will be said about this later. For the moment, I will just say that this is one area where there is widespread incompatibility between the various DFS roms on the market, and it is best to restrict file names to the uppercase letters and the numbers '0' to '9'.

The directory code serves to group together similar files, such that they be acted upon globally by certain DFS commands. The directory character can be 'A' to 'Z' or '$' which is the default. Although other values are theoretically allowed, it is best to avoid them.

4.1.1 Catalogue sector 0.

Bytes 00 to 07
contain the first eight bytes of the disk title.

The remaining 248 bytes consist of file name information for up to 31 files. Thus there are eight bytes of file name information per file. We shall examine the information available for file number 1. The remaining 30 files have identical format.

Bytes 08 to 0E
are the seven-byte file name of file number 1 Spaces are used to pad out the name if it less than seven bytes long.

Byte 0F
contains the directory code for file number 1.

Figure 20 summarises sector 0 of the catalogue.

hex-add	contents
00 - 07	first 8 bytes of title
08 - 0F	file number 1 - name+dir
10 - 17	file number 2 - name+dir
18 - 1F	file number 3 - name+dir
20 - 27	file number 4 - name+dir
28 - 2F	file number 5 - name+dir
30 - 37	file number 6 - name+dir
38 - 3F	file number 7 - name+dir
40 - 47	file number 8 - name+dir
48 - 4F	file number 9 - name+dir
50 - 57	file number 10 - name+dir
58 - 5F	file number 11 - name+dir
60 - 67	file number 12 - name+dir
68 - 6F	file number 13 - name+dir
70 - 77	file number 14 - name+dir
78 - 7F	file number 15 - name+dir
80 - 87	file number 16 - name+dir
88 - 8F	file number 17 - name+dir
90 - 97	file number 18 - name+dir
98 - 9F	file number 19 - name+dir
A0 - A7	file number 20 - name+dir
A8 - AF	file number 21 - name+dir
B0 - B7	file number 22 - name+dir
B8 - BF	file number 23 - name+dir
C0 - C7	file number 24 - name+dir
C8 - CF	file number 25 - name+dir
D0 - D7	file number 26 - name+dir
D8 - DF	file number 27 - name+dir
E0 - E7	file number 28 - name+dir
E8 - EF	file number 29 - name+dir
F0 - F7	file number 30 - name+dir
F8 - FF	file number 31 - name+dir

Figure 20. Catalogue Sector 0.

4.1.2 Catalogue sector 1.

You are probably familiar with the thirty-two bit addresses used by the Acorn OS for file load addresses, execute addresses etc. These sizes allow the addition of a second processor across the Tube. The most significant sixteen bits are set to &FFFF whenever the address is in the I/O processor (the original BBC Microcomputer). Addresses with the most significant sixteen bits set to zero point to the second processor, when present. In order to save space, the disk catalogue uses eighteen-bit addressing. If the most significant two bits are set, the address is in the I/O processor. Otherwise it is in the second processor. Eighteen-bit addressing is sufficient to allow memory locations up to 256K to be addressed. Some of the locations described below hold eighteen-bit addresses. You will see that they do this by holding the least significant sixteen bits in two adjacent memory locations and the most significant two bits as part of a different memory location.

Bytes 00 to 03
contain the last 4 bytes of the title.

Byte 04
contains the number of times the diskette has been written to.

Byte 05
contains 8 * (the number of files on this side of the diskette).

Byte 06 - bits 7 **and 6**
are not used.

Byte 06 - bits 5 **and** 4
contain the boot-up option (i.e. the *OPT4 parameter)

Byte 06 - bits 3 and 2
are not used.

Byte 06 - bits 1 and 0
contain the most significant two bits of the number of sectors on the diskette.

Byte 07
contains the least significant eight bits of the number of sectors on the diskette. Thus in conjunction with bits 1 and 0 of byte 06, this forms a 10-bit number.

The remaining 248 bytes of sector 1 contain the file pointers for each of the 31 files. All of the pointers are 18-bit fields and therefore extend over more than one byte. There are 8 bytes of pointer information per file. We shall examine in detail the pointers for file number 1. The format of these pointers is then repeated a further thirty times.

Bytes 8 to 09
contain the least significant sixteen bits of the load address of file number 1.

Bytes A to 0B
contain the least significant sixteen bits of the execute address of file number 1.

Bytes C to 0D
contain the least significant sixteen bits of the length in bytes of file number 1.

Byte 0E- bits 7 **and 6**
contain the most significant two bits of the execute address of file number 1.

Byte 0E - bits 5 and 4
contain the most significant two bits of the length in bytes of file number **1.**

Byte 0E - bits 3 **and 2**
contain the most significant two bits of the load address of file number 1.

Byte 0E-bits land 2
contain the most significant two bits of the sector number at which file number 1 starts.

Byte 0F
contains the least significant sixteen bits of the sector number at which file number 1 starts. Note that the start sector number is obtained by multiplying the track number at which the file starts by 10, and adding the sector number at which the file starts.

Figure 21 summarises Sector 1 of the catalogue.

hex-add	contents
00 - 03	last 4 bytes of title
04	number of writes to disk
05	8 * number of files
06	bits 7 - 6 not used
06	bits 5 - 4 boot-up option
06	bits 3 - 2 not used
06	bits 1 - 0 no. secs. high
07	no. secs. low
08 - 0F	file number 1 - pointers
10 - 17	file number 2 - pointers
18 - 1F	file number 3 - pointers
20 - 27	file number 4 - pointers
28 - 2F	file number 5 - pointers
30 - 37	file number 6 - pointers
38 - 3F	file number 7 - pointers
40 - 47	file number 8 - pointers
48 - 4F	file number 9 - pointers
50 - 57	file number 10 - pointers
58 - 5F	file number 11 - pointers
60 - 67	file number 12 - pointers
68 - 6F	file number 13 - pointers
70 - 77	file number 14 - pointers
78 - 7F	file number 15 - pointers
80 - 87	file number 16 - pointers
88 - 8F	file number 17 - pointers
90 - 97	file number 18 - pointers
98 - 9F	file number 19 - pointers
A0 - A7	file number 20 - pointers
A8 - AF	file number 21 - pointers
B0 - B7	file number 22 - pointers
B8 - BF	file number 23 - pointers
C0 - C7	file number 24 - pointers
C8 - CF	file number 25 - pointers
D0 - D7	file number 26 - pointers
D8 - DF	file number 27 - pointers
E0 - E7	file number 28 - pointers
E8 - EF	file number 29 - pointers
F0 - F7	file number 30 - pointers
F8 - FF	file number 31 - pointers

Figure 21. Catalogue Sector 1.

4.2 DFS*HELP

The Acorn DFS provides two *HELP commands, *HELP DFS and *HELP UTILS. Figure 22 shows the result of running both *HELP commands (tidied up a bit).

star command	arg 1	arg 2	arg 3
*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>		
*LIB	(<dir>)		
*RENAME	<old fsp>	<new fsp>	
*TITLE	<title>		
*WIPE	<afsp>		
*BUILD	<fsp>		
*DISC			
*DUMP	<fsp>		
*LIST	<fsp>		
*TYPE	<fsp>		

Figure 22. *HELP Arguments.

Most of the star commands specify one to three arguments that may be entered to qualify the action of the command. The notation used in this list needs some explanation.

( )

Any argument in round brackets is optional. If you do not key in an argument, the system will use a default.

< >

When the argument is not sandwiched between '<' and '>', you should enter the argument literally (e.g. the 'L' of the * ACCESS command. When the argument is sandwiched between '<' and '>' a value should be substituted.

<fsp>

This means 'file specification', but not using wild cards (see <asfp> for description of wild cards). The full file specification comprises drive, directory and file name, with ':' and '.' used as delimiters. For example, if you have a file named 'PROGRAM' in directory 'S' on drive 3, then its full file specification is:

:3.S.PROGRAM

If you have already keyed in '*DRIVE 3', so that drive 3 is the default, then you can get away with a file specification of:

S. PROGRAM

If in addition you have already keyed in '*DIR S', so that directory 'S' is the default, then you can get away with a file specification of:

PROGRAM

Note also that file specifications such as :3.PROGRAM are acceptable. DFS level 0.90 inserts a directory value of $, regardless of the current value of the default directory, whenever the directory specification is omitted in this way. DFS level 1.20 behaves rather better and inserts the current value of the default directory.

<old fsp>

This means 'old file specification[7] and means that a <fsp> should be entered here. It is used by *RENAME to identify the file whose name you want to change.

<new fsp>

This means 'new file specification' and means that a <fsp> should be entered here. It is used by ^RENAME to indicate that this <fsp> will be the new file name.

<afsp>

This means 'ambiguous file specification', which can be used to specify more than one file at a time. You may use any of the three formats for <fsp> that we have already discussed. You may also use wild cards. There are two wild cards, '#' and '*'. '#' is used in place of a single character and means that you do not care what that character is. '*' is used in place of several characters and means that you do not care what any of them are. Suppose you print a catalogue of drive 3, and it contains the following files:

```
B.PROGRAM
B.PROG1
B.PRIZES
S.PROGRAM
S.PRNT
S.PIXEL
```

Then:

#.PROGRAM	satisfies B.PROGRAM and S.PROGRAM
B.PRO*	satisfies B.PROGRAM and B.PROG1
#.P*	satisfies all the file names

Wild cards are used with commands that can act on more than one file at a time, such as *WIPE.

<drv>

This means 'drive'. A number in the range 0 to 3 should be entered.

<src drv>

This means 'source drive' and again a number in the range 0 to 3 should be entered. It is used in copy commands etc. to indicate that the file(s) will be copied from this drive.

<dest drv>

This means 'destination drive' and again **a** number in the range 0 to 3 should be entered. It is used in copy commands etc. to indicate that the file(s) will be copied to this drive.

<dir>

This means 'directory' and a single character code should be entered.

<title>

This means 'title' and up to twelve characters of title should be entered.

L

This is only used by the * ACCESS command and means that you should enter 'L' to lock a file and nothing at all (or space) to unlock it.

5 Acorn DFS Star Commands

Each disk-related star command is explained (in alphabetic order) including those which are supported by the DFS implementations of OSFILE and OSFSC, which do not appear on the *HELP displays. Note that, whereas CHAIN, LOAD and SAVE (which are BASIC commands) require file names to be entered within quotation marks, the star commands have no such requirement.

It is important to realise that star commands have a distinct pecking order. The system passes a star command to the OS first. If the OS does not recognise it, it offers the command to each paged ROM in turn that has a service entry. If none of these recognise it, it is offered to the current filing system. Assuming this is the DFS, the DFS checks the catalogue to see if a file of that name appears. If it does, it attempts to *RUN that file. See *RUN for a more detailed description.

*ACCESS <afsp>(L)

This command locks or unlocks one or more files (wild cards may be used) on the diskette. The lock mechanism is used to protect valuable data or program files from accidental erasure. To lock a file, you specify the L parameter. To unlock a file you omit it.

The lock feature is similar to the write-protect feature, except that it applies to specific files rather than to the whole diskette. The lock feature is implemented entirely by software, whereas the write-protect feature has hardware components as well.

A locked file is protected from accidental erasure by *DELETE, *RENAME and *WIPE. Moreover it cannot be overwritten by SAVE, *SAVE or *SPOOL. Note, however, that it will be overwritten if the diskette is re-formatted. It will also be overwritten by 'DESTROY or if the diskette is the destination disk in a 'BACKUP command.

The lock status for a file is held in sector 0 of the disk catalogue. It is the most significant bit of the directory field for that file. If vou examine Figure 20, you will see that the file directory fields occupy bytes &0F, &17, &1F, &27 etc. of sector 0.

Suppose that the first file on a diskette is called $.PROG and that you wish to lock this file. You simply enter:

```
*ACCESS S.PROG L
```

The DFS will set the most significant bit of byte &0F of sector 0, which indicates that the file is locked. Instead of containing &24, the ASCII code which represents the dollar character, it will now contain &A4. This bit-juggling is internal to the DFS and is transparent to you. If at a later date you wish to amend this file, you must first unlock it by entering:

```
*ACCESS  $.PR0G
```

The DFS will now reset the most significant bit of byte &0F, so that it now reverts to &24.

*BACKUP <src drv> <dest drv>

This command is used to create a backup copy of an entire diskette. The destination diskette must first have been formatted. If it already contains files, these will be overwritten regardless of whether or not they are locked. To invoke 'BACKUP, you must first enter 'ENABLE (DFS 0.90). This is a safety procedure. In entering 'ENABLE, you are telling the DFS that you are fully aware of the consequences of the command which is to follow.

Thus to copy a diskette in drive zero onto a diskette in drive 1, you would simply enter:

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

To speed up the data transfer, 'BACKUP uses all of the memory space available to it. The inevitable consequence of this is that you will lose any programs that you happen to have in memory at that time. The more memory available to

*BACKUP, the more data can be copied in a single go, and the faster *BACKUP runs. You should, therefore, ensure that you are in Mode 7 prior to using *BACKUP as this will make 20K of memory available. In Mode 7, *BACKUP can copy a complete 40 track diskette in just 6 goes.

Note that you can also use *BACKUP even if you only have one, single-sided disk drive, though it is rather tedious. You simply enter:

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

The DFS will instruct you when to put the source diskette into the drive so that part of it can be read. It will also instruct you when to swop to the destination diskette so that this can be written. This procedure is repeated until all the contents of the source diskette have been copied onto the destination diskette. Clearly it is even more important to ensure that you are in Mode 7 if you only have one drive.

*BACKUP will not cope with peculiar logical sectoring. It expects track numbering and sector numbering to be standard. It cannot be used to copy any commercial software that uses logical sectoring techniques as a security measure.

Note that DFS 1.20 has dispensed with the need for *ENABLE.

*BUILD<fsp>

This command is used to create an ASCII file directly from keyboard input. A file so created can be used by the *EXEC feature of the DFS. *BUILD is particularly useful in creating !BOOT files (see *OPT 4,3). When creating an ASCII file via *BUILD, it is rather similar to creating a program using AUTO 1,1. *BUILD presents you with automatic line numbers until you terminate keyboard entry by hitting the ESCAPE key. A file created by *BUILD will be saved automatically when ESCAPE is entered. For example, to create a !BOOT file you might well enter something like this:

```
*BUILD  !B00T
```

The system responds with:

```
1 You could now enter:
```

```
1  MODE 7
```

When you hit the RETURN key, the system responds with:

```
1 MODE 7
2
```

You could now enter:

```
2 CHAIN "MENU"
```

When you hit the RETURN key, the system responds with:

```
1 MODE 7
2 CHAIN "MENU" 3
```

If you now hit the ESCAPE key, the two-line ASCII file will be saved on the diskette with a filename of !BOOT.

Note that although the example given uses BASIC statements, the !BOOT file is not a BASIC program, since BASIC programs do not contain statements in ASCII format, but are tokenised (i.e. each BASIC command is held as a one-byte number). Thus you cannot LOAD, RUN or CHAIN a file created by *BUILD. You can however use *EXEC. Moreover, in the case of !BOOT files you can invoke them via Shift -I- Break, providing the *OPT4 option is correctly set.

Note also that in Acorn DFS 1.20, the line numbers presented by *BU1LD are prefixed with leading zeroes.

*CAT (<drv>)

This command displays the disk catalogue for the specified drive (or the current drive if no drive is specified). For example, to display the catalogue of the diskette currently in drive 1, you would enter:

```
*CAT 1
```

The command produces a display such as the example shown in Figure 23.

```
DISK-UTILS  (86)
Drive 1            Option 0 (off)
Directory :2.$     Library :0.$
DDISP              DRAM
RD2                ROMDISK
SECEDIT    L
W.DRAM
```

Figure 23. Typical Catalogue Display.

DISK-UTILS is the disk title.

(86) is the number of times (in hex) that the disk has been written on, since it was last formatted.

Drive 1 is the drive for which this catalogue is displayed.

Option 0 is the value of the boot up (*OPT 4) option for this disk, which is annotated with an explanation such as (Off).

Directory :2.$ means that the default general purpose directory is drive 2 and directory $. (see *DIR and *DRIVE).

Library :0.$ means that the default machine-code library is drive 0 and directory $. (see *LIB).

The remaining lines display the files to be found on the diskette. Those files in the default general purpose directory are displayed first, followed by other files in strict alphabetic sequence of filename including the directory prefix. Two files per line are displayed to ensure that the display fits on the screen. Note also that the lock status is displayed as L for any locked files.

*CAT reads the two catalogue sectors into &E00 to &FFF. These two memory pages are generally used by the DFS to contain catalogue information. Note, however, that the DFS may well manipulate data in these two pages. The user should not regard these memory areas as a short-cut to catalogue information. For example, *CAT zeroises the directory fields of all files in the current directory, to ensure that these are displayed first. Also, having displayed a file name, it sets the most significant bit of the first byte of the file name to 1. This is used as an indicator by *CAT to denote that the file name has already been displayed. *CAT then repeats its Search for the next filename (alphabetically), ignoring those marked as already done, until all the filenames have been displayed.

*COMPACT (<drv>)

When a file is deleted from a diskette, the DFS does not actually delete the whole file. It simply removes the catalogue information (in sectors 0 and 1 of track zero), packs the catalogue entries and re-writes the catalogue to the diskette. This can be useful. If you delete a file by mistake, to restore it you need only re-instate the catalogue entries for that file. However, the effect of deleting a file is to leave a gap on the disk where that file was, which may well be too small for future file saves. This is normally undesirable. *COMPACT can be used to shuffle the disk contents around so that any gaps left by deleted files are overwritten. Thus the effect of *COMPACT is to place all the files at the start of the diskette and to place all the spare disk space immediately following the last file, as one, long, contiguous space. This maximises the potential use of the free space on the disk. As part of the display, *COMPACT will display in hex the number of sectors of free space that exists.

Clearly, once a disk is compacted, it is no longer possible to recover any files that have been deleted in error.

To compact the diskette in drive 1, you simply enter:

```
*COMPACT  1
```

If you omit the drive argument, the default general purpose drive will be compacted.

Figure 24 shows a typical *COMPACT display.

```
Compacting drive 1
$.DDISP      FF1900 FF8023 0011A1 002
$.SECEDIT L  FF1900 FF8023 0012EB 014
$.R0MDISK  FF1900  FF8023  000171  Q27
$.RD2        FF1900 FF8023 000171 029
W.DRAM       FF1900 FF8023 00026A 02B
$.DRAM       FF1900 FF8023 000282 02E
Disk compacted 2EF free sectors
```

Figure 24. Typical *COMPACT display.

Each line of the display represents a file remaining after *COMPACT. For each file, the display shows the lock status, the load address, the execute address, the size in hex bytes and the start sector number. Similar information is provided by *INFO.

Note that, if you are unlucky enough to get a disk error during *COMPACT, the whole diskette may well be unreadable. It is prudent to keep more than one copy of valuable files. Note also that *COMPACT makes free use of available memory, so you will lose any program that happens to be in memory. Make sure that you are in Mode 7 to maximise the memory that *COMPACT can use.

*COPY <src drv> <dest drv> <afsp>

This command copies one or more files (wild cards may be used) from a source diskette in the specified source drive to a destination diskette in the specified destination drive. For example:

```
*COPY 0 1 $.PROG
```

This command will copy $.PROG from the diskette in drive 0 to the diskette in drive 1.

The same drive number can be specified for both source and destination drives. In this case, the user will be instructed by the DFS when to swap diskettes in the drive. *COPY displays a line on the screen for each file that it copies, in a format similar to that shown in Figure 24.

Note that this command makes free use of memory to accelerate the copy process and any user program in memory is likely to be corrupted. Switch to Mode 7 to maximise the memory space before using *COPY. *COPY normally tries to buffer a whole file in memory, before writing back to the destination drive. However, if the file is too big to fit into available memory, it splits it up into smaller chunks.

Unlike *BACKUP, *COPY does not destroy the destination diskette. Each file to be copied, that does not already exist on the destination diskette, is written into the free space area of the destination diskette, assuming there is enough room. Any file to be copied, whose name matches a file already on the destination diskette, overwrites that file in situ on the destination diskette, assuming that the new version of that file is not larger. See Chapter 15 for how to cope with 'Can't Extend' messages, generated when the file is too big.

Note that you will not be able to copy a file, if that file already appears on the destination disk with locked status. Note also that *COPY cannot handle files that use peculiar logical sectoring as a security measure.

*DELETE <fsp>

This command deletes the catalogue entry for a single specified file. The file must not be locked, nor must it be opened. The catalogue is re-written to avoid gaps in the catalogue itself, but the file itself is unchanged. The space occupied by a deleted file may be re-used by the DFS to save any new file which is not bigger than that space. In theory, a deleted file can be recovered until such time as the disk space becomes re-allocated or until the diskette is compacted. In practice, it is a little harder than this. A useful recovery program is described in a later chapter on disk utilities.

To delete a file called 'PROG', you simply enter:

```
*DELETE PROG
```

'DESTROY <afsp>

This command will delete one or more files (wild cards may be used) from a diskette. Because of the destructive nature of this command, it must be preceded by *ENABLE (DFS 0.90). Each file that matches the alternate file specification is displayed in turn on the screen. The user is asked whether or not all these files should be deleted.

For example, to delete all the files on the diskette in the current drive, you simply enter:

```
*ENABLE
*DESTROY #.*
```

and reply Y to the question following the list of files which will be deleted. *DESTROY is equivalent to a series of *DELETEs. Only the catalogue entries are actually removed.

Note that DFS 1.20 dispenses with the need for *ENABLE and prompts for confirmation before printing the filenames.

*DIR(:<drive>.)<dir>

This command sets the default general purpose drive and directory. The default general purpose drive and directory are substituted by the DFS whenever the drive and directory specifications are omitted in a command, unless the command is *RUN or one of its abbreviations (See *L1B and *RUN). The DFS reserves two memory locations to hold the default general purpose drive and directory.

&10CA = *DIR directory (general purpose)
&10CB = *D1R drive (general purpose)

These memory areas are maintained by the DFS at all times and in theory can be accessed directly by the user. The only problem in doing this will occur if you want to run your programs with a non-standard DFS, which may not allocate memory areas in the same way.

On BREAK, the DFS sets &10CB to zero and &10CA to $.

*DIR can be used to change just the default general purpose directory, or both the default general purpose drive and directory. For example:

*DIR A changes the default general purpose directory to A.

*DIR :1. A changes the default general purpose drive to 1 and the default general purpose directory to A. It is exactly the same as *DRIVE 1 followed by *DIR A.

Note that you are also allowed to enter:

```
*DIR  :1
```

This changes the default general purpose drive to 1. In Acorn DFS 0.90, it also changes the default general purpose directory back to $. Acorn DFS 1.20 leaves the default general purpose directory unchanged.

*DISC or *DISK

This command is used to change from any other filing system to the DFS. Other filing systems available on the BBC Microcomputer include:

CFS - the cassette filing system
NFS - the network filing system
RFS - the ROM filing system
TFS - the Telesoftware filing system

The *DISC (or *DISK) command initialises the disk filing system. The following description applies to Acorn DFS 0.90. Other DFS systems will perform similar steps, but the addresses will differ. The steps taken are:

shut down any open spool files.

reset some of the OS vectors, so that they point to a new routine in the OS which can re-direct the calls to the DFS ROM.

&212,&213 FILEV set to &FF1B
&214,&215 ARGSV set to &FF1E
&216,&217 BGETV set to &FF21
&218,&219 BPUTV set to &FF24
&21A,&21B GBPBV set to &FF27
&21C&21D FINDV set to &FF2A
&21E,&21F FSCV set to &FF2D

invoke the extended vector feature and write new DFS entry points in the extended vector table for each of the above OS calls.

&DBA,&DBB OSFILE entry set to &957B
&DBD,&DBE OSARGS entry set to &9007
&DC0,&DC1 OSBGET entry set to &90C1
&DC3,&DC4 OSBPUT entry set to &91AA
&DC6,&DC7 OSGBPB entry set to &95D0
&DC9,&DCA OSFIND entry set to &8E93
&DCC&DCD OSFSC entry set to &95AA

place the number of the DFS ROM in the extended vector table for each of the above OS routines.

&DBC	ROM number for OSFILE
&DBF	ROM number for OSARGS
&DC2	ROM number for OSBGET
&DC5	ROM number for OSBPUT
&DC8	ROM number for OSGBPB
&DCB	ROM number for OSFIND
&DCE	ROM number for OSFSC

notify all paged ROMS of the filing system change.

set the address on which the DFS private workspace begins (in the private workspace table). The entry in the table is normally &17, indicating that the DFS has private work space starting at &1700. This is entered into one of the locations &DF0 to &DFF, depending on the ROM number of the DFS ROM. For example, if the DFS ROM is in the normal position (ROM number &E), then &DFE will be set to &17.

issue a paged ROM call (type &0A) to demand that all other paged ROMs, relinquish their hold of any static workspace above &E00.

find out if a second processor is present and set the Tube flag (&10D7) accordingly.

reset default drives and directories (*DIR and *LIB) to drive 0 and directory $. (&10CA to &10CD).

initialise other flags.

read the disk timing parameters and initialise the 8271 with these values.

*DRIVE <drv>

This command sets the default general purpose drive to be used by all commands except *RUN (or its abbreviations), whenever a file specification omits the drive field (see *DIR for explanation). The<drv>specification must be a number in the range 0 to 3. For example, to specify drive 1 as the default general purpose drive, you simply enter:

```
*DRIVE  1
```

Note that *DRIVE changes only the default general purpose drive number; not the default machine-code library drive number specified by *LIB.

Note also that *DRIVE only checks that the argument is 0 to 3. It does not check that you actually possess a disk drive which tallies with this argument.

*DUMP <fsp>

This command will produce a hexadecimal screen display (with ASCII down the side) of a named disk file. For most files, the user should first enter CTRL/N to set page mode on. The display will then stop at the end of the screen until the user enters Shift. Page mode can subsequently be turned off by entering CTRL/O. Figure 25 shows a typical dump display (in fact it is part of this book).

```
3E38 74 68 65 20 73 63 72 65 the sere
3E40 65 6E 20 75 6E 74 69 6C en until
3E48 20 74 68 65 20 75 73 65 the use
3E50 72 20 65 6E 74 65 72 73 r enters
3E58 20 53 68 69 66 74 2E 20 Shift.
3E60 50 61 67 65 20 6D 6F 64 Page mod
3E68 65 20 63 61 6E 20 73 75 e can su
3E70 62 73 65 71 75 65 6E 74 bsequent
3E78 6C 79 20 62 65 20 74 75 ly be tu
3E80 72 6E 65 64 20 6F 66 66 rned off
3E88 20 62 79 20 65 6E 74 65 by ente
3E90 72 69 6E 67 20 43 54 52 ring CTR
3E98 4C 2F 4F 2E 0D 20 20 20 L/0..
```

Figure 25. Typical *DUMP display.

*ENABLE

This command gives DFS 0.90 advanced warning of either a *DESTROY or *BACKUP command, which can erase many files at a time. The purpose of this is to safeguard against accidentally issuing one of these commands, either by picking the wrong command (*DESTROY when you really wanted *WIPE, for example) or by executing a utility program using a bad abbreviation. For example, suppose you had a machine code program called DESPAIR. You can execute this program by entering *DESPAIR. On a bad day, you might decide to abbreviate this to *DES. Unfortunately, the system would interpret this as *DESTROY. However, no harm would be done, because you would not have first entered *ENABLE and the DFS would simply give the 'Not enabled' error message.

The DFS keeps an Enable Flag at &10C8. It is initialised to a value of &FF. When you enter *ENABLE, it simply sets this flag to 1. The flag is zeroised if the command following *ENABLE is not *BACKUP or *DESTROY (see the chapter on OSFSC for a more detailed explanation).

Whenever the DFS receives a *DESTROY or *BACKUP command, it first checks that the Enable Flag is set to 1 before carrying out the command.

*ENABLE is not necessary in DFS 1.20 as *BACKUP and *DESTROY ask a simple 'go/no go' question to achieve the same end.

*EXEC <fsp>

This command tokenises, loads and executes a specified ASCII text file. The file must contain executable BASIC or star commands. However, these commands must not be tokenised (as is the case with a normal BASIC program), but must instead be written as ASCII statements. Another way of putting this is that the file must have been either:

> created by a text editor other than the one supplied as part of BASIC, such as a word processor or *BUILD
>
> OR
>
> if created by the BASIC text editor, it must have been saved to disk by the *SPOOL command, which de-tokenises BASIC statements and reproduces the original ASCII text.

*HELP DFS

This command lists the main DFS star commands with codified descriptions of the syntax of each. See section 4.2 for a description.

*HELP UTILS

This command lists the additional DFS utility star commands with codified description of the syntax of each. See section 4.2 for a description.

*INFO <afsp>

This command lists catalogue information, for one or more files (wild cards can be used). For example, to list the catalogue entries for the whole side of the diskette in drive 2, you simply enter:

```
*INFO  :2.*
```

The catalogue is listed in reverse order of the position of the file on the disk. That is to say, it starts with the last file and ends with the first file. Figure 26 shows a typical *INFO display.

```
S.DRAM   FF1900  FF8023  000282  02E
S.RD2   FF1900  FF8023  000171  029
$%R0MDISK  FF1900  FF8023  000171  027
S.SECEDIT L FF1900 FF8023 0012EB 014
$.D0ISP     FF1900 FF8023 0011A1 002
```

Figure 26. Typical *INFO display.

Each line in the display describes a single file. The display shows the file name including the directory prefix, the lock status, the load address in hex, the execute address in hex, the size in hex bytes and the hex sector number at which the file starts. To convert this to the track and sector number at which the file starts:

track = start sector D1V 10
sector = start sector MOD 10

*LIB (:<drive>.)<dir>

This command sets the default machine-code library drive and directory. These library programs can be executed using *RUN or one of its abbreviations. The value of this is that you can insert a machine-code utility program, such as a disk editor, in one drive and make it read a diskette in another drive, without having to specify the drive and directory of either file. You use *DIR to define the general purpose drive and directory for the file to be read, and *LIB to define the machine-code library drive and directory for the utility program. For example, to define the default machine-code library drive as 2 and the directory as U, you would simply enter:

```
*LIB :2.U
```

The DFS maintains two memory locations to store these defaults:

&10CC = *LIB directory (machine-code library)
&10CD = *LIB drive (machine-code library)

These two locations are initialised by BREAK to $ and 0, respectively.

*LIB can be used to specify just the default machine-code library drive, just the default machine-code library directory or both.

*LIB :1.A

> changes the default machine-code library drive to 1 and the default machine-code library directory to A.

*LIB A

> changes the default machine-code library directory to A, leaving the default machine-code library drive unchanged.

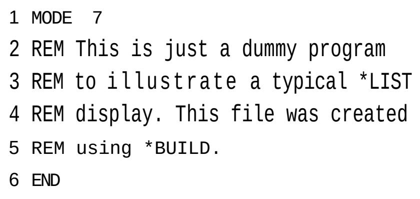
*LIB :1

> changes the default machine-code library drive to 1. In Acorn DFS 0.90, it also changes the default machine-code library directory back to $. Acorn DFS 1.20 leaves the default machine-code library directory unchanged.

*LIST <fsp>

This command displays in ASCII on the screen, the contents (including line numbers) of a single ASCII text file. It can thus be used to display a BASIC program which has been saved using *SPOOL or a file created by *BUILD. It cannot be used to display normal BASIC programs, since these contain tokens rather than ASCII text. Page mode can be used to display files larger than one screen (CTRL/N = page mode on, CTRL/O = page mode off).

Figure 27 shows a typical *LIST screen display.



```
1 MODE  7
2 REM This is just a dummy program
3 REM to illustrate a typical *LIST
4 REM display. This file was created
5 REM using *BUILD.
6 END
```

Figure 27. Typical *LIST display.

Note that in Acorn DFS 1.20, the line numbers are prefixed by leading zeroes.

*LOAD <fsp> (<hex load add>)

This command will load a specified file, regardless of format. It does not matter whether the file contains ASCII text, a BASIC program, a machine-code program or whatever. If you specify a hexadecimal load address, the specified file will be loaded at that memory address. If you omit the load address, the file will be loaded at the load address for that file contained in sector 1 of the catalogue. The screen display that accompanies *LOAD depends on the value of the *OPT 1 option.

For example, to load a file called A.TEST from drive 3 into memory starting at &2400, you simply enter:

```
*LOAD  :3.A.TEST 2400
```

*OPT <a>,<b>

These commands achieve a variety of different objectives. Here we shall only discuss those which directly influence the way in which the DFS behaves.

***OPT** 1. This option decides the amount of information that will be displayed by the DFS when loading and saving files. There is a simple choice between brief messages and extended messages. Extended messages are in *INFO format and include the lock status, load address, execute address, size and start sector for the file.

> *OPT 1,0 disables the display of extended DFS filing system messages.
>
> *OPT 1,1 enables the display of extended DFS filing system messages.

***OPT** 4. This option decides how the !BOOT file will be handled on auto-boot (i.e. when you press Shift + Break). You would use *OPT 4,3 if you had used *BUILD to create an ASCII text file called '!BOOT' which you wanted to *EXEC on auto-boot. Alternatively, if you had written a machine-code program called '!BOOT' that you wanted to *RUN on auto-boot, you would select *OPT 4,2. If, however, your '!BOOT' file was a tokenised BASIC program, you could make the auto-boot feature *LOAD this file by specifying *OPT 4,1. You would still need to enter RUN, in order to execute this program.

> *OPT 4,0 disables the auto-boot facility.
>
> *OPT 4,1 causes the file, !BOOT, to be *LOADed on auto-boot.
>
> *OPT 4,2 causes the file, !BOOT, to be *RUN on auto-boot.
>
> *OPT 4,3 causes the file, !BOOT, to be *EXECed on auto-boot.

*RENAME <old fsp> <new fsp>

This command allows you to rename a single, unlocked file. For example, to change a file name from MARK1 to MARK2, you simply enter:

```
*RENAME MARK1 MARK2
```

You may also use *RENAME to change the directory of a file. For example, to change the directory of a file called TEST from A to B, you simply enter:

```
*RENAME A.TEST B.TEST
```

Naturally you can change both directory and file name in a single operation. For example:

```
*RENAME A.MARK1 B.MARK2
```

This command simply changes the catalogue entry for the file in sector 0. It follows from this that you cannot use *RENAME to change the drive of a file, since this would involve a much more complex operation of copying the file to the new drive and deleting the original from the source drive. Note also that if the file name specified as <new fsp> already exists on the diskette, an error message, ‘File already exists', will be displayed.

*RUN <fsp> (parameters)

This command may be used to *LOAD a machine-code program at its load address and commence execution at its execute address. Both of the addresses are maintained in sector 1 of the catalogue. For example, to execute a machine-code program from drive 2 with a file name of L.PROG, you simply enter:

```
*RUN  :2.L.PROG
```

*RUN has two abbreviations, * on its own and */. Thus the above command could be abbreviated to:

```
*:2.L.PROG
```

or

```
*/:2.L.PR0G
```

Be careful with abbreviations. If the default machine-code library drive is set to 2, you may be tempted to enter '*L.PROG', but because of the search sequence, OS will think *L. means *LOAD. The purpose of the V in the third format above is to remove all such ambiguities.

You can, of course, enter:

```
*LIB :2-L *PROG
```

If the machine-code program expects parameters, these may be appended to the command. For example:

```
*PROG 1,5
```

*SAVE <fsp> <sav add> <end add+l> (<xqt add> <rel add>)

*SAVE <fsp> <sav add> +<size> (<xqt add> <rel add>)

This command will save a single file from memory to disk. You specify the file name to appear in the catalogue. The next specification is the hex memory address where the file starts. The next specification is either the hex end address in memory of the file (plus 1), or + the hex size in bytes of the file. All remaining specifications are optional. If included, they are the hex execution address of the file and the hex memory address to which this file should be re-loaded by a *LOAD command.

Any file can be *SAVEd regardless of its format, though it is primarily intended for machine-code program files.

To save a file on drive 2, which you want to call A. PROG, and which you have created in memory from &2000 to &2FFF, you simply enter:

```
*SAVE :2.A.PR0G 2000  3000
```

or

```
*SAVE :2-A.PROG 2000 +1000
```

If you want this file always to be loaded at &2000 but you want to specify that its execute address is at &2100, you would simply enter:

```
*SAVE :2.A.PR0G 2000 3000 2100
```

or

```
*SAVE :2.A.PR0G 2000 +1000 2100
```

If, however, you want *RUN to load this file at &1900 and to execute at &1A00, you would enter:

```
*SAVE :2.A.PROG 2000 3000 1A00 1900
```

or

```
*SAVE :2.A.PR0G 2000 +1000 1A00 1900
```

*SPOOL <fsp>

This command creates a single ASCII text file with the name specified by the <fsp> argument. It informs the system that whatever it displays on the screen must also be sent to the disk file. This process is terminated when the user enters *SPOOL on its own.

Suppose that you write a BASIC program, and for some reason you wish to obtain an ASCII text copy. Maybe you want to merge another ASCII text file with it, for example. Suppose you want the ASCII text file to be called A.PROG1 and you want it spooled onto drive 3. You simply enter:

*SPOOL :3.A.PROGI (to open the spool file)
LIST (to display the program on screen)
*spool (to close the spool file)

The file spooled in this way is essentially a data file, no longer directly executable by BASIC. It can, however, be executed by *EXEC, which will re-tokenise the program for you.

*TITLE <title>

This command simply gives a title to the diskette in the default general purpose drive. The title can be one to twelve bytes long and will be displayed by the *CAT command. The first 8 bytes of the title are saved in sector 0 of the catalogue and the last 4 bytes are saved in sector 1. For example, to give the current diskette a title of 'Utilities', you simply enter:

```
*TITLE Utilities
```

or

```
*TITLE "Utilities"
```

The quotation marks are normally optional, but are compulsory if the title includes one or more spaces. For example, to specify a title, 'Disk Util', you must enter:

```
*TITLE "Disk Util"
```

*TYPE <fsp>

This command is similar to *LIST in that it displays an ASCII text file on the screen. However, line numbers are omitted from the display. Note that *TYPE cannot be used to display BASIC programs, since these are not in ASCII text format. As with *LIST, it is a good idea to use page mode, if the file display is too big for a single screen (CTRL/N = page mode on, CTRL/O = page mode off). Figure 28 shows a typical TYPE display.

```
MODE 7
REM This is just a dummy program
REM to illustrate a typical *TYPE
REM display. This file was created
REM using *BUILD.
END
```

Figure 28. Typical *TYPE display.

*WIPE <afsp>

This command deletes one or more unlocked files (wild cards can be used). The command lists each unlocked file in turn which conforms with the specification, <afsp>. For each file in turn, you are asked whether or not you really want to delete it. Suppose, for example, that you want to delete five out of ten files that exist on the diskette in drive 3. You can save a lot of keyboard entries by simply entering:

```
*WIPE  :3.*
```

When the command displays a file name that you want to keep, you reply N to the question. When the command displays a file name that you want to delete, you reply Y. It is a good idea to use this technique only if there are not too many distractions around you. It can be exasperating to enter Y when you mean N. However, *WIPE does not actually delete files. It simply removes their catalogue entries and re-writes the catalogue with no gaps. The recovery program in the disk utility chapter will come to your rescue if you have this sort of problem, provided that you have neither compacted the disk, nor overwritten the deleted disk space with newly saved files.

6 Acorn DFS Internals

6.1 Acorn DFS Memory Map

This section describes the allocation of memory areas used by DFS 0.90. It assumes that no other ROMs claim workspace. There is no need to address these memory locations directly because the information can be derived by other, safer means.

&0D00 to &0D9E This area is allocated to the several NMI routines generated by the DFS. When the DFS is in dialogue with the 8271 Floppy Disk Controller, it normally expects an interrupt to result. Before initiating an 8271 FDC command, it places the appropriate interrupt handler routine in this area.

&0D9F to &0DEF This is the expanded vector set. By this mechanism, the OS ROM can pass control to specified addresses of a filing system ROM. See the description of *DISK for the extended vectors used by DFS 0.90.

&0DF0 to &0DFF This specifies, for each of the possible 16 paged ROMs, the page number on which the ROMs private work space begins. The DFS private work space begins at &1700, so one of these bytes (depending on the ROM number of the DFS ROM) will contain &17.

&0E00 to &0FFF We have already seen that &E00 to &FFF is used by the DFS to hold a copy of the disk catalogue. Remember that the DFS corrupts these areas on occasions (see *CAT for example). The catalogue (in its uncorrupted form) is listed here in its entirety. You will find this very useful if you ever want to patch a catalogue using a Disk Editor.

&0E00 to &0E07 First 8 bytes of the disk title.

&0E08 to &0E0E Filename 1.

&0E0F Directory 1.

&0E10 to &0E16 Filename 2.

&0E17 Directory 2.

&0E18 to &0E1E Filename 3.

&0E1F Directory 3.

&0E20 to &0E26 Filename 4.

&0E27 Directory 4.

&0E28 to &0E2E Filename 5.

&0E2F Directory 5.

&0E30 to &0E36 Filename 6.

&0E37 Directory 6.

&0E38 to &0E3E Filename 7.

&0E3F Directory 7.

&0E40 to &0E46 Filename 8.

&0E47 Directory 8.

&0E48 to &0E4E Filename 9.

&0E4F Directory 9.

&0E50 to &0E56 Filename 10.

&0E57 Directory 10.

&0E58 to &0E5E Filename 11.

&0E5F Directory 11.

&0E60 to &0E66 Filename 12.

&0E67 Directory 12.

&0E68 to &0E6E Filename 13.

&0E6F Directory 13.

&0E70 to &0E76 Filename 14.

&0E77 Directory 14.

&0E78 to &0E7E Filename 15.

&0E7F Directory 15.

&0E80 to &0E86 Filename 16.

&0E87 Directory 16.

&0E88 to &0E8E Filename 17.

&0E8F Directory 17.

&0E90 to &0E96 Filename 18.

&0E97 Directory 18.

&0E98 to &0E9E Filename 19.

&0E9F Directory 19.

&0EA0 to &0EA6 Filename 20.

&0EA7 Directory 20.

&0EA8 to &0EAE Filename 21.

&0EAF Directory 21.

&0EB0 to &0EB6 Filename 22.

&0EB7 Directory 22.

&0EB8 to &0EBE Filename 23.

&0EBF Directory 23.

&0EC0 to &0EC6 Filename 24.

&0EC7 Directory 24.

&0EC8 to &0ECE Filename 25.

&0ECF Directory 25.

&0ED0 to &0ED6 Filename 26.

&0ED7 Directory 26.

&0ED8 to &0EDE Filename 27.

&0EDF Directory 27.

&0EE0 to &0EE6 Filename 28.

&0EE7 Directory 28.

&0EE8 to &0EEE Filename 29.

&0EEF Directory 29.

&0EF0 to &0EF6 Filename 30.

&0EF7 Directory 30.

&0EF8 to &0EFE Filename 31.

&0EFF Directory 31.

&0F00 to &0F03 Last four bytes of disk title.

&0F04 Disk cycles, i.e. number of times the disk has been written on since it was last formatted. Note that this number cannot exceed 255. The 256th write resets it to zero.

&0F05 The number of files on this disk multiplied by eight.

&0F06 bits 7 to 6 = not used, bits 5 to 4 = boot-up option, bits 3 to 2 = not used, bits 1 to 0 = most significant two bits of the number of sectors on the disk.

&0F07 Least significant eight bits of the number of sectors on the disk.

&0F08 to &0F09 File 1 Load Address (Least significant 16 bits).

&0F0A to &0F0B File 1 Exec Address (Least significant 16 bits).

&0F0C to **&0F0D** File 1 Length in bytes (Least significant 16 bits).

&0F0E File 1 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F0F File 1 Start Sector (Least Significant 8 bits).

&0F10 to &0F11 File 2 Load Address (Least significant 16 bits).

&0F12 to &0F13 File 2 Exec Address (Least significant 16 bits).

&0F14 to &0F15 File 2 Length in bytes (Least significant 16 bits).

&0F16 File 2 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F17 File 2 Start Sector (Least Significant 8 bits).

&0F18 to &0F19 File 3 Load Address (Least significant 16 bits).

&0F1A to &0F1B File 3 Exec Address (Least significant 16 bits).

&0F1C to &0F1D File 3 Length in bytes (Least significant 16 bits).

&0F1E File 3 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F1F File 3 Start Sector (Least Significant 8 bits).

&0F20 to &0F21 File 4 Load Address (Least significant 16 bits).

&0F22 to &0F23 File 4 Exec Address (Least significant 16 bits).

&0F24 to &0F25 File 4 Length in bytes (Least significant 16 bits).

&0F26 File 4 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F27 File 4 Start Sector (Least Significant 8 bits).

&0F28 to &0F29 File 5 Load Address (Least significant 16 bits).

&0F2A to &0F2B File 5 Exec Address (Least significant 16 bits).

&0F2C to &0F2D File 5 Length in bytes (Least significant 16 bits).

&0F2E File 5 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to *6* = Start Sector.

&0F2F File 5 Start Sector (Least Significant 8 bits).

&0F30 to &0F31 File 6 Load Address (Least significant 16 bits).

&0F32 to &0F33 File 6 Exec Address (Least significant 16 bits).

&0F34 to &0F35 File 6 Length in bytes (Least significant 16 bits).

&0F36 File 6 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F37 File 6 Start Sector (Least Significant 8 bits).

&0F38 to &0F39 File 7 Load Address (Least significant 16 bits).

&0F3A to &0F3B File 7 Exec Address (Least significant 16 bits).

&0F3C to &0F3D File 7 Length in bytes (Least significant 16 bits).

&0F3E File 7 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F3F File 7 Start Sector (Least Significant 8 bits).

&0F40 to &0F41 File 8 Load Address (Least significant 16 bits).

&0F42 to &0F43 File 8 Exec Address (Least significant 16 bits).

&0F44 to &0F45 File 8 Length in bytes (Least significant 16 bits).

&0F46 File 8 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F47 File 8 Start Sector (Least Significant 8 bits).

&0F48 to &0F49 File 9 Load Address (Least significant 16 bits).

&0F4A to &0F4B File 9 Exec Address (Least significant 16 bits).

&0F4C to &0F4D File 9 Length in bytes (Least significant 16 bits).

&0F4E File 9 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F4F File 9 Start Sector (Least Significant 8 bits).

&0F50 to &0F51 File **10** Load Address (Least significant 16 bits).

&0F52 to &0F53 File 10 Exec Address (Least significant 16 bits).

&0F54 to &0F55 File 10 Length in bytes (Least significant 16 bits).

&0F56 File 10 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 '= Start Sector.

&0F57 File 10 Start Sector (Least Significant 8 bits).

&0F58 to &0F59 File 11 Load Address (Least significant 16 bits).

&0F5A to &0F5B File 11 Exec Address (Least significant 16 bits).

&0F5C to &0F5D File 11 Length in bytes (Least significant 16 bits).

&0F5E File 11 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F9F File 11 Start Sector (Least Significant 8 bits).

&0F60 to &0F61 File 12 Load Address (Least significant 16 bits).

&0F62 to &0F63 File 12 Exec Address (Least significant 16 bits).

&0F64 to &0F65 File 12 Length in bytes (Least significant 16 bits).

&0F66 File 12 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F67 File 12 Start Sector (Least Significant 8 bits).

&0F68 to &0F69 File 13 Load Address (Least significant 16 bits).

&0F6A to &0F6B File 13 Exec Address (Least significant 16 bits).

&0F6C to &0F6D File 13 Length in bytes (Least significant 16 bits).

&0F6E File 13 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F6F File 13 Start Sector (Least Significant 8 bits).

&0F70 to &0F71 File 14 Load Address (Least significant 16 bits).

&0F72 to &0F73 File 14 Exec Address (Least significant 16 bits).

&0F74 to &0F75 File 14 Length in bytes (Least significant 16 bits).

&0F76 File 14 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F77 File 14 Start Sector (Least Significant 8 bits).

&0F78 to &0F79 File 15 Load Address (Least significant 16 bits).

&0F7A to &0F7B File 15 Exec Address (Least significant 16 bits).

&0F7C to &0F7D File 15 Length in bytes (Least significant 16 bits).

&0F7E File 15 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to0 = Start Sector.

&0F7F File 15 Start Sector (Least Significant 8 bits).

&0F80 to &0F81 File 16 Load Address (Least significant 16 bits).

&0F82 to &0F83 File 16 Exec Address (Least significant 16 bits).

&0F84 to &0F85 File 16 Length in bytes (Least significant 16 bits).

&0F86 File 16 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F87 File 16 Start Sector (Least Significant 8 bits).

&0F88 to &0F89 File 17 Load Address (Least significant 16 bits).

&0F8A to &0F8B File 17 Exec Address (Least significant 16 bits).

&0F8C to &0F8D File 17 Length in bytes (Least significant 16 bits).

&0F8E File 17 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F8F File 17 Start Sector.(Least Significant 8 bits).

&0F90 to &0F91 File 18 Load Address (Least significant 16 bits).

&0F92 to &0F93 File 18 Exec Address (Least significant 16 bits).

&0F94 to &0F95 File 18 Length in bytes (Least significant 16 bits).

&0F96 File 18 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F97 File 18 Start Sector (Least Significant 8 bits).

&0F98 to &0F99 File 19 Load Address (Least significant 16 bits).

&0F9A to &0F9B File 19 Exec Address (Least significant 16 bits).

&0F9C to &0F9D File 19 Length in bytes (Least significant 16 bits).

&0F9E File 19 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0F9F File 19 Start Sector (Least Significant 8 bits).

&0FA0 to &0FA1 File 20 Load Address (Least significant 16 bits).

&0FA2 to &0FA3 File 20 Exec Address (Least significant 16 bits).

&0FA4 to &0FA5 File 20 Length in bytes (Least significant 16 bits).

&0FA6 File 20 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0FA7 File 20 Start Sector (Least Significant 8 bits).

&0FA8 to &0FA9 File 21 Load Address (Least significant 16 bits).

&0FAA to &0FAB File 21 Exec Address (Least significant 16 bits).

&0FAC to &0FAD File 21 Length in bytes (Least significant 16 bits).

&0FAE File 21 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0FAF File 21 Start Sector (Least Significant 8 bits).

&0FBO to &0FB1 File 22 Load Address (Least significant 16 bits).

&0FB2 to &0FB3 File 22 Exec Address (Least significant 16 bits).

&0FB4 to &0FB5 File 22 Length in bytes (Least significant 16 bits).

&0FB6 File 22 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0FB7 File 22 Start Sector (Least Significant 8 bits).

&0FB8 to &0FB9 File 23 Load Address (Least significant 16 bits).

&0FBA to &0FBB File 23 Exec Address (Least significant 16 bits).

&0FBC to &0FBD File 23 Length in bytes (Least significant 16 bits).

&0FBE File 23 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0FBF File 23 Start Sector (Least Significant 8 bits).

&0FC0 to &0FC1 File 24 Load Address (Least significant 16 bits).

&0FC2 to &0FC3 File 24 Exec Address (Least significant 16 bits).

&0FC4 to &0FC5 File 24 Length in bytes (Least significant 16 bits).

&0FC6 File 24 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0FC7 File 24 Start Sector (Least Significant 8 bits).

&0FC8 to &0FC9 File 25 Load Address (Least significant 16 bits).

&0FCA to &0FCB File 25 Exec Address (Least significant 16 bits).

&0FCC to &0FCD File 25 Length in bytes (Least significant 16 bits).

&OFCE File 25 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0FCF File 25 Start Sector (Least Significant 8 bits).

&0FD0 to &0FD1 File 26 Load Address (Least significant 16 bits).

&0FD2 to &0FD3 File 26 Exec Address (Least significant 16 bits).

&0FD4 to &0FD5 File 26 Length in bytes (Least significant 16 bits).

&0FD6 File 26 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0FD7 File 26 Start Sector (Least Significant 8 bits).

&0FD8 to &0FD9 File 27 Load Address (Least significant 16 bits).

&0FDA to &OFDB File 27 Exec Address (Least significant 16 bits).

&OFDC to &0FDD File 27 Length in bytes (Least significant 16 bits).

&0FDE File 27 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0FDF File 27 Start Sector (Least Significant 8 bits).

&0FE0 to &0FE1 File 28 Load Address (Least significant 16 bits).

&0FE2 to &0FE3 File 28 Exec Address (Least significant 16 bits).

&0FE4 to &0FE5 File 28 Length in bytes (Least significant 16 bits).

&0FE6 File 28 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0FE7 File 28 Start Sector (Least Significant 8 bits).

&0FE8 to &0FE9 File 29 Load Address (Least significant 16 bits).

&0FEA to &0FEB File 29 Exec Address (Least significant 16 bits).

&0FEC to &0FED File 29 Length in bytes (Least significant 16 bits).

&0FEE File 29 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 toO = Start Sector.

&0FEF Fjle 29 Start Sector (Least Significant 8 bits).

&0FF0 to &0FF1 File 30 Load Address (Least significant 16 bits).

&0FF2 to &0FF3 File 30 Exec Address (Least significant 16 bits).

&0FF4 to &0FF5 File 30 Length in bytes (Least significant 16 bits).

&0FF6 File 30 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0FF7 File 30 Start Sector (Least Significant 8 bits).

&0FF8 to &0FF9 File 31 Load Address (Least significant 16 bits).

&0FFA to &0FFB File 31 Exec Address (Least significant 16 bits).

&0FFC to &0FFD File 31 Length in bytes (Least significant 16 bits).

&0FFE File 31 Most significant 2 bits, bits 7 to 6 = Exec Address, bits 5 to 4 = Length, bits 3 to 2 = Load Address, bits 1 to 0 = Start Sector.

&0FFF File 31 Start Sector (Least Significant 8 bits).

&1000 to &11FF This is DFS general work space. It includes control block areas for operating system calls. It also contains DFS specific tags, such as:

&10C8 The Enable Flag.

&10CA The default general purpose directory.

&10CB The default general purpose drive.

&10CC The default machine-code library directory.

&10CD The default machine-code library drive.

&10D7 The Tube Present Flag.

&1100 to &11FF The Acorn DFS allows you to open up to five files simultaneously. This area contains information about the five available file buffers and their parameter blocks. Included in this overall allocation are:

&1120 to &112D File 1 Filename. Note that alternate bytes are processing bytes used by the DFS. For example, a file name of 'ABCDEFG' would be held as 'A.B.C.D.E.F.G/, where the full-stops represent processing bytes of different values.

&112E File 1 Directory.

&1130 to &1132 File 1 value of BASIC PTR# (least significant byte first).

0 File 1 pointer to buffer area page.

1 to &1136 File 1 value of BASIC EXT# (least significant byte first).

&1140 to &114D File 2 Filename. Note that alternate bytes are processing bytes. For example, a file name of 'ABCDEFG' would be held as 'A.B.CD.E.F.G.', where the full-stops represent processing bytes of different values.

&114E File 2 Directory.

&1150 to &1152 File 2 value of BASIC PTR# (least significant byte first).

0 File 2 pointer to buffer area page.

1 to &1156 File 2 value of BASIC EXT# (least significant byte first).

&1160 to &116D File 3 Filename. Note that alternate bytes are processing bytes. For example, a file name of 'ABCDEFG' would be held as 'A.B.C.D.E.F.G.', where the full-stops represent processing bytes of different values.

&116E File 3 Directory.

&1170 to &1172 File 3 value of BASIC PTR# (least significant byte first).

0 File 3 pointer to buffer area page.

1 to &1176 File 3 value of BASIC EXT# (least significant byte first).

&1180 to &118D File 4 Filename. Note that alternate bytes are processing bytes. For example, a file name of 'ABCDEFG' would be held as 'A.B.C.D.E.F.G/, where the full-stops represent processing bytes of different values.

&118E File 4 Directory.

&1190 to &1192 File 4 value of BASIC PTR# (least significant byte first).

0 File 4 pointer to buffer area page.

1 to &1196 File 4 value of BASIC EXT# (least significant byte first).

&11A0 to &11AD File 5 Filename. Note that alternate bytes are processing bytes. For example, a file name of 'ABCDEFG' would be held as 'A.B.CD.E.F.G.', where the full-stops represent processing bytes of different values.

&11AE File 5 Directory.

&11B0 to &11B2 File 5 value of BASIC PTR# (least significant byte first).

&11B3 File 5 pointer to buffer area page.

&11B4 to &11B6 File 5 value of BASIC EXT# (least significant byte first).

&1200 to &12FF Buffer area for the first opened file.

&1300 to **&13FF** Buffer area for the second opened file.

&1400 to &14FF Buffer area for the third opened file.

&1500 to &15FF Buffer area for the fourth opened file.

&1600 to &16FF Buffer area for the fifth opened file.

&1700 to &18FF Private Workspace Save Area. The DFS uses this area of memory to save its environment, if another paged ROM demands the memory area from &0E00 to &16FF.

This description of the memory areas allows us to make correct judgments about the minimum value of PAGE that we can get away with (thereby leaving more room for our program code).

Assuming that we do not switch filing systems (to TAPE, say) and no other ROM has requirements for absolute or private workspace, the normal default value for PAGE will be &1900. If however, we know in advance that we shall not switch filing systems or invoke any ROMs that will demand the absolute workspace (shared by all paged ROMs) from &0E00, then we also know that memory areas above &16FF will not be required by the DFS and we can reset PAGE to &1700. This will still allow us to open five files simultaneously. We may also know in advance that we do not require five simultaneously open files. For four open files, we can set PAGE to &1600. For three open files, we can set PAGE to &1500. For two open files we can set PAGE to &1400. For just one file, we can set PAGE to &1300. We cannot sensibly reduce PAGE below this value.

6.2 Acorn DFS as a Service ROM

The Acorn DFS is purely a service ROM; that is to say it has a service entry at &8003 but no language entry at &8000. As a service ROM it has to interface with the OS in a standard way. The OS can enter the DFS in one of two ways. Firstly it can call OSFILE, OSARGS, OSBGET, OSBPUT, OSGBPB, OSFIND or OSFSC via the extended vector mechanism as described under *DISK in chapter 5. Secondly the OS can issue a service call to the DFS. When this happens, the highest priority paged ROM is given the first opportunity to handle the service call. A ROM may choose to:

a) ignore the service call, in which case it exits with all registers preserved.

b) handle the call, but allow other service ROMs to handle it also, in which case registers are changed, but the A register is never zeroised.

c) handle the call and prevent other service ROMs from seeing it, in which case the A register is set to zero.

Note also that any paged ROM can itself issue service calls to other ROMs via OSBYTE &8F.

On entry to a ROM with a service call, the registers are set up as follows:

A - Service Type Code
X - ROM number of current ROM (and in &F4)
Y - Optional parameter defining the required service further

The OS issues service calls to each service ROM in priority sequence. Since Service Type zero means ignore the call, any service ROM which exits with the A register set to zero effectively blocks all subsequent ROMs from seeing the original call. OS 1.2 keeps a ROM Type Table starting at &2A1, with a one byte entry for each ROM, indexed by ROM number. Any ROM with a service entry has bit 7 set in this table and the OS uses this byte to determine whether or not to issue a service call to a given ROM. To issue a service call, it sends the ROM number to the ROM paging register at &FE30 and jumps to &8003. Note that service ROM code is always in the I/O processor and is never copied across the Tube.

DFS 0.90 handles the following service calls. DFS 1.20 behaves similarly, but in addition handles the &12 call to re-initialise open files temporarily on hold status.

01 Absolute Workspace Specification
On BREAK, the OS invites each service ROM in turn to specify an upper limit to the absolute workspace. The absolute workspace starts at &E00 and is only allocated to one ROM at a time. Thus the OS puts &E in the Y register before asking the first service ROM. Each service ROM examines the value of the Y register and inserts its own upper limit in the Y register, unless the Y register specifies a page greater or equal to its own upper limit. Clearly all service ROMs must preserve the A register when handling this call. The DFS simply inspects the Y register and changes it to &17 if it is less than &17 on entry. When the last service ROM has seen the service call, the OS will know the size of the absolute workspace to set aside. For example, if no other ROM changes the Y register after the DFS ROM, then the OS will set aside the memory area from &E00 to &16FF as the absolute workspace to be shared by all service ROMs, but only to be used by one ROM at a time.

02 Private Workspace Specification
On BREAK, the OS next invites each service ROM in turn to specify its requirements for private workspace which will be exclusive to that ROM. On entry, the Y register contains the next free page available for use by that ROM as private workspace. When the DFS gets it turn, it adds two to the Y register, thereby preserving 2 pages (512 bytes) of private workspace. It also stores the original value of the Y register (the start page number of its private work space) in the appropriate slot from &DF0 to &DFF. This table contains an entry for each ROM, indexed on ROM number. The DFS additionally zeroises an indicator in its private workspace

which specifies which ROM currently has possession of that space. Clearly all ROMs must preserve the A register once again.

03 Auto-boot
On BREAK, the OS issues a Service Type 3 call to each ROM in turn. As far as the DFS is concerned, the OS sets the keyboard register (&ED) and Y as follows:

&ED = &32	means disk-specific initialise (internal key for 'D', known as D-Break)
&ED = 0	means general initialise
&ED = other	means initialise not for DFS
Y = 0	means auto-boot enabled
Y = other	means auto-boot disabled

If &ED = other, the DFS knows that this is an initialise request for some other ROM and ignores the call completely. The DFS initialises if &ED = 0 or &32. It also attempts an auto-boot if Y=0. You can use this mechanism yourself to initialise (and auto-boot) the disk system. Use OSBYTEs &78 and &8F.

04 Unrecognised Star Command
The OS issues this service call to each service ROM whenever the user enters a star command which it does not recognise. On entry &F2,&F3 point to the start of the page containing the command string (terminated by carriage return). The Y register contains the offset of the string within that page. The DFS matches the command string against a table of its known star commands. If it finds a match, it simply executes that command. Note that this includes *DISK, the command which selects the DFS in the first place. If the DFS cannot match the star command, then it should simply exit with the registers preserved. If no paged ROM answers this call, then the OS will offer it to the current filing system via OSFSC (with A=3). Thus when the DFS is the current filing system it may get a second opportunity to handle an unrecognised star command. When entered via OSFSC, it treats the star command string as a filename which it attempts to *RUN.

08 Unrecognised OSWORD
When the OS receives an OSWORD call that it cannot handle, it issues this service call. On entry, the original A, X and Y

registers supplied to OSWORD are contained in &EF, &F0 and &F1 respectively. The DFS looks for values of &7D, &7E and &7F in &EF and exits for any other value, A matched value causes it to execute the appropriate DFS OSWORD function.

09 *HELP
The OS issues this service call whenever the user enters *HELP. On entry &F2,&F3 point to the start of the page containing the rest of the command string after *HELP (terminated by carriage return). The Y register contains the offset of the string within that page. In the case of the DFS it has to recognise strings of DFS and UTILS and provide the required information. If there is no characters after *HELP it simply types its name on the screen.

0A Claim Static Workspace
The DFS can receive this call from another paged ROM which needs the absolute workspace. It checks its private workspace to see if it currently holds the absolute workspace, simply exiting if not. If it does hold the absolute workspace, it copies the important memory areas in &1000 to &11FF across to the private workspace. This allows it to re-create its environment when once again it gains control of the absolute workspace. It then resets the indicator and zeroises the A register (to stop the call being passed unnecessarily to other ROMs) before exiting. It also closes any currently open file.

As well as responding to this call, the DFS also issues it to other paged ROMs whenever it requires the absolute workspace itself. It issues this call via OSBYTE &8F, very much the recommended way of doing it. It stores its own ROM number in the indicator (&17D5 in DFS 0.90) in private workspace to remind itself that it currently owns the absolute workspace.

0C Claim NMI Space
DFS 0.90 issues this call before placing an NMI interrupt handler at &D00 onwards. The call is made with Y = &FF, and any ROM that currently owns the NMI space should release it and alter Y to the value of its own ROM number. Typically you would expect the Econet ROM to work in this way. The DFS stores the value contained in Y (in &A0). Note that although the DFS issues this call itself, it does not

respond to the same call from another ROM. This is because it does not need to hog the NMl space in advance of performing a disk I/O operation. Of course, the DFS uses OSBYTE &8F to perform the service calL

0B Release NMl Space
At the end of a disk I/O, when the NMl space is no longer required, the DFS checks &A0. If this is set to a value other than &FF, it loads this value into the Y register and frees up the .NMl space via an &0B service call using OSBYTE &8F. Each ROM in turn compares its own ROM number (in X and &F4) to the Y register, and if they match, reasserts control over the NMl space. Once again, DFS 0.90 issues this call but does not respond to the same call from another ROM.

0F Vectors claimed
Whenever the DFS initialises, as soon as it has reset the OS vectors, it issues this call to notify the other paged ROMs that a filing system change is taking place.

6.3 Acorn DFS and the Tube

The Acorn DFS contains a significant amount of code to handle data I/O across the Tube for those who have a second processor.

At initialisation, the DFS checks for the existence of the Tube via OSBYTE &EA. DFS 0.90 sets an indicator at &10D7 (&FF = no Tube, &00 = Tube). When performing disk 1/Os, the DFS examines the appropriate memory address to see if I/O involves the Tube. The I/O processor is invoked by setting the top 16 bits of an address to &FFFF. If they are not &FFFF and the Tube is detected, the DFS will attempt to read or write over the Tube. When a Tube is present, the OS will have placed some code at &406 to be used by filing systems. To invoke the Tube, the filing system executes this code having first set the registers as follows:

A = 0 Read from Second Processor required
A = 1 Write to Second Processor required
A = 4 Start execution in Second Processor required

X = LSB of address of four byte control block
Y = MSB of address of four byte control block

the control block is simply the 32 bit address within the second processor.

There is also a tube register at &FEE5. When &406 is entered with A = 0, this initialises the Tube hardware/software in such a way that a LDA &FFE5 instruction reads the next byte from the second processor. Similarly, if &406 is entered with A = 1, subsequent STA &FEE5 instructions cause data to be written in the second processor. With A = 4 on entry to &406, control is passed to the second processor.

7 Acorn DFS OSFILE

The DFS provides disk-specific OSFILE support which may be called from machine-code programs. This software also supports *BUILD, CHAIN, *EXEC, *LOAD, LOAD, *SAVE, SAVE and *SPOOL; i.e. those commands which work on a whole file at a time.

OSFILE is entered at &FFDD in the OS ROM, where it is vectored by the FILEV vector (&212,213) to &FF1B, which switches to the DFS ROM and then vectors via the extended FILEV vector (&DBA,&DBB) to &957B in the DFS ROM.

The following section describes each of the available DFS OSFILE calls in turn. You pick the OSFILE feature that you want by loading the correct argument into the A register. Other arguments needed by OSFILE are supplied by you in an eighteen byte long parameter block, the address of which you store in the X and Y registers. You store the filename on which OSFILE is to work, terminated by &0D, in a separate string. The first two bytes of the parameter block point to this string. Those commands that expect to find the specified file already on the diskette return the 'File not found' message if this is not the case.

A demonstration program accompanies each description. Since OSFILE is intended for use by machine-code programs, the demonstration programs are all in machine-code also. Chapter 14 explains how to call DFS routines from BASIC.

OSFILE   Save Memory Block

on entry:

A = 0
X = LSB of address of param block
Y = MSB of address of param block
param block &00 - &01 = address of filename string
param block &02 - &05 = load address
param block &06 - &09 = execute address
param block &0A - &0D = start address to save from
param block &0E - &11 = end address + 1 to save from

(all addresses are LSB first)

action:

OSFILE saves the specified memory block. It creates a catalogue entry using the file name pointed to in the param block, and stores the relevant addresses in sector 1 of the catalogue.

on exit:

A = destroyed
X = unchanged
Y = unchanged
P = destroyed

```
   10 REM -----------------------------------------------------
   20 REM     DEMONSTRATION
   30 REM -----------------------------------------------------
   40 REM OSFILE Save Memory Block
   50 REM -----------------------------------------------------
   60
   70 REM This program assembles the code below, and then saves
   80 REM just the machine-code bits of itself to a file, "Myself".
   90
  100 DIM code% &100
  110 osfile = &FFDD
  120 FOR pass% = 0 TO 2 STEP 2
  130 P% = code%
  140 [ OPT pass%
  150 \    parameter block
  160 .parms EQUW fname   \ address of filename string
  170       EQUD parms    \ load address
  180       EQUD start    \ execute address
  190       EQUD parms    \ save start address
  200       EQUD end      \ save end address
  210 .parad EQUW parms   \ address of param block
  220 .fname EQUS "Myself" \ filename string
  230       EQUB &D       \ termination
  240
  250 .start LDA #0       \ specify save
  260       LDX parad     \ point X and Y
  270       LDY parad+1 \ at parms
  280       JSR osfile    \ call OSFILE
  290       RTS          \ bye bye
  300 .end  EQUB 0        \ dummy byte
  310 ]
  320 NEXT pass%
  330 CALL start
  340 *INFO Myself
  350 END
```

>RUN

```
$.Myself  001CE2 001CFD 000027 006
```

OSFILE   Update Catalogue Entry (Addresses Only)

on entry:

A = 1
X = LSB of address of param block
Y = MSB of address of param block
param block &00 - &01 = address of filename string
param block &02 - &05 = load address param block &06 -
&09 = execute address param block &0A - &11 = does
not matter

(all addresses are LSB first)

action:

OSFILE updates the catalogue with the contents of the param block.

on exit:

A = destroyed
X = unchanged
Y = unchanged
P = destroyed

```
   10 REM
   20 REM     DEMONSTRATION
   30 REM -------------------------------------------------------
   40 REM OSFILE Update Catalogue Entry
   50 REM -------------------------------------------------------
   60
   70 REM This program changes the catalogue entries for "Myself".
   80 REM Change load address to &1900.
  90 REM Change execute address to &191B.
  120
  130 DIM code% &100
  140 osfile = &FFDD
  150 FOR pass% = 0 TO 2 STEP 2
  160 P% = code%
  170 [ OPT pass%
  180 \    parameter block
  190 .parms EQUW fname   \ address of filename string
  200      EQUD &1900   \ new load address
  210      EQUD &191B   \ new execute address
  220      EQUD 0      \
  230      EQUD 0      \
  240 .parad EQUW parms   \ address of param block
  250 .fname EQUS "Myself" \ filename string
  260      EQUB &D     \ termination
  270
  280 .start LDA #1     \ specify update cat
  290      LDX parad   \ point X and Y
  300      LDY parad+1 \ at parms
  310      JSR osfile  \ call OSFILE
  320      RTS        \ bye bye
  330 ]
  340 NEXT pass%
  350 CALL start
  360 *INFO Myself
  370 END

>RUN
S.Myself   001900 00191B 000027 006
```

OSFILE   Update Catalogue With Load Address Only

on entry:

A = 2
X = LSB of address of pa ram block
Y = MSB of address of param block
param block &00 - &01 = address of filename string
param block &02 - &05 = load address
param block &06 - &11 = does not matter

(all addresses are LSB first)

action:

OSFILE updates the catalogue with the new load address supplied in the param block.

on exit:

A = destroyed
X = unchanged
Y = unchanged
P = destroyed

```
10 REM
20 REM     DEMONSTRATION
30 REM
40 REM OSFILE Update Load Address
50 REM
60
70 REM This program changes the load address only for "Myself"
80 REM It changes it to &5000.
90
100 DIM code% 6100
110 osfile = &FFDD
120 FOR pass% = 0 TO 2 STEP 2
130 P% = code%
140 [  OPT pass%
150 \    parameter block
160 .parms EQUW fname   \ address of filename string
170      EQUD &5000   \ new load address
180      EQUD 0      \
190      EQUD 0      \
200      EQUD 0      \
210 .parad EQUW parms   \ address of param block
220 .fname EQUS "Myself" \ filename string
230      EQUB &D     \ termination
240
250 .start LDA #2     \ specify load address
260      LDX parad   \ point X and Y
270      LDY parad+1 \ at parms
280      JSR osfile  \ call OSFILE
290      RTS        \ bye bye
300 ]
310 NEXT pass%
320 CALL start
330 *INFO Myself
340 END
>RUN
S.Myself    005000 00191B 000027 006
```

OSFILE   Update Catalogue With Execute Address Only

on entry:

A = 3
X = LSB of address of param block
Y = MSB of address of param block
param block &00 - &01 = address of filename string
param block &02 - &05 = does not matter
param block &06 - &09 = execute address
param block &0A - &11 = does not matter

(all addresses are LSB first)

action:

OSFILE updates the catalogue with the new execute address supplied in the param block.

on exit:

A = destroyed
X = unchanged
Y = unchanged
P = destroyed

```
   10 REM  ------------------------------------------------------
   20 REM     DEMONSTRATION
   30 REM  ------------------------------------------------------
   40 REM OSFILE Update Execute Address
   50 REM  ------------------------------------------------------
   60
   70 REM This program changes the execute address only for "Myself*.
   80 REM It changes it to &501B.
   90
  100 DIM code% &100
  110 osfile = &FFDD
  120 FOR pass% = 0 TO 2 STEP 2
  130 P% = code%
  140 [ OPT pass%
  150 \    parameter block
  160 .parms EQUW fname   \ address of filename string
  170      EQUD 0      \
  180      EQUD &501B   \ new execute address
  190      EQUD 0      \
  200      EQUD 0      \
  210 .parad EQUW parms   \ address of param block
  220 .fname EQUS "Myself" \ filename string
  230      EQUB &D     \ termination
  240
  250 .start LDA #3     \ specify xqt address
  260      LDX parad   \ point X and Y
  270      LDY parad+1 \ at parms
  280      JSR osfile  \ call OSFILE
  290      RTS        \ bye bye
  300 ]
  310 NEXT pass%
  320 CALL start
  330 *INFO Myself
  340 END

>RUN
$.Myself   005000 00501B 000027 006
```

OSFILE   Update Catalogue With Lock Status Only

on entry:

A = 4
X = LSB of address of param block
Y = MSB of address of param block
param block &00 - &01 = address of filename string
param block &02 - &0D = does not matter
param block &0E - lock status (0 = unlocked, &A = locked)
param block &0F - &11 = does not matter

(all addresses are LSB first)

action:

OSFILE updates the catalogue entry for the specified file with the lock status bit supplied in the param block.

on exit:

A = destroyed
X = unchanged
Y '= unchanged
P = destroyed

```
10 REM
20 REM     DEMONSTRATION
30 REM
40 REM OSFILE Update Lock Status
50 REM
60
70 REM This program changes the lock status only for "Myself".
80 REM It changes it to locked.
90
100 DIM code% &100
110 osfile = &FFDD
120 FOR pass% = 0 TO 2 STEP 2
130 P% = code%
140 [ OPT pass%
150 \    parameter block
160 .parms EQUW fname   \ address of filename string
170      EQUD 0      \
180      EQUD 0      \
190      EQUD 0      \
200      EQUB &A     \ new lock status
210      EQUD 0      \
220 .parad EQUW parms   \ address of param block
230 .fname EQUS "Myself" \ filename string
240      EQUB &D     \ termination
250
260 .start LDA #4     \ specify lock status
270      LDX parad   \ point X and Y
280      LDY parad+1 \ at parms
290      JSR osfile  \ call OSFILE
300      RTS        \ bye bye
310 ]
320 NEXT pass%
330 CALL start
340 *INFO Myself
350 END
>RUN

S.Myself L 005000 00501B 000027 006
```

OSFILE   Read a File's Catalogue Entry

on entry:

A = 5
X = LSB of address of param block
Y = MSB of address of param block
param block &00 - &01 = address of filename string
param block &02 - &11 = does not matter

action:

OSFILE reads the catalogue entries for the specified file into the param block.

on exit:

A = destroyed
X = unchanged
Y = unchanged
P = destroyed
param block &02 - &05 = load address
param block &06 - &09 = execute address
param block &0A - &0D = length of file in bytes
param block &0E = lock status (0=unlocked, &A = locked)

(all addresses are LSB first)

```
10 REM ----------------------------------------------
20 REM     DEMONSTRATION
30 REM ----------------------------------------------
40 REM OSFILE Read Catalogue For File
50 REM ----------------------------------------------
60
70 REM This program reads the catalogue entry for "Myself".
80 REM It then prints it.
90
100 DIM code% &100
110 osfile = &FFDD
120 FOR pass% = 0 TO 2 STEP 2
130 P% = code%
140 [ OPT pass%
150 \   parameter block
160 .parms EQUW fname   \ address of filename string
170     EQUD 0     \ DFS writes load address
180     EQUD 0      \ DFS writes exec address
190     EQUD 0      \ DFS writes file size
200     EQUB 0     \ DFS writes lock status
210     EQUD 0     \
220 .parad EQUW parms   N address of param block
230 .fname EQUS "Myself" \ filename string
240      EQUB &D     \ termination
250
260 .start LDA #5     \ specify read cat
270      LDX parad   \ point X and Y
280      LDY parad+1 \ at parms
290      JSR osfile  \ call OSFILE
300     RTS        \ bye bye
310 ]
320 NEXT pass%
330 CALL start
340 PRINT "load address = ";~parms!2
350 PRINT " xqt address = ";~parms!6
360 PRINT "size of file = ";~parms!10
370 PRINT " lock status = ";~parms?14
380 END

>RUN
load address = 5000
 xqt address = 501B
size of file = 27
 lock status = A
```

OSFILE   Delete Specified File

on entry:

A = 6
X = LSB of address of param block
Y = MSB of address of param block
param block &00 - &01 = address of filename string
param block &02 - &11 = does not matter

action:

OSFILE deletes the specified file's catalogue entry.

on exit:

A = destroyed
X = unchanged
Y = unchanged
P = destroyed

```
 10 REM
 20 REM    DEMONSTRATION
 30 REM
 40 REM OSFILE Delete File
 50 REM
 60
 70 REM This program deletes the file, "Myself".
 80
 90 DIM code% &100
100 osfile = &FFDD
110 FOR pass% = 0 TO 2 STEP 2 120 P% = code%
130 [ OPT pass%
140 \    parameter block
150 .parms EQUU fname   \ address of filename string
160      EQUD 0      \
170      EQUD 0      \
180      EQUD 0      \
190      EQUD 0      \
200 .parad EQUW parms   \ address of param block
210 .fname EQUS "Myself" \ filename string
220      EQUB &D     \ termination
230
240 .start LDA #6     \ specify delete
250      LDX parad   \ point X and Y
260      LDY parad+1 \ at parms
270      JSR osfile  \ call OSFILE
280     RTS        \ bye bye
290 ]
300 NEXT pass%
310 CALL start
320 *INFO Myself 330 END
```

```
>RUN


File not found at  line 380
```

OSFILE   Load Specified File

on entry:

A = &FF
X = LSB of address of param block
Y = MSB of address of param block
param block &00 - &01 = address of filename string
param block &02 - &05 = load address
param block &06 = load indicator
    (0 = use load address in this block, non-zero = use load address in catalogue)
param block &07 - &11 = does not matter

(all addresses are LSB first)

action:

OSFILE loads the specified file.

on exit:

A = destroyed
X = unchanged
Y = unchanged
P = destroyed

```
10 REM
20 REM     DEMONSTRATION
30 REM
40 REM OSFILE Load File
50 REM
60
70 REM This program loads "Myself" at its default
80 REM load address of &5000.
90
100 DIM code% &100
110 osfile = &FFDD
120 FOR pass% = 0 TO 2 STEP 2
130 P% = code%
140 [ OPT pass%
150 \    parameter block
160 .parms EQUW fname   \ address of filename string
170      EQUD 0      \
180      EQUB 1      \ Load flag
190      EQUD 0      \
200      EQUD 0      \
210      EQUD 0      \
220 .parad EQUW parms   \ address of param block
230 .fname EQUS "Myself" \ filename string
240      EQUB &D     \ termination
250
260 .start LDA #&FF    \ specify load
270      LDX parad   \ point X and Y
280      LDY parad+1 \ at parms
290      JSR osfile  \ call OSFILE
300      RTS        \ bye bye
310 ]
320 NEXT pass%
330 *OPT 1,2
340 CALL start
350 END

>RUN
>$.Myself    005000 00501B 000027 006
```

8 Acorn DFS OSARGS

The DFS provides disk-specific OSARGS support which may be called from machine-code programs. This software also supports the BASIC commands, EXT# and PTR#.

OSARGS is entered at &FFDA in the OS ROM where it is vectored by the ARGSV vector (&214,215) to &FF1E, which switches to the DFS ROM and then vectors via the extended ARGSV vector (&DBD,&DBE) to &9007 in the DFS ROM.

The following section describes each of the available DFS OSARGS calls in turn. The OSARGS feature is selected by loading the correct arguments into the A and Y registers. You point the X register to a four byte control block in page zero, through which you communicate with OSARGS.

In general, you will either set the Y register to zero for a function which is not specific to a single file, or alternatively you will set the Y register to the file handle to identify the file to be processed. Each of the five possible files is given a different handle at open time by the system. The file handles in DFS 0.90 and 1.20 are shown below, but these could change in later releases.

&11= 1st file handle
&12= 2nd file handle
&13= 3rd file handle
&14= 4th file handle
&15= 5th file handle

As you will see later, OSFIND can be used to open a file and provide you with the correct file handle. The terms 'channel' and 'file handle' appear to be interchangeable. Certainly if you make a mistake with the file handle, the DFS responds with a 'Channel' message.

Each of the calls is accompanied by some demonstration code. As the DFS routines are intended for use in machine-code programs, the demonstration programs are also in machine-code. Chapter 14 explains how to call these routines from BASIC.

OSARGS Get Current Filing System

on entry:

A = 0
X = does not matter
Y = 0

action:

OSARGS establishes which, if any, filing system is currently operating. It returns a code in the A register as follows:

A = 0	no filing system
A = 1	1200 baud cassette
A = 2	300 baud cassette
A = 3	ROM filing system
A = 4	Disk filing system
A = 5	ECONET filing system
A = 6	Telesoftware system

on exit:

A = filing system code
X = unchanged
Y = unchanged
P = destroyed

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSARGS Current Filing System
50 REM
60
70 DIM code% &100
80 osargs = &FFDA
90 FOR pass% = 0 TO 2 STEP 2
100 P% = code%
110 [OPT pass%
120 .start LDA #0      \ specify get
130       TAY  \ filing system
140       JSR osargs    \ call OSARGS
150      STA &70      \ save filing system
160      RTS   \ bye bye
170 ]
180 NEXT pass%
190 ?&70 = 0
200 CALL start
210 IF ?&70 = 4 PRINT "success"
220 END

>RUN

success
```

OSARGS Get Address of *RUN Parameters

on entry:

A = 1
X = address of control block
Y = 0

action:

OSARGS locates the address of any parameters which followed a *RUN command, and returns this address in the control block pointed to by X. The address is least significant byte first.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
control block = address of *RUN parameters

```
10 REM ----------------------------------------
20 REM                DEMONSTRATION
30 REM ----------------------------------------
40 REM OSARGS *RUN parms
60
70 DIM code% &100
80 osargs = &FFDA
90 FOR pass% = 0 TO 2 STEP 2
100 P%  = code%
110 [ OPT pass%
120 .start LDA #1       \  specify *RUN parms
130        LDX #&70     \ put in &70,&71
140        LDY #0       \  handle zero
150        JSR osargs   \ call OSARGS
160        RTS          \ bye bye
170 ]
180 NEXT pass%
190 REM create a machine code program
200 REM called "test", which consists
210 REM only of RTS.
220
230 ?&5000 = &60
240 *SAVE test 5000 +1 5000
250
260 REM *RUN it, just to introduce a
270 REM parameter = PARM
280
290 *RUN test PARM
300
310 REM CALL start to get address of
320 REM parameters in &70,&71.
330
340 CALL start
350
360 REM now print what's in there.
370
380 I%  = ?&70 + 256*?&71 - 1
390 REPEAT
400 I%  = I% + 1
410 PRINT CHR$(?I%)
420 UNTIL ?I% < &20
430 END

RUN
P
A
R
M
```

OSARGS Do Outstanding Disk Updates (All Files)

on entry:

A = &FF
X = does not matter
Y = 0

action:

OSARGS establishes which files are currently open for output. For each open output file, it checks to see if the file buffer contains data that has not yet been written to the diskette and, if so, sends this data to the disk.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed

```
10 REM
20 REM     DEMONSTRATION
30 REM
40 REM OSARGS Do Disk Updates
50 REM
60
70 DIM code% &100
80 osargs = &FFDA
90 FOR pass% = 0 TO 2 STEP 2
100 P% = code%
110 [ OPT pass%
120 .start LDA #&FF     \ specify disk updates
130        LDY #0       \  handle zero
140        JSR osargs   \ call OSARGS
150        RTS          \ bye bye
160 ]
170 NEXT pass%
180 CALL start
190 REM I cannot think of a way of
200 REM demonstrating this call, but
210 REM this is how you code it.
220 END
```

OSARGS Read PTR#

on entry:

A = 0
X = address of control block
Y = file handle

action:

OSARGS gets the value of PTR# for the specified file and transfers it, least significant byte first, to the control block pointed to by X.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
control block = PTR#

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSARGS Read PTR#
50 REM
60
70 DIM code% &100
80 osargs = &FFDA
90 FOR pass% = 0 TO 2 STEP 2
100 P% = code%
110 [ OPT pass%
120 .start LDA #0       \ specify read PTR#
130        LDX #&70     \ put PTR# in &70-&73
140        LDY &80      \ handle
150        JSR osargs   \ call OSARGS
160        RTS          \ bye bye
170 ]
180 NEXT pass%
190 REM open a file and write 12 bytes
200 ?&80 = OPENOUT("test")
210 FOR I% = 1 TO 12
220    BPUT# ?&80,&20
230 NEXT
240 REM we expect PTR# to be 12
250 REM let's find out.
260 CALL start
270 PRINT "PTR = ";!&70
280 CLOSE# ?&80
290 END

>RUN
PTR =12
```

OSARGS Write PTR#

on entry:

A = 1
X = address of control block
Y = file handle
control block = new value for PTR#, least significant byte first

action:

OSARGS transfers the new value of PTR# for the specified file to the control block pointed to by X.

on exit:
A = unchanged
X = unchanged
Y = unchanged
P = destroyed

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSARGS Write PTR#
50 REM
60
70 DIM code% &100
80 osargs = &FFDA
90 FOR pass% = 0 TO 2 STEP 2
100 P% = code%
110 [ OPT pass%
120 .start LDA #1        \ specify write PTR#
130        LDX #&70      \ get PTR# from &70-&73
140        LDY &80       \ handle
150        JSR osargs    \ call OSARGS
160        RTS           \ bye bye
170 3
180 NEXT pass%
190 REM open a file
200 ?&80 = OPENUP("test")
210 REM get OSARGS to set PTR to 5
215 !&70 = 5
220 CALL start
230 REM let's see if it worked
240 PRINT "PTR = ";PTR# ?&80
250 CLOSE# ?&80
260 END

>RUN
PTR = 5
```

OSARGS Read EXT#

on entry:

A = 2
X = address of control block
Y = file handle

action:

OSARGS gets the value of EXT# for the specified file and transfers it, least significant byte first, to the control block pointed to by X.

on exit:
A = unchanged
X = unchanged
Y = unchanged
P = destroyed
control block = EXT#

```
10 REM
20 REM     DEMONSTRATION
30 REM
40 REM OSARGS Read EXT#
50 REM
60
70 DIM code% &100
80 osargs = &FFDA
90 FOR pass% = 0 TO 2 STEP 2
100 P% = code%
110 [ OPT pass%
120 .start LDA #2      \ specify read EXT#
130       LDX #&70     \ put EXT# in &70-&73
140       LDY &80      \ handle
150       JSR osargs    \ call OSARGS
160      RTS   \ bye bye
170 ]
180 NEXT pass%
190 REM open a file and write 12 bytes
200 ?&80 = OPENOUT("test")
210 FOR I% = 1 TO 12
220    BPUT# ?&80,&20
230 NEXT
240 REM we expect EXT# to be 12.
250 REM let's find out.
260 CALL start
270 PRINT "EXT = ";!&70
280 CLOSE# ?&80
290 END

>RUN
EXT = 12
```

OSARGS Do Outstanding Disk Updates (One File)

on entry:

A = &FF
X = does not matter
Y = file handle

action:

OSARGS checks to see if the specified file is open for output and if its buffer contains data that has not yet been written to the diskette. If so, OSARGS sends this data to the disk.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSARGS Do A Disk Update
50 REM
60
70 DIM code% &100
80 osargs = &FFDA
90 FOR pass% = 0 TO 2 STEP 2
100 P% = code%
110 [ OPT pass%
120 .start LDA #&FF     \ specify a disk update
130       LDY #&11     \ handle
140       JSR osargs    \ call OSARGS
150       RTS  \ bye bye
160 ]
170 NEXT pass%
180 CALL start
190 REM I cannot think of a way of
200 REM demonstrating this call, but
210 REM this is how you code it for
220 REM file handle 811.
220 END
```

9 Acorn DFS OSBGET/ OSBPUT/OSGBPB

9.1 Acorn DFS OSBGET

The DFS provides disk-specific OSBGET support which may be called from machine-code programs. This software also supports the BASIC command, BGET#.

OSBGET is entered at &FFD7 within the OS ROM and is then vectored by the BGETV vector (&216,217) to &FF21, which switches to the DFS ROM and then vectors via the extended BGETV vector (&DC0,&DC1) to &90C1 in the DFS ROM.

OSBGET reads a single byte from a specified disk file. You specify which file you want by placing the file handle in the Y register. See OSARGS for a description of file handles. Note that the file must first be opened for input or update (use OSFIND). The value of PTR# determines which byte is to be read from the file. OSBGET returns the byte read in the A register.

A demonstration program, in machine-code, is supplied.

OSBGET Read Data Byte

on entry:

A = does not matter
X = does not matter
Y = file handle

action:

OSBGET uses PTR# for the specified file to determine the next byte to be read. It places this byte in A.

on exit:

A = byte read
X = unchanged
Y = unchanged
P = destroyed

9.2 Acorn DFS OSBPUT

The DFS provides disk-specific OSBPUT support which may be called from machine-code programs. This software also supports the BASIC command, BPUT#.

OSBPUT is entered at &FFD4 within the OS ROM and is vectored by the BPUTV vector (&218,219) to &FF24, which switches to the DFS ROM and then vectors via the extended BPUTV vector (&DC3,&DC4) to &91AA in the DFS ROM.

OSBPUT writes a single byte to a specified disk file. You specify which file you want by placing the file handle m the Y register. See OSARGS for a description of file handles. Note that the file must first be opened for output or update (use OSFIND). The value of PTR# determines where in the file to write the byte. You place the byte to be written in the A register.

A demonstration program, in machine-code, is supplied.

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSBGET read one byte
50 REM
60 osbget = &FFD7
70 DIM code% &100
80 FOR pass% = 0 TO 2 STEP 2
90 P% = code%
100  OPT pass%
110 .start  LDY &80      \ get handle
120         JSR osbget   \ read byte
130         STA &70      \ save byte read
140         RTS         \ bye bye
150 ]
160 NEXT pass%
170 REM open a file, "test" and write "A"
180 REM to it. close it. re-open for input.
190 REM call OSBGET to read the byte.
200 REM close file again. Display byte read.
210
220 ?&80 = OPENOUT("test")
230 BPUT# ?&80,65
240 CLOSE# ?&80
250 ?&80 = OPENUP("test")
260 CALL start
270 CL0SE# ?&80
280 PRINT "byte read = ";CHR$(?&70)
290 END

>RUN
byte read = A
```

OSBPUT Write Data Byte

on entry:

A = data byte to be written
X = does not matter
Y = file handle

action:

OSBPUT uses PTR# for the specified file to determine where to write the next byte. It writes the byte in A to this position.

on exit:
A = unchanged
X = unchanged
Y = unchanged
P = destroyed

9.3 Acorn DFS OSGBPB

The DFS provides disk-specific OSGBPB support which may be called from machine-code programs. This software also supports the BASIC commands, INPUT# and PRINT#.

OSGBPB is entered at &FFD1 within the OS ROM and is vectored by the GBPBV vector (&21A,21B) to &FF27, which switches to the DFS ROM and then vectors via the extended GBPBV vector (&DC6,&DC7) to &95D0 in the DFS ROM.

OSGBPB reads or writes several bytes at a time. It also can perform other utility functions. You specify which function you want in the A register. You communicate with OSGBPB through a thirteen byte long control block, and you point the X and Y registers at this control block. The first byte of the control block is always a file handle (see OSARGS) for those functions which read or write specific file data.

Note that the read/write functions of OSGBPB overwrite the control block. Memory addresses are incremented as each successive byte is read/written and pointers are decremented to zero. In effect OSGBPB uses the control block as its own working area.

Machine-code demonstration programs are supplied for each of these calls. For use within BASIC see Chapter 14.

Unlike the NFS version of OSGBPB, which goes a long way towards minimising the packet overhead, DFS 0.90 OSGBPB invokes OSBGET/OSBPUT in its implementation. Thus the ordinary data I/O calls via OSGBPB are no faster than the equivalent number of OSBGET or OSBPUT calls. Thus such usage of OSGBPB is largely useless.

```
10 REM
20 REM       DEMONSTRATION
30 REM
40 REM OSBPUT write one byte
50 REM
60 osbput = &FFD4
70 DIM code% &100
80 FOR pass% = 0 TO 2 STEP 2
90 P% = code%
100 [ OPT pass%
110 .start LDY &80   \ get handle
120        LDA #65   \ byte = "A"
130        JSR osbput \ write byte
140        RTS \ bye bye
150 ]
160 NEXT pass%
170 REM open a file, "test" and call OSBPUT
180 REM to write "A" to it. close file.
190 REM re-open for input, read the byte.
200 REM close file again. Display byte read.
210
220 ?&80 = OPENOUT("test")
230 CALL start
240 CLOSE# ?&80
250 ?&80 = OPENUP("test")
260 A% = BGET# ?&80
270 PRINT "byte read = ";CHR$(A%)
280 CLOSE# ?&80
290 END

>RUN
byte read = A
```

OSGBPB Write Block Using User Pointer

on entry:

A = 1
X = LSB of address of control block
Y = MSB of address of control block
control block &00 = file handle
control block &01 to &04 = memory address where data is
control block &05 to &08 = number of bytes to write control
block &09 to &0C = sequential pointer to use

(control block entries are least significant byte first)

action:

OSGBPB updates PTR# with the user pointer in the control block and uses PTR# to identify where on disk to write. It uses the control block to identify the specified file, and to write the required number of bytes from the specified memory location to the disk.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
control block overwritten

```
10 REM
20 REM       DEMONSTRATION
30 REM
40 REM OSGBPB write block using user ptr
50 REM
60 osgbpb = &FFD1
70 DIM code% &100
80 FOR pass% = 0 TO 2 STEP 2
90 P% = code%
100 [ OPT pass%
110 .data   EQUS "testing" \
120 .conb   EQUB 0      \ handle
130         EQUD data    \ point to data
140         EQUD conb-data \ size
150         EQUD 5       \ user ptr
160 .conad  EQUW conb    \ address of con block
170 .start  LDA &80      \ get handle
180         STA conb     \ put in con block
190         LDA #1       \ function
200         LDX conad    \ point X and Y
210         LDY conad+1   \ at con block
220         JSR osgbpb   \ write block of data
230         RTS       \ bye bye
240 ]
250 NEXT pass%
260 REM open a file, "test" and write
270 REM "I am thirsty" to it. call OSGBPB
280 REM to write "testing" to it, starting
290 REM at byte 5. see what's there.
300
310 A$  = "I am thirsty"
320 ?&80 = OPENOUT("test")
330 FOR I% =  1 TO LEN(A$)
340    BPUT# ?&80,ASC(MID$(A$,I%,1))
350 NEXT
360 CALL start
370 CLOSE# ?&80
380 *DUMP test
390 END

>RUN
I am testing....
```

(NOTE: the hex listing part of *DUMP has been omitted for clarity).

OSGBPB Write Block Using PTR#

on entry:

A = 2
X = LSB of address of control block
Y = MSB of address of control block
control block &00 = file handle
control block &01 to &04 = memory address where data is
control block &05 to &08 = number of bytes to write
control block &09 to &0C = does not matter

(control block entries are least significant byte first)

action:

OSGBFB uses the control block to identify the specified file, PTR# to establish where on the disk to write, and the control block to write the required number of bytes from the specified memory location to the disk.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
control block overwritten

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSGBPB write block using PTR#
50 REM
60 osgbpb = &FFD1
70 DIM code% &100
80 FOR pass% = 0 TO 2 STEP 2
90 P% = code%
100 [ OPT pass%
110 .data   EQUS ", really."
120 .conb   EQUB 0      \ handle
130         EQUD data    \ point to data
140         EQUD conb-data \ size
150         EQUD 0       \
160 .conad  EQUW conb    \ address of con block
170 .start  LDA &80      \ get handle
180         STA conb     \ put in con block
190         LDA #2       \ function
200         LDX conad    \ point X and Y
210         LDY conad+1   \ at con block
220         JSR osgbpb    \ write block of data
230         RTS       \ bye bye
240 ]
250 NEXT pass%
260 REM open a file, "test" and write
270 REM "I am thirsty" to it.
280 REM set PTR to 12. call OSGBPB
290 REM to write ", really." to it, starting
300 REM at PTR. see what's there.
310
320 A$  = "I am thirsty"
330 ?&80 = OPENOUT("test")
340 FOR I% = 1 TO LEN(A$)
350    BPUT# ?&80,ASC(MID$(A$,I%,1))
360 NEXT
370 PTR# ?&80 = 12
380 CALL start
390 CLOSE# ?&80
400 *DUMP test
410 END

>RUN
I am thirsty, really....
```

(NOTE: the hexadecimal part of *DUMP has been omitted for clarity).

OSGBPB Read Block Using User Pointer

on entry:

A = 3
X = LSB of address of control block
Y = MSB of address of control block
control block &00 = file handle
control block &01 to &04 = memory address to put data
control block &05 to &08 = number of bytes to read
control block &09 to &0C = sequential pointer to use

(control block entries are least significant byte first)

action:

OSGBPB updates PTR# with the user pointer in the control block and uses this to determine where on disk to read. It uses the control block to identify the specified file, and to read the required number of bytes from the disk into the specified
memory location.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
control block overwritten

```
10 REM
20 REM  DEMONSTRATION
30 REM
40 REM OSGBPB read block using user ptr
50 REM
60 osgbpb = &FFD1
70 DIM code% &100
80 FOR pass% =0 TO 2 STEP 2
90 P% =  code%
100 [ OPT pass%
110 .conb   EQUB 0       \ handle
120         EQUD &70     \ point to data
130         EQUD 7       \ size
```

```
 140        EQUD 5       \ user ptr
 150 .conad  EQUW conb   \ address of con block
 160 .start  LDA &80     \ get handle
 170       STA conb      \ put in con block
 180       LDA #3        \ function
 190       LDX conad     \ point X and Y
 200       LDY conad+1   \ at con block
 210       JSR osgbpb    \ read block of data
 220       RTS           \ bye bye
 230 ]
 240 NEXT pass%
 250 REM open a file, "test" and write
 260 REM "I am thirsty" to it. close.
 270 REM re-open and call OSGBPB
 280 REM to read 7 bytes starting
 290 REM at byte 5 into &70 to &76.
 300 A$  = "I am thirsty"
 310 ?&80 = OPENOUT("test")
 320 FOR I% = 1 TO LEN(A$)
 330    BPUT# ?&80,ASC(MID$(A$,I%,1))
 340 NEXT
 350 CLOSE# ?&80
 360 ?&80 = OPENUP("test")
 370 CALL start
 380 CLOSE# ?&80
 390 FOR I% = &70 TO &76
 400    PRINT CHR$(?I%)
 410 NEXT
 420 END
```

```
>RUN
t
h
i
r
s
t
y
```

OSGBPB Read Block Using PTR#

on entry:

A = 4
X = LSB of address of control block
Y = MSB of address of control block
control block &00 = file handle
control block &01 to &04 = memory address to put data
control block &05 to &08 = number of bytes to read
control block &09 to &0C = does not matter

(control block entries are least significant byte first)

action:

OSGBPB uses the the control block to identify the specified file, PTR# to establish where on the disk to read, and the control block to read the required number of bytes from the disk Into the specified memory location.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
control block overwritten

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSGBPB read block using PTR#
50 REM
60 osgbpb = &FFD1
70 DIM code% &100
80 FOR pass% = 0 TO 2 STEP 2
90 P% = code%
100 [ OPT pass%
110 .conb   EQUB 0       \ handle
120        EQUD &70     \ point to data
130        EQUD 7       \ size
140        EQUD 0       \ ptr
150 .conad  EQUW conb    \ address of con block
160 .start  LDA &80      \ get handle
170       STA conb     \ put in con block
180       LDA #4       \ function
190       LDX conad    \ point X and Y
200       LDY conad+1   \ at con block
210       JSR osgbpb   \ read block of data
220       RTS  \ bye bye
230 ]
240 NEXT pass%
250 REM open a file, "test" and write
260 REM "I am thirsty" to it. close.
270 REM set PTR to 5.
280 REM re-open and call OSGBPB
290 REM to read 7 bytes starting
300 REM at PTR into &70 to &76.
310 A$  = "I am thirsty"
320 ?&80 = OPENOUT("test")
330 FOR I% = 1 TO LEN(A$)
340    BPUT# ?&80,ASC(MID$(A$,I%,1))
350 NEXT
360 CLOSE# ?&80
370 ?&80 = OPENUP("test")
380 PTR# ?&80 = 5
390 CALL start
400 CLOSE# ?&80
410 FOR I% = &70 TO &76
420    PRINT CHR$(?I%)
430 NEXT
440 END
```

```
>RUN
t
h
i
r
s
t
y
```

OSGBPB Read Disk Title + Boot Up Option

on entry:

A = 5
X = LSB of address of control block
Y = MSB of address of control block
control block &00 = does not matter
control block &01 to &04 = memory address to return info
control block &05 to &08 = does not matter control block &09 to &0C = does not matter

(control block entries are least significant byte first)

action:

OSGBPB uses the memory address specified in the control block, in which to place:

length of title (1 byte) .
the title (n bytes, where n = length of title)
boot up option = *OPT 4 parameter (1 byte)

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
memory area = required information

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSGBPB read title + boot-up
50 REM
60 osgbpb = &FFD1
70 DIM code% &100
80 FOR pass% = 0 TO 2 STEP 2
90 P% = code%
100 [ OPT pass%
110 .data   EQUS "     " \ save area for title
120         EQUS "     " \ + boot-up option
130 .conb   EQUB 0       \ handle
140         EQUD data    \ point to data
150         EQUD 0       \ size
160         EQUD 0       \ ptr
170 .conad  EQUW conb    \ address of con block
180 .start  LDA #5       \ function
190         LDX conad    \ point X and Y
200         LDY conad+1   \ at con block
210         JSR osgbpb   \ get title
220         RTS     \ bye bye
230 ]
240 NEXT pass%
250 REM call OSGBPB to read title etc.
260 REM Display results.
270 CALL start
280 T$ = ""
290 FOR I% = 1 TO ?data
300   T$ = T$ + CHR$(data?I%)
310 NEXT
320 B% = data?((?data)+1)
330 PRINT "title = ";T$
340 PRINT "*OPT4,";B%
350 END

>RUN
title = DFS guide
*OPT4,0
```

OSGBPB Read *DIR Drive + Directory

on entry:

A = 6
X = LSB of address of control block
Y = MSB of address of control block
control block &00 = does not matter
control block &01 to &04 = memory address to return info
control block &05 to &08 = does not matter
control block &09 to &0C = does not matter

(control block entries are least significant byte first)

action:

OSGBPB uses the memory address specified in the control block, in which to place details of the default general purpose (*DIR) drive and directory as follows:

length of drive number = 1 (1 byte)
drive number in ASCII (1 byte)
length of directory = 1 (1 byte)
directory character in ASCII (1 byte)

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
memory area = required information

```
10 REM
20 REM       DEMONSTRATION
30 REM
40 REM OSGBPB read *DIR drive + dir
50 REM
60 osgbpb = &FFD1
70 DIM code% &100
80 FOR pass% = 0 TO 2 STEP 2
90 P% = code%
100 [ OPT pass%
110 .data   EQUS "     " \ save area for drive + dir
120 .conb   EQUB 0      \ handle
130         EQUD data    \ point to data
140         EQUD 0       \ size
150         EQUD 0       \ ptr
160 .conad  EQUW conb    \ address of con block
170 .start  LDA #6      \ function
180         LDX conad    \ point X and Y
190         LDY conad+1   \ at con block
200         JSR osgbpb    \ get drive + dir
210         RTS       \ bye bye
220 ]
230 NEXT pass%
240 REM set drive = 3, dir = S
250 REM call OSGBPB to read the default general purpose
260 REM drive and directory. Display results.
270 *DIR :3.S
280 CALL start
290 PRINT " drive = ";CHR$(data?1)
300 PRINT "directory = ";CHR$(data?3)
310 END

>RUN
drive = 3 directory = S
```

OSGBPB Read *LIB Drive + Directory

on entry:

A = 7
X = LSB of address of control block
Y = MSB of address of control block
control block &00 - does not matter
control block &01 to &04 = memory address to return info
control block &05 to &08 = does not matter
control block &09 to &0C = does not matter

(control block entries are least significant byte first)

action:

OSGBPB uses the memory address specified in the control block, in which to place details of the default machine-code library (*L1B) drive and directory as follows:

length of drive number =1 (1 byte)
drive number in ASCII (1 byte)
length of directory = 1 (1 byte)
directory character in ASCII (1 byte)

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
memory area = required information

```
10 REM
20 REM       DEMONSTRATION
30 REM
40 REM OSGBPB read *LIB drive + dir
50 REM
60 osgbpb = &FFD1
70 DIM code% &100
80 FOR pass% = 0 TO 2 STEP 2
90 P% = code%
100 [ OPT pass%
110 .data   EQUS "     " \ save are for drive + dir
120 .conb   EQUB 0      \ handLe
130         EQUD data    \ point to data
140         EQUD 0       \ size
150         EQUD 0       \ ptr
160 .conad  EQUW conb    \ address of con block
170 .start  LDA #7       \ function
180         LDX conad    \ point X and Y
190         LDY conad+1   \ at con block
200         JSR osgbpb    \ get drive + dir
210         RTS        \ bye bye
220 ]
230 NEXT pass%
240 REM set drive = 3, dir = S
250 REM call OSGBPB to read the default machine-code
260 REM library drive and directory. Display results.
270 *LIB :3.S
280 CALL start
290 PRINT " drive = ";CHR$(data?1)
300 PRINT "directory = ";CHR$(data?3)
310 END

>RUN
 drive = 3
directory = S
```

OSGBPB Read Filenames

on entry:

A = 8
X = LSB of address of control block
Y = MSB of address of control block
control block &00 = does not matter
control block &01 to &04 = memory address to return info
control block &05 to &08 = number of filenames to read
control block &09 to &0C = sequential pointer = 0 first call

(control block entries are least significant byte first)

action:

OSGBPB uses the memory address specified in the control block, in which to place details of the specified number of filenames as follows:

length of filename 1 (1 byte)
filename 1
length of filename 2 (1 byte)
filename 2
.... etc.

OSGBPB sets the C flag if the last filename in the catalogue has been returned. It also alters the control block. In byte zero, it places the number of times that the disk has been written. It adjusts number of filenames to read to the value of the number of filenames still to be read (i.e. not yet done), but only if it encountered some error condition. It also adjusts the memory address ready to receive more filenames. Finally it stores the number of bytes transferred in control block + 9. Thus OSGBPB can be re-entered with A = 8, without the need to re-adjust the contents of the control block.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
memory area = required information control
block set up for repeat call

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSGBPB read filenames
50 REM
60 osgbpb = &FFD1
70 DIM names 31*9
80 DIM code% &100
90 FOR pass% = 0 TO 2 STEP 2
100 P% = code%
110 [ OPT pass%
120 .conb   EQUB 0       \ handle
130         EQUD names    \ point to data
140         EQUD 31      \ number
150         EQUD 0       \ ptr
160 .conad  EQUW conb    \ address of con block
170 .start  LDA #8      \ function
180         LDX conad    \ point X and Y
190         LDY conad+1   \ at con block
200         JSR osgbpb   \ read filenames
210         RTS     \ bye bye
220 ]
230 NEXT pass%
240 REM call OSGBPB to read filenames.
250 CALL start
260 REM Display results.
270 K% = 0
280 REPEAT
290    FOR J% = 1 TO names?K%
300       PRINT CHR$(names?(K%+J%));
310    NEXT
320 PRINT
330 K% = K% + names?K% + 1
340 UNTIL K% = conb?9
350 END

>RUN
BOOK0        B00K5
B00K1        B00K6
B00K2        B00K7
B00K3        B00K8
B00K4        B00K9
```

10 Acorn DFS OSFIND

The DFS provides disk-specific OSFIND support which may be called from machine-code programs. This software also supports the BASIC commands, OPENIN, OPENOUT, OPENUP and CLOSE#.

OSFIND is entered at &FFCE within the OS ROM where it is vectored by the FINDV vector (&21C,21D) to &FF2A, which switches to the DFS ROM and then vectors via the extended FINDV vector (&DC9,&DCA) to &8E93 in the DFS ROM.

OSFIND is used to open or close a specified file. A file must be opened first by OSFIND, before it can be used by OSARGS, OSBGET, OSBPUT or OSGBPB. To open a file you put the filename, terminated by &0D, in a string and point the X and Y registers at this string.

Machine-code demonstration programs are supplied.

OSFIND Open a File

on entry:

A = &40 (open for input only) or
A = &80 (open for output only) or
A = &C0 (open for input/output)
X = LSB of address of filename string
Y = MSB of address of filename string

action:

OSFIND checks that less than five files are already open. If the file is output only, OSFIND attempts to delete that filename from the disk. OSFIND determines the correct file handle (see OSARGS), which it returns in A. If OSFIND could not open the file, it returns 0 in A.

on exit:

A = file handle (or zero = error)
X = unchanged
Y = unchanged
P = destroyed

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSFIND open one file
50 REM  -
60 osfind = &FFCE
70 DIM code% &100
80 FOR pass% = 0 TO 2 STEP 2
90 P% = code%
100 [ OPT pass%
110 .fname  EQUS "test"  \ filename
120         EQUB &D      \ terminate
130 .fnadd  EQUW fname   \ filename address
140 .start  LDA #&80     \ specify output
150         LDX fnadd    \ point X and Y at
160         LDY fnadd+1  \ filename
170         JSR osfind   \ do it. See Note 1
180         BNE ok       \ check ok
190         BRK           \ no. Note 2
195         EQUB 6 : EQUS "Not opened":EQUB 0
200 .ok     STA &70      \ save handle
210         RTS       \ bye bye
220 ]
230 NEXT pass%
240 REM Call OSFIND to open a file for
250 REM output. Close it using the
260 REM handle stored in &70. See if it's there.
270 CALL start
280 CLOSE# ?&70
290 *INFO test
300 END

>RUN
$.test    000000 000000 000000 006
```

Editor's note 1: It may be a wise precaution to ensure the Z flag correctly reflects the contents of A immediately after the OSFind call. Instructions such as ORA #0, AND #&FF, TAX and TAY as appropriate will all achieve this.

Note 2: Errors are normally generated by the use of BRK, an error number, the error message itself and then a zero terminating byte. Hence the addition of line 195.

OSFIND Close a File

on entry:

A = 0
X = does not matter
Y = file handle

action:

OSFIND checks that the file is opened. If so it closes the file and completes its catalogue entries.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSFIND close one file
50 REM
60 osfind = &FFCE
70 DIM code% &100
80 FOR pass% = 0 TO 2 STEP 2
90 P% = code%
100 [ OPT pass%
140 .start  LDA #0      \ specify close
160         LDY &70      \ handle
170         JSR osfind    \ do it
210         RTS     \ bye bye
220 ]
230 NEXT pass%
240 REM Open a file. Put handle in &70.
250 REM Call OSFIND to close it.
260 ?&70 = OPENOUT("test")
270 CALL start
280 *INFO test
290 END

>RUN
S.test    000000 000000 000000 006
```

OSFIND Close All Files

on entry:

A = 0
X = does not matter
Y = 0

action:

OSFIND closes all opened files and completes their catalogue entries.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSFIND close all files
50 REM
60 osfind = &FFCE
70 DIM code% &100
80 FOR pass% = 0 TO 2 STEP 2
90 P% = code%
100 [ OPT pass%
110 .start  LDA #0      \ specify close
120         LDY #0      \ handle
130         JSR osfind   \ do it
140         RTS       \ bye bye
150 ]
160 NEXT pass%
170 REM open 3 files.
180 REM call OSFIND to close them.
190 ?&70 = OPENOUT("test1")
200 ?&71 = OPENOUT("test2")
210 ?&72 = OPENOUT("test3")
220 CALL start
230 *INFO test
240 REM Note that the DFS allocates a default
250 REM file size of 64 (&40) sectors
260 REM to each of the three files.
270 END

>RUN
$.test3 000000 000000 000000 086
$.test2 000000 000000 000000 046
$.test1 000000 000000 000000 006
```

11 Acorn DFS OSFSC

The DFS provides disk-specific OSFSC support which may be called from machine-code programs. This software also supports the BASIC command, EOF#, as well as the star commands *CAT, *OPT and *RUN.

OSFSC is vectored by the FSCV vector (&21E,21F) to &FF2A, which switches to the DFS ROM and then vectors via the extended FSCV vector (&DCQ&DCD) to &95AA in the DFS ROM.

OSFSC is primarily written for internal use by the DFS/OS. However, you are free to use these functions yourself. You put the appropriate code in the A register to specify which function you want.

OSFSC does not have a direct entry address only an indirect entry at &21E. Note the correct way to code for this in the machine-code demonstration programs which accompany the description of each call.

OSFSC *OPT Handler

on entry:

A = 0
X = *OPT parameter 1
Y = *OPT parameter 2

action:

OSFSC does *OPT X,Y

on exit:

A = destroyed
X = destroyed
Y = destroyed
P = destroyed

```
10 REM
20 REM       DEMONSTRATION
30 REM
40 REM OSFSC *0PT handler
50 REM
70 DIM code% &1Q0
80 FOR pass% = 0 TO 2 STEP 2
90 P% = code%
100 [ OPT pass%
140 .start  LDA #0       \ specify *OPT4,3
150         LDX #4       \
160         LDY #3       \
170         JSR osfsc    \ call it
210          RTS       \ bye bye
211 .osfsc  JMP (&21E)    \ OSFSC
220 ]
230 NEXT pass%
235 REM Call osfsc to do *OPT4,3.
240 REM Then *CAT to confirm it.
250 CALL start
290 *CAT
310 END

>RUN
Drive 1       Option 3 (EXEC)
Directory :1.$ Library :0.$

testing
```

OSFSC Check EOF

on entry:

A = 1
X = file handle of file to be checked
Y = does not matter

action:

OSFSC checks to see if the end of the specified file has been reached, placing the result as a code in X.

on exit:

A = destroyed
X = 0 (not end of file) or
X = &FF (end of file)
Y = destroyed
P = destroyed

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSFSC EOF handler
50 REM
60 DIM code% &100
70 FOR pass% = 0 TO 2 STEP 2
80 P% = code%
90 [ OPT pass%
100 .start  LDA #1       \ function
110         LDX &70      \ handle
120         JSR osfsc    \ call it
130         STX &71      \ save EOF
140         RTS        \ bye bye
150 .osfsc  JMP (&21E)    \ OSFSC
160 ]
170 NEXT pass%
180 REM Open a file/'test". Write
190 REM some bytes to it. Close it.
200 REM Re-open for input. Read a
210 REM byte at a time- Call OSFSC
220 REM to check for end of file.
230 REM Print number of bytes and
240 REM EXT# to confirm.
250 ?&70 = OPENOUT("test")
260 FOR I% = 1 TO 9
270    BPUT# ?&70,1
280 NEXT
290 CLOSE# ?&70
300 ?&70 = OPENUP("test")
310 num% = 0
320 REPEAT
330   CALL start
340    IF ?&71 = &FF GOTO 370
350    A% = BGET# ?&70
360    num% = num% + 1
370 UNTIL ?&71 = &FF
380 PRINT "bytes read = ";num%
390 PRINT "     EXT# = ";EXT# ?&70
400 CLOSE# ?&70
410 END

>RUN
bytes read = 9
      EXT# = 9
```

OSFSC */ Handler

on entry:

A = 2
X = LSB of address of rest of command string after */
Y = MSB of address of rest of command string after */

action:

OSFSC *RUNs the file whose name comes after the '*/' Note that this function is the entry point used internally by the DFS to handle */, the abbreviated form of *RUN. (see *RUN).

on exit:

A = destroyed
X = destroyed
Y = destroyed
P = destroyed

```
10 REM
20 REM     DEMONSTRATION
30 REM
40 REM OSFSC /* handler
50 REM
60 DIM code% &100
70 FOR pass% = 0 TO 2 STEP 2
80 P% = code%
90 [ OPT pass%
100 .comrest EQUS "test" \ rest of command
110          EQUB &D      \ terminate
120 .com     EQUW comrest  \ address
130 .start   LDA #2       \ function
140          LDX com      \ point X and Y
150          LDY com+1    \ at rest of command
160          JSR osfsc    \ call it
170          RTS       \ bye bye
180 .osfsc  JMP (&21E)    \ OSFSC
190 ]
200 NEXT pass%
210 REM Set up a small machine-code
220 REM program/'test", to print
230 REM the letter "A". Kid OSFSC
240 REM into believing it has got
250 REM a "*/test" to handle.
260 I% = &70
270 REPEAT
280 READ ?I%
290 I% = I% + 1
300 UNTIL ?(I%-1) = &60
310 *SAVE test 70 +6 70
320 CALL start
330 END
340 REM small machine-code program
350 DATA &A9,65,&20,&EE,&FF,&60

>RUN
A
```

OSFSC Unrecognised Star Command Handler

on entry:

A = 3
X = LSB of address of rest of command string after *
Y = MSB of address of rest of command string after *

action:

OSFSC searches the catalogue for a file whose name occurred in the star command and attempts to *RUN that file.

on exit:

A = destroyed
X = destroyed
Y = destroyed
P = destroyed

```
10 REM
20 REM       DEMONSTRATION
30 REM
40 REM OSFSC unrecognised star command
50 REM
60 DIM code% &100
70 FOR pass% = 0 TO 2 STEP 2
80 P% = code%
90 [ OPT pass%
100 .comrest EQUS "test" \ rest of command
110          EQUB &D      \ terminate
120 .com     EQUW comrest  \ address
130 .start   LDA #3       \ function
140          LDX com      \ point X and Y
150          LDY com+1    \ at rest of command
160          JSR osfsc    \ call it
170          RTS    \ bye bye
180 .osfsc  JMP (&21E)    \ OSFSC
190 ]
200 NEXT pass%
210 REM Set up a small machine-code
220 REM program, "test", to print
230 REM the letter "A". Kid OSFSC
240 REM into believing it has got
250 REM a "*test" to handle.
260 I% = &70
270 REPEAT
280 READ ?I%
290 I% = I% + 1
300 UNTIL ?(I%-1) = &60
310 *SAVE test 70 +6 70
320 CALL start
330 END
340 REM small machine-code program
350 DATA &A9,65,&20,&EE,&FF/&60

>RUN
A
```

OSFSC *RUN Handler

on entry:

A = 4
X = LSB of address of rest of command string after *RUN
Y = MSB of address of rest of command string after *RUN

action:

OSFSC uses X and Y to locate a filename string, terminated by &0D.
It attempts to *RUN this disk file.

on exit:

A = destroyed
X = destroyed
Y = destroyed
P = destroyed

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSFSC *RUN handler
50 REM
60 DIM code% &100
70 FOR pass% = 0 TO 2 STEP 2
80 P% = code%
90 [ OPT pass%
100 .comrest EQUS "test" \ rest of command
110          EQUB &D      \ terminate
120 .com     EQUW comrest  \ address
130 .start   LDA #4       \ function
140          LDX com      \ point X and Y
150          LDY com+1     \ at rest of command
160          JSR osfsc    \ call it
170          RTS       \ bye bye
180 .osfsc  JMP (&21E)    \ OSFSC
190 ]
200 NEXT pass%
210 REM Set up a small machine-code
220 REM program,"test", to print
230 REM the letter "A". Kid OSFSC
240 REM into believing it has got
250 REM a "*RUN test" to handle.
260 I% = &70
270 REPEAT
280 READ ?I%
290 I% = I% + 1
300 UNTIL ?(I%-1) = &60
310 *SAVE test 70 +6 70
320 CALL start
330 END
340 REM small machine-code program
350 DATA &A9,65,&20,&EE,&FF,&60

>RUN
A
```

OSFSC *CAT Handler

on entry:

A = 5
X = LSB of address of rest of command string after *CAT
Y = MSB of address of rest of command string after *CAT

action:

OSFSC uses X and Y to locate the parameters of an *CAT command and then does the *CAT.

on exit:

A = destroyed
X = destroyed
Y = destroyed
P = destroyed

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSFSC *CAT handler
50 REM
60 DIM code% &100
70 FOR pass% = 0 TO 2 STEP 2
80 P% = code%
90 [ OPT pass%
100 .comrest EQUS "0"     \ drive 0
110          EQUB &D      \ terminate
120 .com     EQUW comrest  \ address
130 .start   LDA #5       \ function
140          LDX com      \ point X and Y at
150          LDY com+1    \ rest of command
160          JSR osfsc    \ call it
170          RTS       \ bye bye
180 .osfsc  JMP (&21E)    \ OSFSC
190 ]
200 NEXT pass%
210 REM call osfsc to do *CAT 0
220 CALL start
230 END

>RUN Drive 0   Option 3
Directory      (EXEC)
:0.$           Library :0.$

    Testing
```

OSFSC Shutdown Files

on entry:

A = 6
X = does not matter
Y = does not matter

action:

OSFSC invokes OSBYTE &77 to shutdown any of the five files currently open. The DFS enters here if another filing system demands control.

on exit:

A = destroyed
X = destroyed
Y = destroyed
P = destroyed

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSFSC shutdown files
50 REM
60 DIM code% &100
70 FOR pass% = 0 TO 2 STEP 2
80 P% = code%
90 [OPT pass%
100 .start  LDA #6       \ function
110         JSR osfsc    \ call it
120         RTS     \ bye bye
130 .osfsc  JMP (&21E)    \ OSFSC
140 ]
150 NEXT pass%
160 REM I cannot think of a way of
170 REM demonstrating this call, but
180 REM this is how you code it.
220 CALL start
230 END
```

OSFSC Give File Handle Ranges

on entry:

A = 7
X = does not matter
Y = does not matter

action:

OSFSC puts the lowest possible DFS file handle (&11) in X and the highest possible file handle (&15) in Y.

on exit:

A = destroyed
X = lowest DFS file handle
Y = highest DFS file handle
P = destroyed

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSFSC handles
50 REM
60 DIM code% &100
70 FOR pass% = 0 TO 2 STEP 2
80 P% = code%
90 [ OPT pass%
100 .start  LDA #7  \  function
110         JSR osfsc    \ call it
120         STX &70      \ save low range
130         STY &71      \ save high range
140         RTS     \ bye bye
150 .osfsc  JMP (&21E)    \ OSFSC
160 ]
170 NEXT pass%
180 CALL start
190 PRINT " lowest DFS handle = &";"?&70
200 PRINT "highest DFS handle = &";"?&71
210 END

>RUN
 lowest DFS handle = &11
highest DFS handle = &15
```

OSFSC *ENABLE Checker

on entry:

A = 8
X = does not matter
Y = does not matter

action:

The DFS Enable Flag is at &10C8. It can have one of three values:

&FF = no record of *ENABLE being received
&00= a transitory state. This flag is on its way from &01 to &FF.
&01= *ENABLE was the last star command.

This OSFSC call sets the N flag (minus flag) of the status register if *ENABLE was not the last received star command. It also resets the Enable Flag as follows:

Before Call	After Call	N flag
&FF	&FF	1
&00	&FF	1
&01	&00	0

Thus OSFSC both switches the Enable Flag and also tests for Enable (use BPL to check for enabled).

on exit:

A = destroyed
X = destroyed
Y = destroyed
P = destroyed

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSFSC enable checker
50 REM
60 DIM code% &100
70 FOR pass% = 0 TO 2 STEP 2
80 P% = code%
90 [ OPT pass%
100 .start  LDA #8       \ function
110         JSR osfsc    \ call it
120         BPL enab     \ check enabled
130         LDA #0      \ set disabled
140         STA &70      \
150         RTS     \
160 .enab   LDA #1       \ set enabled
170         STA &70      \
180         RTS     \ bye bye
190 .osfsc  JMP (&21E)    \ OSFSC
200 ]
210 NEXT pass%
220 DIM A$(2)
230 A$(0) = "disabled"
240 A$(1) = "enabled"
250 *ENABLE
260 PRINT "enable flag = &";~?&10C8
270 CALL start
280 PRINT "enable flag = &";~?&10C8;
290 PRINT " status = ";A$(?&70)
300 CALL start
310 PRINT "enable flag = &";~?&10C8;
320 PRINT " status = ";A$(?&70)
330 CALL start
340 PRINT "enable flag = &";~?&10C8;
350 PRINT " status = ";A$(?&70)
360 END

>RUN
enable flag = &1
enable flag = &0 status = enabled
enable flag = &FF status = disabled
enable flag = &FF status = disabled
```

12 Acorn DFS OSWORD

OSWORD is entered at &FFF1 and is vectored by the WORDV vector (&20C,20D) into an address within the OS ROM. OSWORD is a multi-functional piece of software. You choose which function you want by loading the appropriate code into the A register. The OS ROM then routes the OSWORD call as follows:

A = &00 to &0D stays in the OS ROM.

A = &0E to &DF saves A,X and Y in &EF, &F0 and &F1 respectively and issues a paged ROM service call type &08 (unrecognised OSWORD) to each paged ROM with a service entry, until one of them handles the call.

A = &E0 to &FF vectors via USERV (&200,&201). This allows you to write your own OSWORD routines.

Thus the DFS is given OSWORD calls in the range &0E to &DF. Of these it recognises just three, &7D to &7F, and ignores the rest. OSWORD &7D and &7E are specific functions, but OSWORD &7F (which is the bulk of the code) contains many sub-functions. In summary, the DFS OSWORD support consists of:

OSWORD &7D read number of times disk has been written.

OSWORD &7E read number of sectors on the disk.

OSWORD &7F execute an 8271 FDC command through all phases.

12.1 OSWORD &7D

OSWORD &7D reads the catalogue for the current default general purpose drive. From this it extracts the number of times that the disk has been written, often called the disk cycles. On entry, you point the X and Y registers at a one-byte long memory address to receive the result.

OSWORD Read Times Disk Written

on entry:

A = &7D
X = LSB of address to store result
Y = MSB of address to store result

action:

OSWORD reads the catalogue for the current default general purpose drive, as specified by the most recent *DRIVE command. It places the result in a one-byte memory area specified in X and Y on entry.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
result (1 byte) = times disk written

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSWORD read disk cycles
50 REM
60 osword = &FFF1
70 DIM code% &100
80 FOR pass% = 0 TO 2 STEP 2
90 P% = code%
100 [OPT pass%
110 .start  LDA #67D      \ function
120         LDX #&70      \ address for cycles
130         LDY #&00      \
140         JSR osword     \ OSWORD
150         RTS
160 ]
170 NEXT pass%
180 CALL start
190 PRINT "cycles = ";?&70
200 END

>RUN
cycles =150
```

12.2 OSWORD &7E

OSWORD &7E reads the catalogue for the current default general purpose drive. From this it extracts the total number of sectors available on the disk. This will be 800 (&320) for 80 track diskettes and 400 (&190) for 40 track diskettes. On entry, you point the X and Y registers at a four-byte long memory address to receive the result.

OSWORD Read Number of Sectors

on entry:

A = &7E
X = LSB of address to store result
Y = MSB of address to store result

action:

OSWORD reads the catalogue for the current default general purpose drive, as specified by the most recent *DRIVE command. It places the result in a four-byte memory area specified in X and Y on entry. The result is stored in a strange way:

result byte 0 = 0
result byte 1 = LSB of number of sectors
result byte 2 = MSB of number of sectors
result byte 3 = 0

result byte 3 allows for expansion, in the form of Winchesters for example. If you like, you can regard the result as an ordinary four-byte integer, in which case the result is 256 * number of sectors.

A word of caution, There is no magic hardware that writes the number of sectors on the diskette into the catalogue in the first place. It is written there by software, normally by the formatter program. You can rely on this field for all your own diskettes, but be a bit wary of commercial software. Much of this is written in a combined 40 track / 80 track format about which we will say much more later. It stands to reason that, for such diskettes, the value written in the number of sectors field may not be what you would expect.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
result (4 bytes) = 256*number of sectors

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSWORD read number of sectors
50 REM
60 osword = &FFF1
70 DIM code% &100
80 FOR pass% = 0 TO 2  STEP 2
90 P% = code%
100 [ OPT pass%
110 .start  LDA #&7E       \ function
120         LDX #&70       \ put number in &70-&73
130         LDY #&00       \
140         JSR osword     \ OSWORD
150         RTS
160 ]
170 NEXT pass%
180 CALL start
190 T% = (?&71+256*?&72) DIV 10
200 PRINT "number of tracks = ";T%
210 END

>RUN
number of tracks = 80
```

12.3 OSWORD &7F

OSWORD &7F executes a single 8271 FDC command, as explained in Chapter 3, through all four phases from start to finish. This relieves you of the burden of writing Status Register checking code, NMI interrupt handlers etc.

You communicate with OSWORD via two different memory areas. Firstly you build a parameter block which defines to OSWORD exactly what you require. On entry you point the X and Y registers at this parameter block. Secondly you specify a buffer area into which any data to be read will be placed, or from which any data to be written will be taken. You specify this buffer area address within the parameter block. Not all 8271 commands use the buffer area, but OSWORD &7F is designed around a standard interface, so you must leave room for it regardless of whether or not it is to be used.

The standard format of the parameter block for all OSWORD &7F calls is as follows:

param block &00 = drive number
param block &01 to &04 = address of buffer (LSB first)
param block &05 = number of parameters for this 8271 command
param block &06 = 8271 command code
param block &07 onwards = the parameters required by the 8271 command

You will see that the parameter block has a variable length, depending on the number of parameters that the 8271 command needs. OSWORD will always return the contents of the Result Register in the next available byte of the parameter block. These are as follows:

param block &07 for 0 parameters
param block &08 for 1 parameter
param block &09 for 2 parameters
param block &0A for 3 parameters
param block &0B for 4 parameters
param block &0C for 5 parameters

Thus we see that the parameter block is from seven to twelve bytes long.

OSWORD &7F also standardises the drive number code for you. In chapter 3 we explained that the 8271 was told which drive to use via the command code. The command codes were tabulated in Figure 19 for drive 0. To issue a command for a different drive, we add a value to the command code:

drive 1 add &40 to command code
drive 2 add &80 to command code
drive 3 add &C0 to command code

You will doubtless be relieved to hear that OSWORD handles all this for you. You always specify the command code for drive 1 (add &40) and OSWORD uses the drive number that you insert into the first byte of the parameter block to calculate the correct command code. In fact, OSWORD additionally reads the drive status register to ensure that the drive is ready, unless you specify a negative drive number (&7F to &FF).

If you specify a negative drive number, the DFS skips the drive status check and processes the command for the currently selected drive. Figure 29 tabulates the command codes that can be used with OSWORD.

hex comm and	no. parms	command
4A	2	write data 128 bytes
4B	3	write data multi-sector
4E	2	write deleted data 128 bytes
4F	3	write deleted data multi-sector
52	2	read data 128 bytes
53	3	read data multi-sector
56	2	read data and deleted data 128 bytes
57	3	read data and deleted data multi-sector
5B	3	read sector id(s)
5E	2	verify data and deleted data 128 bytes
5F	3	verify data and deleted data multi-sector
63	5	format track
69	1	seek
6C	0	read drive status
75	4	initialise 8271
75	4	load bad tracks
7A	2	write special register
7D	1	read special register

Figure 29. OSWORD &7F Command Set.

Many of the commands tabulated above incorporate an automatic 'SEEK' of the required physical track. Many of the commands then check the track by comparing the physical track number with the logical track number contained in the sector ids. If these do not match, they automatically step to the next two tracks in turn, each time trying to match the track number. Only if this fails do they return an error code.

We shall now discuss each one of these commands in turn (command code order). Machine-code demonstration programs accompany the more useful OSWORD calls.

OSWORD Write Data 128 bytes

on entry:

A = &7F
X = LSI3 of address of pa ram block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = address of buffer (LSB first)
param block &5 = &02
param block &6 = &4A
param block &7 = logical track number
param block &8 = logical sector number

action:

The 8271 uses the appropriate Track Register as a base from which to 'seek' the specified track and it then searches for a sector id which matches the track and sector number specified in the parameter block. If it cannot find the sector it returns a value of &18 to param block &9. Otherwise, it writes a data mark at the start of the data field and copies 128 bytes of data from the buffer (address specified in parameter block) to the first 128 bytes of the specified sector. It re-calculates the data CRC and writes this to disk.

The 8271 is designed around a sector size of 128 bytes (not 256 bytes as used by the BBC Microcomputer). In the multi-sector commands it offers the facility to vary the sector size. Thus it is most unlikely that you will want to use the 128 byte calls, as the multi-sector calls are much better suited to the format of the BBC Microcomputer.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &9 = Result Register (see Chapter 3)

bit 7,6 = 0
bit 5 = deleted data flag
bit 4,3 = completion type
bit 2,1 = completion code
bit 0 = 0

NO DEMONSTRATION PROGRAM

OSWORD Write Data Multi-sector

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = address of buffer (LSB first)
param block &5 = &03
param block &6 = &4B
param block &7 = logical track number
param block &8 = start logical sector number
param block &9 = sector size / number of sectors

The most significant three bits represent the sector size:

0	=	128 bytes per sector
1	=	256 bytes per sector
10	=	512 bytes per sector
11	=	1024 bytes per sector
100	=	2048 bytes per sector
101	=	4096 bytes per sector
110	=	8192 bytes per sector
111	=	16384 bytes per sector

The least significant five bits represent the number of sectors per track and can have any value between zero and &1F.

In practice, the Acorn DFS works with 256 bytes per sector. The allowable values for this whole field are thus:

&21 =	1 sector of 256 bytes
&22 =	2 sectors of 256 bytes
&23 =	3 sectors of 256 bytes
&24 =	4 sectors of 256 bytes
&25 =	5 sectors of 256 bytes
&26 =	6 sectors of 256 bytes
&27 =	7 sectors of 256 bytes
&28 =	8 sectors of 256 bytes
&29 =	9 sectors of 256 bytes
&2A=	10 sectors of 256 bytes

action:

The 8271 uses the appropriate Track Register as a base from which to 'seek' the specified track and it then searches for a sector id which matches the track and sector number specified in the parameter block. If it cannot find the sector it returns a value Of &18 to param block &A. Otherwise, it writes a data mark at the start of the data field and copies the first sector of data from the buffer (address specified in parameter block) to the specified disk sector. It re-calculates the data CRC and writes this to disk.

If you have specified more than one sector, the 8271 adds 1 to the sector number and searches for this sector. If it finds it, it writes this sector from the current position in the buffer.

This procedure is repeated until either an error occurs, or all the sectors have been successfully written.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &A = Result Register (see Chapter 3)

bit 7,6 = 0
bit 5 = deleted data flag
bit 4,3 = completion type
bit 2,1 = completion code
bit 0 = 0

```
10 REM
20 REM       DEMONSTRATION
30 REM
40 REM OSWORD write data multi-sector
50 REM
60 DIM wdata 255, rdata 255, mc% 255
70 $wdata = STRING$(255,"A") : $rdata = STRING$(255," ")
80 osword = &FFF1
90 FOR pass% = 0 TO 2 STEP 2
100 P% = mc%
110 [OPT  pass%
120 \    write parameter block
130 .wparm EQUB 3        \ drive 3
140        EQUD wdata    \ write buffer
150        EQUB 3       \ no. of parms
160        EQUB &4B      \ command
170        EQUB 79       \ log track
180        EQUB 0       \ log sector
190        EQUB &21      \ 1 sector
200 .wres  EQUB 0       \ result
210 \    read parameter block
220 .rparm EQUB 3       \ drive 3
230        EQUD rdata    \ read buffer
240        EQUB 3        \ no. of parms
250        EQUB &53      \ command
260        EQUB 79       \ log track
270        EQUB 0       \ log sector
280        EQUB &21      \ 1 sector
290 .rres  EQUB 0       \ result
300 .raddr EQUW rparm    \ addr of rparm
310 .waddr EQUW wparm    \ addr of wparm
320
330 .start LDA #&7F     \ function
340        LDX waddr    \ point X & Y
350        LDY waddr+1   \ at wparm
360        JSR osword   \ write a sect
370        LDA wres     \ get result
380        AND #&1E     \ extract error bits
390        BNE end      \ if bad
400 .read  LDA #&7F     \ function
410        LDX raddr    \ point X & Y
420        LDY raddr+1   \ at rparm
430        JSR osword    \ read back
440 .end   RTS ]
```

```
450 NEXT pass%
460 REM This demonstration writes 255 bytes of Letter "A" from
470 REM wdata to drive 3, track 79, sector 0. It then reads this
480 REM data back again into rdata and checks it's the same.
490 CALL start
500 PRINT "write result = &";-?wres;" read result = &";~?rres
510 IF $rdata <> $wdata PRINT "bad"
520 END

>RUN
write result = &0 read result = &0
```

OSWORD Write Deleted Data 128 bytes

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = address of buffer (LSB first)
param block &5 = &02
param block &6 = &4E
param block &7 = logical track number
param block &8 = logical sector number

action:

The 8271 uses the appropriate Track Register as a base from which to 'seek' the specified track and it then searches for a sector id which matches the track and sector number specified in the parameter block. If it cannot find the sector it returns a value of &18 to param block &9. Otherwise, it writes a deleted data mark at the start of the data field and copies 128 bytes of data from the buffer (address specified in parameter block) to the first 128 bytes of the specified sector. It re-calculates the data CRC and writes this to disk.

The 8271 is designed around a sector size of 128 bytes (not 256 bytes as used by the BBC Microcomputer). In the multi-sector commands it offers the facility to vary the sector size. Thus it is most unlikely that you will want to use the 128 byte calls, as the multi-sector calls are much better suited to the format of the BBC Microcomputer.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &9 = Result Register (see Chapter 3)

> bit 7,6 = 0
> bit 5 = deleted data flag
> bit 4,3 = completion type
> bit 2,1 = completion code
> bit 0 = 0

NO DEMONSTRATION PROGRAM

OSWORD Write Deleted Data Multi-sector

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = address of buffer (LSB first)
param block &5 = &03
param block &6 = &4F
param block &7 = logical track number
param block &8 = start logical sector number
param block &9 = sector size / number of sectors

The most significant three bits represent the sector size:

000 =	128 bytes per sector
001 =	256 bytes per sector
010 =	512 bytes per sector
011 =	1024 bytes per sector
100 =	2048 bytes per sector
101 =	4096 bytes per sector

110 =	8192 bytes per sector
111 =	16384 bytes per sector

The least significant five bytes represent the number of sectors per track and can have any value between zero and &1F.

In practice, the Acorn DFS works with 256 bytes per sector. The allowable values for this whole field are thus:

&21 =	1 sector of 256 bytes
&22 =	2 sectors of 256 bytes
&23 =	3 sectors of 256 bytes
&24 =	4 sectors of 256 bytes
&25 =	5 sectors of 256 bytes
&26 =	6 sectors of 256 bytes
&27 =	7 sectors of 256 bytes
&28 =	8 sectors of 256 bytes
&29 =	9 sectors of 256 bytes
&2A =	10 sectors of 256 bytes

action:

The 8271 uses the appropriate Track Register as a base from which to 'seek' the specified track and it then searches for a sector id which matches the track and sector number specified in the parameter block. If it cannot find the sector it returns a value of &18 to param block &A. Otherwise, it writes a deleted data mark at the start of the data field and copies the first sector of data from the buffer (address specified in parameter block) to the specified disk sector. It re-calculates the data CRC and writes this to disk.

If you have specified more than one sector, the 8271 adds 1 to the sector number and searches for this sector. If it finds it, it writes this sector from the current position in the buffer.

This procedure is repeated until either an error occurs, or all the sectors have been successfully written.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &A = Result Register (see Chapter 3)

bit 7,6 = 0
bit 5 = deleted data flag
bit 4,3 = completion type bit
2,1 = completion code bit 0 =
0

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSWORD write deleted data multi-sector
50 REM
60 DIM wdata 255, rdata 255, mc% 255
70 $wdata = STRING$(255,"A") : $rdata = STRING$(255," ")
80 osword = &FFF1
90 FOR pass% = 0 TO 2 STEP 2
100 P% = mc%
110 [OPT  pass%
120 \    write parameter block
130 .wparm EQUB 3       \ drive 3
140        EQUD wdata    \ write buffer
150        EQUB 3       \ no- of parms
160        EQUB &4F     \ command
170        EQUB 79      \ log track
180        EQUB 0       \ log sector
190        EQUB &21      \ 1 sector
200 .wres  EQUB 0       \ result
210 \    read parameter block
220 .rparm EQUB 3       \ drive 3
230        EQUD rdata    \ read buffer
240        EQUB 3       \ no. of parms
250        EQUB &57      \ command
260        EQUB 79      \ log track
270        EQUB 0       \ log sector
280        EQUB &21      \ 1 sector
290 .rres  EQUB 0       \ result
300 .raddr EQUW rparm    \ addr of rparm
310 .waddr EQUW wparm    \ addr of wparm
320
330 .start LDA #&7F     \ function
340        LDX waddr     \ point X & Y
350        LDY waddr+1   \ at wparm
360        JSR osword    \ write a sect
370        LDA wres     \ get result
380        AND #&1E     \ extract error bits
390        BNE end      \ if bad
400 .read  LDA #&7F     \ function
410        LDX raddr     \ point X & Y
420        LDY raddr+1   \ at rparm
430        JSR osword    \ read back
440 .end   RTS ]
450 NEXT pass%
460 REM This demonstration writes 255 bytes of Letter "A" from
470 REM wdata to drive 3, track 79, sector 0. It then reads
 this
480 REM data back again into rdata and checks it's the same.
490 CALL start
5p0 PRINT "write result = &";~?wres
510 PRINT " read result = &";~?rres;" note delete bit"
520 IF $rdata <> $wdata PRINT "bad"
530 END

>RUN
write result = &0
read result = &20 note delete bit
```

OSWORD Read Data 128 bytes

on entry:

A = &7F

X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = address of buffer (LSB first)
param block &5 = &02
param block &6 = &52
param block &7 = logical track number
param block &8 = logical sector number

action:

The 8271 uses the appropriate Track Register as a base from which to 'seek' the specified track and it then searches for a sector id which matches the track and sector number specified in the parameter block. If it cannot find the sector it returns a value of &18 to param block &9. It then checks that the data field contains a data mark. If instead it starts with a deleted data mark, the 8271 skips to the result phase, but reports no error. If there was a data mark, it copies 128 bytes of data from the start of the sector to the buffer (address specified in parameter block), checking the CRC en route.

The 8271 is designed around a sector size of 128 bytes (not 256 bytes as used by the BBC Microcomputer). In the multi-sector commands it offers the facility to vary the sector size. Thus it is most unlikely that you will want to use the 128 byte calls, as the multi-sector calls are much better suited to the format of the BBC Microcomputer.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
buffer = data read
param block &9 = Result Register (see Chapter 3)

bit 7,6 = 0
bit 5 = deleted data flag
bit 4,3 = completion type
bit 2,1 = completion code
bit 0 = 0

NO DEMONSTRATION PROGRAM

OSWORD Read Data Multi-sector

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = address of buffer (LSB first)
param block &5 = &03
param block &6 = &53
param block &7 = logical track number
param block &8 = logical sector number
param block &9 = sector size / number of sectors

The most significant three bits represent the sector size:

000 =	128 bytes per sector
001 =	256 bytes per sector
010 =	512 bytes per sector
011 =	1024 bytes per sector
100 =	2048 bytes per sector
101 =	4096 bytes per sector
110 =	8192 bytes per sector
111 =	16384 bytes per sector

The least significant five bits represent the number of sectors per track and can have any value between zero and &lF.

In practice, the Acorn DFS works with 256 bytes per sector. The allowable values for this whole field are thus:

&21 =	1 sector of 256 bytes
&22 =	2 sectors of 256 bytes
&23 =	3 sectors of 256 bytes
&24 =	4 sectors of 256 bytes
&25 =	5 sectors of 256 bytes
&26 =	6 sectors of 256 bytes
&27 =	7 sectors of 256 bytes
&28 =	8 sectors of 256 bytes
&29 =	9 sectors of 256 bytes
&2A =	10 sectors of 256 bytes

action:

The 8271 uses the appropriate Track Register as a base from which to 'seek' the specified track and it then searches for a sector id which matches the track and sector number specified in the parameter block. If it cannot find the sector it returns a value of &18 to param block &A. It then checks that the data field contains a data mark. If instead it starts with a deleted data mark, the 8271 skips over that sector, but counts it as one of the sectors it had to read. If there was a data mark, it copies the sector of data to the buffer (address specified in parameter block), checking the CRC en route.

If you have specified more than one sector, the 8271 adds 1 to the sector number and searches for this sector. If it finds it, it repeats the steps above. This procedure is repeated until either an error occurs, or all the sectors have been read successfully.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
buffer = data read
param block &A = Result Register (see Chapter 3)

bit 7,6 = 0
bit 5 = deleted data flag
bit 4,3 = completion type
bit 2,1 = completion code
bit 0 = 0

```
10 REM
20 REM  DEMONSTRATION
30 REM
40 REM OSWORD read data muLti-sector
50 REM
60 DIM wdata 255, rdata 255, mc% 255
70 $wdata = STRING$(255,"A") : Srdata = STRING$(255," ")
80 osword = &FFF1
90 FOR pass% = 0 TO 2 STEP 2
100 P% = mc%
```

```
110 [OPT pass%
120 \   write parameter block
130 .wparm     EQUB 3  \ drive 3
140           EQUD wdata      \ write buffer
150           EQUB 3  \ no. of parms
160           EQUB &4B        \ command
170           EQUB 79 \ Log track
180           EQUB 0  \ Log sector
190           EQUB &21        \ 1 sector
200 .wres      EQUB 0  \ result
210 \   read parameter block
220 .rparm     EQUB 3  \ drive 3
230           EQUD rdata      \ read buffer
240           EQUB 3  \ no. of parms
250           EQUB &53        \ command
260           EQUB 79 \ log track
270           EQUB 0  \ log sector
280           EQUB &21        \ 1 sector
290 .rres      EQUB 0  \ result
300 *. raddr   EQUW rparm      \ addr of rparm
310 .waddr     EQUW wparm      \ addr of wparm
320
330 .start     LDA #&7F        \ function
340           LDX waddr       \ point X & Y
350           LDY waddr+1     \ at wparm
360           JSR osword      \ write a sect
370           LDA wres        \ get result
380           AND #&1E        \ save it
390           BNE end \ if bad
400 .read      LDA #&7F        \ function
410           LDX raddr       \ point X & Y
420           LDY raddr+1     \ at rparm
430           JSR osword      \ read back
440 .end       RTS ]
450 NEXT pass%
460 REM This demonstration writes 256 bytes of letter "A" from
470 REM wdata to drive 3, track 79, sector 0. It then reads this
480 REM data back again into rdata and checks it's the same.
490 CALL start
500 PRINT "write result = &";~?wres;" read result = &";~?rres
510 IF $rdata <> $wdata PRINT "bad"
520 END

>RUN
write result = &0 read result = &0
```

OSWORD Read Data and Deleted Data 128 bytes

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = address of buffer (LSB first)
param block &5 = &02
param block &6 = &56
param block &7 = logical track number
param block &8 = logical sector number

action:

The 8271 uses the appropriate Track Register as a base from which to 'seek' the specified track and it then searches for a sector id which matches the track and sector number specified in the parameter block. If it cannot find the sector it returns a value of &18 to param block &9. It copies 128 bytes of data from the start of the sector to the buffer (address specified in parameter block), checking the CRC en route. If the data field contained a deleted data mark, it sets the deleted data flag in param block &9.

The 8271 is designed around a sector size of 128 bytes (not 256 bytes as used by the BBC Microcomputer). In the multi-sector commands it offers the facility to vary the sector size. Thus it is most unlikely that you will want to use the 128 byte calls, as the multi-sector calls are much better suited to the format of the BBC Microcomputer.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
buffer = data read
pa ram block &9 = Result Register (see Chapter 3)

bit 7,6 = 0
bit 5 = deleted data flag
bit 4,3 = completion type
bit 2,1 = completion code
bit 0 = 0

NO DEMONSTRATION PROGRAM

OSWORD Read Data and Deleted Data Multi-sector

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = address of buffer (LSB first)
param block &5 = &03
param block &6 = &57
param block &7 = logical track number
param block &8 = logical sector number
param block &9 = sector size / number of sectors

The most significant three bits represent the sector size:

000 =	128 bytes per sector
001 =	256 bytes per sector
010 =	512 bytes per sector
011 =	1024 bytes per sector
100 =	2048 bytes per sector
101 =	4096 bytes per sector
110 =	8192 bytes per sector
111 =	16384 bytes per sector

The least significant five bits represent the number of sectors per track and can have any value between zero and &1F.

In practice, the Acorn DFS works with 256 bytes per sector. The allowable values for this whole field are thus:

&21 =	1 sector of 256 bytes
&22 =	2 sectors of 256 bytes
&23 =	3 sectors of 256 bytes
&24 =	4 sectors of 256 bytes
&25 =	5 sectors of 256 bytes
&26 =	6 sectors of 256 bytes
&27 =	7 sectors of 256 bytes
&28 =	8 sectors of 256 bytes
&29 =	9 sectors of 256 bytes
&2A	10 sectors of 256 bytes

action:

The 8271 uses the appropriate Track Register as a base from which to 'seek' the specified track and it then searches for a sector id which matches the track and sector number specified in the parameter block. If it cannot find the sector it returns a value of &18 to param block &A. It copies the sector of data to the buffer (address specified in parameter block), checking the CRC en route.

If you have specified more than one sector, the 8271 adds 1 to the sector number and searches for this sector. If it finds it, it repeats the steps above. This procedure is repeated until either an error occurs, or all the sectors have been successfully read.

If any of the sectors contains a deleted data mark, the deleted data flag is set in param block &A.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
buffer = data read
param block &A = Result Register (see Chapter 3)

bit 7,6 = 0
bit 5 = deleted data flag
bit 4,3 = completion type
bit 2,1 = completion code
bit 0 = 0

```
10 REM
2b REM  DEMONSTRATION
30 REM
40 REM OSWORD read deleted data multi-sector
50 REM
60 DIM wdata 255, rdata 255, mc% 255
70 $wdata = STRING$(255,"A") : $rdata = STRING$(255," ")
80 osword = &FFF1
90 FOR pass% = 0 TO 2 STEP 2
100 P% =  mc%
110 [OPT  pass%
120 \    write parameter block
130 .wparm EQUB 3        \ drive 3
140        EQUD wdata    \ write buffer
150        EQUB 3       \ no, of parms
160        EQUB &4F      \ command
170        EQUB 79       \ log track
180        EQUB 0        \ log sector
190        EQUB &21      \ 1 sector
200 .wres  EQUB 0        \ result
210 \    read parameter block
220 .rparm EQUB 3        \ drive 3
230        EQUD rdata     \ read buffer
240        EQUB 3        \ no. of parms
250        EQUB &57      \ command
260       EQUB 79       \ log track
270       EQUB 0        \ log sector
280       EQUB &21      \ 1 sector
290 .rres EQUB 0       \ result
300 .raddr EQUW rparm    \ addr of rparm
310 .waddr EQUW wparm    \ addr of wparm
320
330 .start LDA #&7F     \ function
340        LDX waddr    \ point X & Y
350        LDY waddrH   \ at wparm
360        JSR osword   \ write a sect
370        LDA wres     \ get result
380        AND #&1E     \ save it
390        BNE end      \ if bad
400 .read  LDA #&7F     \ function
410        LDX raddr    \ point X & Y
420        LDY raddr+1   \ at rparm
430        JSR osword    \ read back
440 .end   RTS ]
450 NEXT pass%
460 REM This demonstration writes 256 bytes of letter "A" from
470 REM wdata to drive 3, track 79, sector 0. It then reads this
480 REM data back again into rdata and checks it's the same.
490 CALL start
500 PRINT "write result = &";~?wres
510 PRINT " read result = &";~?rres;" note delete bit"
520 IF $rdata <> $wdata PRINT "bad"
530 END

>RUN
write result = &0
 read result = &20 note delete bit
```

OSWORD Read Sector Ids

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = address of buffer (LSB first)

param block &5 = &03
param block &6 = &5B
param block &7 = physical track number
param block &8 = 0
param block &9 = number of ids to be read (&0 to &A)

action:

The 8271 uses the appropriate Track Register as a base from which to 'seek' the specified track. It then proceeds to read off the required number of sector ids and copies them into the buffer area; the address of which you supplied in the parameter block. It does not check that the track numbers of these sector ids (logical track numbers) match the physical track number that you specified in the parameter block, but sector id CRCs are verified. For each sector id that it reads, it returns four bytes as follows:

byte 0 = logical track number
byte 1 = head number (normally 0)
byte 2 = logical sector number
byte 3 = data size (0=128,1=256,2=512,.....,7=16384)

Sector ids are transferred in physical sector order; i.e starting at the index pulse.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
buffer = sector ids
param block &A = Result Register (see Chapter 3)

    bit 7,6 = 0
    bit 5 = deleted data flag
    bit 4,3 = completion type
    bit 2,1 = completion code
    bit 0 = 0

```
10 REM
20 REM       DEMONSTRATION
30 REM
40 REM OSWORD read sector ids
50 REM
60 DIM sects 10*4, mc% 255
70 osword = &FFF1
80 FOR pass% = 0 TO 2 STEP 2
90 P% = mc%
100 [OPT  pass%
110 \    parameter block
120 .param EQUB 3       \ drive 3
130        EQUD sects    \ buffer
140        EQUB 3       \ no. of parms
150        EQUB &5B      \ command
160        EQUB 0       \ phys track
170        EQUB 0        \
180        EQUB &A       \ 10 sector
190 .res   EQUB 0        \ result
200 .paddr EQUW param    \ addr of param
210
220 .start LDA #&7F    \  function
230        LDX paddr    \ point X & Y
240        LDY paddr+1   \ at param
250        JSR osword   \ read ids
260 .end   RTS ]
270 NEXT pass%
280 REM This demonstration lists all ten sector ids for
drive 3,
290 REM track 0. Note the logical sector stagger.
300 CALL start
310 PRINT "phys-sect log-track head log-sect size" : PRINT
320 J% = 0
330 I% = sects - 1
340 REPEAT
350 PRINT "   ";~J%;
360 I% = I% + 1
370 PRINT "      ";~?I%;
380 I% = I% + 1
390 PRINT "      ";~?I%;
400 I% = I% + 1
410 PRINT "      ";~?I%;
420 I% = I% + 1
430 PRINT "      ";~?I%
```

```
440 J% = J% + 1
450 UNTIL I% > sects+38
460 END
```

```
>RUN
phys-sect Log-track head log-sect size

    0        0       0      7      1
    1        0       0      8      1
    2        0       0      9      1
    3        0       0      0      1
    4        0       0      1      1
    5        0       0      2      1
    6        0       0      3      1
    7        0       0      4      1
    8        0       0      5      1
    9        0       0      6      1
```

OSWORD Verify Data and Deleted Data 128 bytes

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &5 = &02
param block &6 = &5E
param block &7 = logical track number
param block &8 = logical sector number

action:

The 8271 uses the appropriate Track Register as a base from which to 'seek' the specified track and it then searches for a sector id which matches the track and sector number specified in the parameter block. If it cannot find the sector it returns a value of &18 to param block &9. It reads 128 bytes of data, checking the CRC at the end. It sets param block &9 accordingly. Note that the data read goes nowhere. It is only read to verify that the sector is readable.

The 8271 is designed around a sector size of 128 bytes (not 256 bytes as used by the BBC Microcomputer). In the multi-sector commands it offers the facility to vary the sector size. Thus it is most unlikely that you will want to use the 128 byte calls, as the multi-sector calls are much better suited to the format of the BBC Microcomputer.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &9 = Result Register (see Chapter 3)

bit 7,6 = 0
bit 5 = deleted data flag
bit 4,3 = completion type
bit 2,1 = completion code
bit 0 = 0

NO DEMONSTRATION PROGRAM

OSWORD Verify Data and Deleted Data Multi-sector

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &5 = &03
param block &6 = &5F
param block &7 = logical track number
param block &8 = logical sector number
param block &9 = sector size / number of sectors

The most significant three bits represent the sector size:

0	=	128 bytes per sector
1	=	256 bytes per sector

010 =	512 bytes per sector
011 =	1024 bytes per sector
100 =	2048 bytes per sector
101 =	4096 bytes per sector
110 =	8192 bytes per sector
111 =	16384 bytes per sector

The least significant five bits represent the number of sectors per track and can have any value between zero and &1F.

In practice, the Acorn DFS works with 256 bytes per sector. The allowable values for this whole field are thus:

&21 =	1 sector of 256 bytes
&22 =	2 sectors of 256 bytes
&23 =	3 sectors of 256 bytes
&24 =	4 sectors of 256 bytes
&25 =	5 sectors of 256 bytes
&26 =	6 sectors of 256 bytes
&27 =	7 sectors of 256 bytes
&28 =	8 sectors of 256 bytes
&29 =	9 sectors of 256 bytes
&2A =	10 sectors of 256 bytes

action:

The 8271 uses the appropriate Track Register as a base from which to 'seek' the specified track and it then searches for a sector id which matches the track and sector number specified in the parameter block. If it cannot find the sector it returns a value of &18 to param block &A. It reads the sector of data only to check the CRC.

If you have specified more than one sector, the 8271 adds 1 to the sector number and searches for this sector. If it finds it, it repeats the steps above. This procedure is repeated until either an error occurs, or all the sectors have been successfully verified. Param block &A is set accordingly.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &A = Result Register (see Chapter 3)

> bit 7,6 = 0
> bit 5 = deleted data flag
> bit 4,3 = completion type
> bit 2,1 = completion code bit 0 = 0

```
10 REM
20 REM  DEMONSTRATION
30 REM
40 REM OSWORD verify data & deleted data multi-sector
50 REM
60 DIM mc% &100
70 osword = &FFF1
80 FOR pass% = 0 TO 2 STEP 2
90 P% = mc%
100 [OPT pass%
110 \ parameter block
120 .param EQUB 3     \ drive 3
130        EQUD &FFFF  \ no buffer
140        EQUB 3     \ no. parms
150        EQUB &5F   \ command
160        EQUB 0     \ log track
170        EQUB 0     \ log sector
180        EQUB &2A    \ 10 sectors
190 .res   EQUB 0     \ result
200 .parad EQUW param  \ addr of param
210
220 .start LDA #&7F   \ function
230        LDX parad  \ point X & Y
240        LDY parad+1 \ at param
250        JSR osword \ OSWORD
260        RTS
270 ]
280 NEXT pass%
290 REM verify drive 3,  track 0.
```

```
300
310 CALL start
320 PRINT "result = &";~?res
330 IF (?res AND &1E) =0 PRINT "ok" ELSE PRINT "bad"
340 END

>RUN
result = &0
ok
```

OSWORD Format Track

on entry:

A = &7F

X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = address of sector table (LSB first)
param block &5 = &05
param block &6 = &63
param block &7 = physical track number
param block &8 = gap 3 size
param block &9 = sector size / number of sectors

The most significant three bits represent the sector size:

000	128 bytes per sector
001	256 bytes per sector
010	512 bytes per sector
011	1024 bytes per sector
100	2048 bytes per sector
101	4096 bytes per sector
110	8192 bytes per sector
111	16384 bytes per sector

The least significant five bits represent the number of sectors per track and can have any value between zero and &1F.

In practice, the Acorn DFS works with 256 bytes per sector. The allowable values for this whole field are thus:

&21 =	1 sector of 256 bytes
&22 =	2 sectors of 256 bytes
&23 =	3 sectors of 256 bytes
&24 =	4 sectors of 256 bytes
&25 =	5 sectors of 256 bytes
&26 =	6 sectors of 256 bytes
&27 =	7 sectors of 256 bytes
&28 =	8 sectors of 256 bytes
&29 =	9 sectors of 256 bytes
&2A =	10 sectors of 256 bytes

param block &A = gap 5 size
param block &B = gap 1 size

action:

Before you use this command, you have to create a sector table containing the sector ids that you want written on the track. Each sector id is a four byte entry consisting of:

logical track
head (use 0 normally)
logical sector
data size (0=128,1=256,2=512,..........,7=16384)

The 8271 uses the appropriate Track Register as a base from which to 'seek' the specified physical track. It then:

a) writes the sector ids in the correct part of the track, using your sector table.

b) calculates and writes the sector id CRC bytes.

c) writes the correct number of &FF bytes in the gaps using the gap sizes specified in the parameter block. See section 1.2 for suitable gap sizes.

d) fills the data field with bytes of &E5.

e) calculates and writes the data CRC bytes.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &C = Result Register (see Chapter 3)

bit 7,6 = 0
bit 5 = deleted data flag
bit 4,3 = completion type
bit 2,1 = completion code
bit 0 = 0

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSWORD format track
50 REM
55 DIM sectab 39
60 DIM mc% &100
70 osword = &FFF1
80 FOR pass% = 0 TO 2 STEP 2
90 P% = mc%
100 [OPT pass%
110 \ parameter block
120 .param EQUB 3     \ drive 3
130        EQUD sectab \ sector table
140        EQUB 5     \ no. parms
150        EQUB &63    \ command
160        EQUB 79     \ phys track
170        EQUB 21     \ gap 3
180        EQUB &2A    \ 10 sectors
182        EQUB 0     \ gap 5
186        EQUB 16    \ gap 1
190 .res   EQUB 0     \ result
200 .parad EQUW param  \ .addr of param
210
220 .start LDA #&7F   \ function
230        LDX parad  \ point X & Y
240        LDY parad+1 \ at param
250        JSR osword \ OSWORD
260        RTS
```

```
270 ]
280 NEXT pass%
290 REM create sectab and format drive 3, track 79.
300
310 FOR I% = 0 TO 9
320    sectab?(I%*4)  =79
330    sectab?((4*I%)+1> = 0
340    sectab?((4*I%)+2) = I%
350    sectab?((4*I%)+3) = 1
360 NEXT
370 CALL start
380 PRINT "result = &";~?res
390 IF ?res = 0 PRINT "ok" ELSE PRINT "bad"
400 END

>RUN
result = &O
ok
```

OSWORD Seek

on entry:

A = &7F

X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &5 = &01
param block &6 = &69
param block &7 = physical track number

action:

The 8271 uses the appropriate Track Register as a base from which to 'seek' the specified physical track, but it does not load the head and therefore does not check the sector ids for correct track. Note that if param block &7 is zero, the 8271 adopts a special procedure. It steps the head outwards until it trips the track 0 switch. It is prepared to try 255 of these steps. If the TRK0 signal is still missing, it assumes that the drive is faulty.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &8 = Result Register (see Chapter 3)

> bit 7,6 = 0
> bit 5 = deleted data flag
> bit 4,3 = completion type
> bit 2,1 = completion code
> bit 0 = 0

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSWORD seek track
50 REM
60 DIM mc% &100
70 osword = &FFF1
80 FOR pass% = 0 TO 2 STEP 2
90 P% = mc%
100 [OPT pass%
110 \ parameter block
120 .param EQUB 3     \ drive 3
130        EQUD &FFFF  \ no buffer
140        EQUB 1     \ no. parms
150        EQUB &69    \ command
160        EQUB 0     \ phys track
190 .res   EQUB 0     \ result
200 .parad EQUW param  \ addr of param
210
220 .start LDA #&7F   \ function
230        LDX parad  \ point X & Y
240        LDY parad+1 \ at param
250        JSR osword  \ OSWORD
260        RTS
270 ]
280 NEXT pass%
290 REM seek drive 3, track 0.
300
310 CALL start
```

```
320 PRINT "result = &";~?res
330 IF ?res = 0 PRINT "ok" ELSE PRINT "bad"
340 END

>RUN
result = &0
ok
```

OSWORD Read Drive Status

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &5 = &00
param block &6 = &6C

action:

The 8271 has a Drive Control Input Register in which it maintains the current status of the control signals sent to it by the disk drive. When it receives this command, it simply copies this register to the Result Register, which OSWORD then makes available to you at param block &7. Each bit has a meaning as follows:

bit 7	unused
bit 6	READY1
bit 5	FAULT
bit 4	INDEX
bit 3	WR PROTECT
bit 2	READY0
bit l	TRK0
bit 0	COUNT/OP1

These signals are described in Chapter 3. It is unlikely that you will want to use this command yourself, because the DFS does this automatically for you to ensure that the disk drive is healthy. The command is useful to a computer repair specialist, however.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &7 = Drive Status

```
10 REM
20 REM  demonstration
30 REM
40 REM OSWORD read drive status
50 REM
60 DIM mc% &100
70 osword = &FFF1
80 FOR pass% = 0 TO 2 STEP 2
90 P% = mc%
100 [OPT pass%
110 \ parameter block
120 .param EQUB 3     \ drive 3
130        EQUD &FFFF  \ no buffer
140        EQUB 0     \ no. parms
150        EQUB &6C    \ command
190 .res   EQUB 0     \ result
200 .parad EQUW param  \ addr of param
210
220 .start LDA #&7F   \ function
230        LDX parad  \ point X & Y
240        LDY parad+1 \ at param
250        JSR osword \ OSWORD
260        RTS
270 ]
280 NEXT pass%
290 REM read drive status for drive 3.
300 CALL start
310 mask% = 256
320 FOR I% = 0 TO 7
330   mask% = mask% DIV 2
340    READ S$
350    IF I%  = 0 GOTO 400
360    level% = ?res AND mask%
370    status$ = "down"
380    IF level% <> 0 status$ = "up"
390    PRINT S$;" = ";status$
```

```
400 NEXT
410 END
420 DATA "          ","READY1    "
430 DATA "FAULT     ","INDEX     "
440 DATA "WR PROTECT","READY0    "
450 DATA "TRKO      ","C0UNT/0P1 "
```

```
>RUN
READY1        = up
FAULT         = down
INDEX         = up
WR PROTECT    = down
READY0        = down
TRK0          = down
C0UNT/0P1     = up
```

OSWORD Initialise 8271

on entry:

A = &7F

X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &5 = &04
param block &6 = &75
param block &7 = &0D
param block &8 = step time
param block &9 = settling time
param block &A = unload time / load time

(see Chapters 2 and 3 for a description of these parameters)

action:

The 8271 saves these parameters and uses them to time its future actions. This command is drive independent, though you have to stick to the standard OSWORD parameter block. The DFS uses this command, on hard reset, to pass on the values implied by links 3 and 4 to the 8271. See Chapter 2 for a related OSBYTE call that you can experiment with.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &B = Result Register (see Chapter 3)

bit 7,6 = 0
bit 5 = deleted data flag
bit 4,3 = completion type
bit 2,1 = completion code
bit 0 = 0

```
10 REM
20 REM      DEMONSTRATION
30 REM
40 REM OSWORD initialise 8271
50 REM
60 DIM mc% &100
70 osword = &FFF1
80 FOR pass% = 0 TO 2 STEP 2
90 P% = mc%
100 [ OPT pass%
110 \ parameter block
120 .param EQUB 0     \ drive
130        EQUD &FFFF  \ no buffer
140        EQUB 4     \ no. parms
150        EQUB &75    \ command
160        EQUB &D    \ tie-breaker
170        EQUB 2     \ 2*step
180        EQUB 8     \ 2*settle
182        EQUB &C0   \ unload/load
190 .res   EQUB 0     \ result
200 .parad EQUW param  \ addr of param
210
220 .start LDA #&7F   \ function
230        LDX parad  \ point X & Y
240        LDY parad+1 \ at param
250        JSR osword \ OSWORD
260        RTS
270 ]
280 NEXT pass%
```

```
290 REM *SAVE using the slow, power up defaults.
300 REM (step=24, settle=20,unload=12,load=64)
310 TIME = 0
320 *SAVE DUMMY 1900 +5000
330 T1 = TIME
340 *DELETE DUMMY
350 REM reset disk timings to
360 REM (step=4,settle=16,unload=12,load=0)
370 REM and rerun *SAVE.
380 CALL start
390 TIME = 0
400 *SAVE DUMMY 1900 +5000
410 T2 = TIME
420 PRINT "slow time = ";T1/100;" sees"
430 PRINT "fast time = ";T2/100;" sees"
440 END

>RUN
slow time = 3.54 sees
fast time = 2.71 sees
```

OSWORD Load Bad Tracks

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &04
param block &6 = &75
param block &7 = drive pair (&10 = drive 0/2, &18 = drive 1/3)
param block &8 = bad track number 1
param block &9 = bad track number 2
param block &A = current track

action:

This command allows you to tell the 8271 that there are one or two bad tracks on the diskette in a given pair of drives. It uses the value that you supply for the current track to calculate the tracks that are bad. The command lasts till a hard reset. Effectively, the 8271 will have nothing to do with these two tracks. Note that this will apply even if you swop diskettes. This is really a throwback to the bad old days when diskettes were less reliable. The DFS itself does not use this command and it is unlikely that you will want to either. Using it is likely to lead to 'Disk fault' error messages, unless you avoid using the DFS altogether.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &B = Result Register (see Chapter 3)

> bit 7,6 = 0
> bit 5 = deleted data flag
> bit 4,3 = completion type
> bit 2,1 = completion code
> bit 0 = 0

NO DEMONSTRATION PROGRAM

12.4 OSWORD &7F Write Special Register

The 8271 allows you to overwrite its own internal registers which it uses in its normal operation. In some instances this can be really helpful as it allows you to perform a number of tricks that would not otherwise be possible. The parameter block that you use for this is as follows:

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &02
param block &6 = &7A
param block &7 = register number (see Figure 18)
param block &8 = value to put in register

I can see no point in overwriting some of these registers; on the contrary, it seems to me to be a particularly dangerous thing to do. The following descriptions are confined to those

Special Registers where there may be some value in overwriting them, however unlikely. The only Special Registers that you will definitely want to overwrite are the Track Registers. The Chapter on utility programs deals with this in depth. Thus no demonstration programs are offered for this section.

OSWORD Write Bad Track Register 1 - Drive 0/2

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &02
param block &6 = &7A
param block &7 = &10
param block &8 = value to put in register

action:

This command allows you to alter the contents of the 8271 Special Register, the Bad Track Register 1 - Drive 0/2. This would allow you to specify the first of two bad track numbers for that drive pair, but you could equally use the Load Bad Track command which you should consult for an explanation.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &9 = Result Register (see Chapter 3)

    bit 7,6 = 0
    bit 5 = deleted data flag
    bit 4,3 = completion type
    bit 2,1 = completion code
    bit 0 = 0

NO DEMONSTRATION PROGRAM

OSWORD Write Bad Track Register 2 - Drive 0/2

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &02
param block &6 = &7A
param block &7 = &11
param block &8 = value to put in register

action:

This command allows you to alter the contents of the 8271 Special Register, the Bad Track Register 2 - Drive 0/2. This would allow you to specify the second of two bad track numbers for that drive pair, but you could equally use the Load Bad Track command which you should consult for an explanation.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &9 = Result Register (see Chapter 3)

    bit 7,6 = 0
    bit 5 = deleted data flag
    bit 4,3 = completion type
    bit 2,1 = completion code
    bit 0 = 0

NO DEMONSTRATION PROGRAM

OSWORD Write Track Register - Drive 0/2

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &02
param block &6 = &7A
param block &7 = &12
param block &8 = value to put in register

action:

This command allows you to alter the contents of the 8271 Special Register, the Track Register - Drive 0/2. The 8271 uses this register to identify its current track position for the diskette in drive 0/2. It is most useful to be able to change this register as it gives you a way of handling logical sectoring. Suppose you tell the 8271 to seek track 5 of the diskette in drive 0 and then to read all the sector ids on that track. Suppose as a result of this you discover that the sector ids all have a logical track number of 8. All you have to do is reset the track register to 8 and you can now read the track. Remember to set it back to 5 afterwards.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &9 = Result Register (see Chapter 3)

    bit 7,6 = 0
    bit 5 = deleted data flag bit 4,3 =
    completion type bit 2,1 =
    completion code bit 0 = 0

NO DEMONSTRATION PROGRAM

(Overwriting track registers is an important technique that helps us to handle logical sectoring. See the chapter on Utility Programs for more information.)

OSWORD Write Mode Register

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &02
param block &6 = &7A
param block &7 = &17
param block &8 = value to put in register

action:
This command allows you to alter the contents of the 8271 Special Register, the Mode Register. The format of the Mode Register is:

bit 7,6 must both be 1
bit 5,4,3,2 must all be zero
bit 1 = stepper mode (0 = double, 1 = single).
bit 0 must be 1 (specifies non-DMA)

You must not alter any of the bits from 7 to 2, nor bit 0. You may alter bit 1 if you have trawled the junk yards for your disk drive and found a double-sided drive with only one head stepper mechanism, so that the two heads move in unison on either side of the diskette. If you have one of these rarities you will have to change bit 1 from its default value of 0 to 1. Thus the Mode Register defaults to &C1, but you may change it to &C3 if you have one of these strange drives. Under no other circumstances should you use this command.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &9 = Result Register (see Chapter 3)

bit 7,6 = 0
bit 5 = deleted data flag

bit 4,3 = completion type
bit 2,1 = completion code
bit 0 = 0

NO DEMONSTRATION PROGRAM

OSWORD Write Bad Track Register 1 - Drive 1/3

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &02
param block &6 = &7A
param block &7 = &18
param block &8 = value to put in register

action:

This command allows you to alter the contents of the 8271 Special Register, the Bad Track Register 1 - Drive 1/3. This would allow you to specify the first of two bad track numbers for that drive pair, but you could equally use the Load Bad Track command which you should consult for an explanation.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &9 = Result Register (see Chapter 3)

bit 7,6 = 0
bit 5 = deleted data flag
bit 4,3 = completion type
bit 2,1 = completion code
bit 0 = 0

NO DEMONSTRATION PROGRAM

OSWORD Write Bad Track Register 2 - Drive 1/3

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &() = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &02
param block &6 = &7A
param block &7 = &19
param block &8 = value to put in register

action:

This command allows you to alter the contents of the 8271 Special Register, the Bad Track Register 2 - Drive 1/3. This would allow you to specify the second of two bad track numbers for that drive pair, but you could equally use the Load Bad Track command which you should consult for an explanation.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &9 = Result Register (see Chapter 3)

bit 7,6 = 0
bit 5 = deleted data flag
bit 4,3 = completion type
bit 2,1 = completion code
bit 0 = 0

NO DEMONSTRATION PROGRAM

OSWORD Write Track Register - Drive 1/3

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &02
param block &6 = &7A
param block &7 = &1A
param block &8 = value to put in register

action:

This command allows you to alter the contents of the 8271 Special Register, the Track Register - Drive 1/3. The 8271 uses this register to identify its current track position for the diskette in drive 1/3. It is most useful to be able to change this register as it gives you a way of handling logical sectoring. Suppose you tell the 8271 to seek track 5 of the diskette in drive 1 and then to read all the sector ids on that track. Suppose as a result of this you discover that the sector ids all have a logical track number of 8. All you have to do is reset the track register to 8 and you can now read the track. Remember to set it back to 5 afterwards.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &9 = Result Register (see Chapter 3)

    bit 7,6 = 0
    bit 5 = deleted data flag
    bit 4,3 = completion type
    bit 2,1 = completion code
    bit 0 = 0

NO DEMONSTRATION PROGRAM

(Overwriting track registers is an important technique that helps us to handle logical sectoring. See the chapter on Utility Programs for more information.)

12.5 OSWORD &7F Read Special Register

The 8271 allows you to read its own internal registers which it uses in its normal operation. In some instances this can be really helpful as a diagnostic aid. The parameter block for these commands is as follows:

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &01
param block &6 = &7D
param block &7 = register number (see Figure 18)

In all cases the 8271 transfers the contents of its Special Register to the Result Register, which OSWORD then passes to you in param block &8. The following demonstration program reads all the Special Registers. The commands are then presented one-by-one.

```
10 REM
20 REM       DEMONSTRATION
30 REM
40 REM OSWORD read the special registers
50 REM
60 DIM mc% &100
70 osword = &FFF1
80 FOR pass% = 0 TO 2 STEP 2
90 P% = mc%
100 [ OPT  pass%
110 \    parameter block
120 .param EQUB 1       \ drive 1
130        EQUD &FFFF    \ no buffer
HO         EQUB 1        \ no. of parms
150        EQUB &7D      \ command
160        EQUB 0       \ register
170 .res   EQUB 0        \ result
180 .parad EQUW param    \ addr of param
190
200 .start LDA &70 \ get reg
210        STA param+7 \ save it
220        LDA #&7F     \ function
230       LDX parad    \ point X & Y
240       LDY parad+1   \ at param
250       JSR osword    \ OSWORD
260 .end RTS
270 ]
280 NEXT pass%
290 REM Read Special Registers. Print contents.
300
310 DIM R$(9),reg%(9)
320 FOR I% = 0 TO 9
330    READ R$(I%)
340    READ reg%(I%)
350 NEXT
360 FOR I% = 0 TO 9
370    ?&70 = reg%(I%)
380    CALL start
390    PRINT R$(I%);" = &";~?res
400 NEXT
410 END
420 DATA "Scan Sector Register",&06
430 DATA "Bad Track 1 Reg - Drive 0/2",&10
440 DATA "Bad Track 2 Reg - Drive 0/2",&11
450 DATA "Track Register Drive 0/2",&12
460 DATA "Mode Register",&17
470 DATA "Bad Track Reg 1 - Drive 1/3",&18
480 DATA "Bad Track Reg 2 - Drive 1/3",&19
490 DATA "Track Register Drive 1/3",&1A
500 DATA "Drive Control Input",&22
510 DATA "Drive Control Output",&23

>RUN
Scan Sector Register = &A8
Bad Track Reg 1 - Drive 0/2 = &FF
Bad Track Reg 2 - Drive 0/2 = &FF
Track Register - Drive 0/2 = &0
Mode Register = &C1
Bad Track Reg 1 - Drive 1/3 = &FF
Bad Track Reg 2 - Drive 1/3 = &FF
Track Register - Drive 1/3 = &3
Drive Control Input = &D1
Drive Control Output = &A8
```

OSWORD Read Scan Sector Register

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &01
param block &6 = &7D
param block &7 = &06

action:

This command allows you to read the contents of the 8271 Special Register, the Scan Sector Register. The contents of this register are normally unpredictable. However, when a multi-sector command fails due to a CRC error, the Scan Sector Register contains the offending sector number.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &8 = Scan Sector Register

OSWORD Read Bad Track Register 1 - Drive 0/2

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block
param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &01
param block &6 = &7D
param block &7 = &10

action:

This command allows you to read the contents of the 8271 Special Register, the Bad Track Register 1 - Drive 0/2, which contains the first of two bad track numbers for that drive pair.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &8 = Bad Track Register 1 - Drive 0/2

OSWORD Read Bad Track Register 2 - Drive 0/2

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &01
param block &6 = &7D
param block &7 = &11

action:

This command allows you to read the contents of the 8271 Special Register, the Bad Track Register 2 - Drive 0/2 which contains the second of two bad track numbers for that drive pair,

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &8 = Bad Track Register 2 - Drive 0/2

OSWORD Read Track Register - Drive 0/2

on entry:

A = &7F
X = LSI3 of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &01
param block &6 = &7D
param block &7 = &12

action:

This command allows you to read the contents of the 8271 Special Register, the Track Register - Drive 0/2. The 8271 uses this register to identify its current track position for that drive pair.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &8 = Track Register- Drive 0/2

OSWORD Read Mode Register

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &01
param block &6 = &7D
param block &7 = &17

action:

This command allows you to read the contents of the 8271 Special Register, the Mode Register. The format of the Mode Register is:

bit 7,6 must both be 1
bit 5,4,3,2 must all be zero
bit 1 = actuator mode (0 = double, 1 = single).
bit 0 must be 1 (specifies non-DMA)

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &8 = Mode Register

OSWORD Read Bad Track Register 1 - Drive 1/3

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &01
param block &6 = &7D
param block &7 = &18

action:

This command allows you to read the contents of the 8271 Special Register, the Bad Track Register 1 - Drive 1/3 which contains the first of two bad track numbers for that drive pair.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &8 = Bad Track Register 1 - Drive 1/3

OSWORD Read Bad Trick Register 2 - Drive 1/3

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &01
param block &6 = &7D
param block &7 = &19

action:

This command allows you to read the contents of the 8271 Special Register, the Bad Track Register 2 - Drive 1/3 which contains the second of two bad track numbers for that drive pair.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &8 = Bad Track Register 2 - Drive 1/3

OSWORD Read Track Register - Drive 1/3

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block
param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &7 = &01
param block &6 = &7D
param block &7 = &1A

action:

This command allows you to read the contents of the 8271 Special Register, the Track Register - Drive 1/3. The 8271 uses this register to identify its current track position for that drive pair.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &8 = Track Register - Drive 1/3

OSWORD Read Drive Control Input Register

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &5 = &01
param block &6 = &7D
param block &7 = &22

action:

The 8271 has a Drive Control Input Register in which it maintains the current status of the control signals sent to it by the disk drive. When it receives this command, it simply copies this register to the Result Register, which OSWORD then makes available to you at param block &8. Each bit has a meaning as follows:

bit 7 = unused
bit 6 = READY1
bit 5 = FAULT
bit 4 = INDEX
bit 3 = WR PROTECT

bit 2 = READY0
bit 1 = TRK0
bit 0 = COUNT/OP1

These signals are described in Chapter 3, but the 8271 presents them all in positive logic form (0 = false, 1 = true). It is unlikely that you will want to use this command yourself, because the DFS does this automatically for you to ensure that the disk drive is healthy. The command is useful to computer repair specialists, however.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &8 = Drive Control Input Register

OSWORD Read Drive Control Output Register

on entry:

A = &7F
X = LSB of address of param block
Y = MSB of address of param block

param block &0 = drive number (0 to 3)
param block &1 to &4 = does not matter
param block &5 = &01
param block &6 = &7D
param block &7 = &23

action:

The 8271 has a Drive Control Output Register in which it maintains the current status of the control signals which it outputs. When it receives this command, it simply copies this register to the Result Register, which OSWORD then makes available to you at param block &8. Each bit has a meaning as follows:

bit 7 = SELECT 1
bit 6 = SELECT 0
bit 5 = FAULT RESET/OP0
bit 4 = LOW CURRENT
bit 3 = LOAD HEAD
bit 2 = DIRECTION
bit 1 = SEEK/STEP
bit 0 = WR ENABLE

These signals are described in Chapter 3. It is unlikely that you will want to use this command yourself, except possibly as a diagnostic aid in the event of hardware failure.

on exit:

A = unchanged
X = unchanged
Y = unchanged
P = destroyed
param block &8 = Drive Control Output Register

13 NMI INTERRUPTS

When the BBC Microcomputer is running, there are many background tasks that need to be tackled as a matter of urgency. The updating of the microcomputer's internal clocks is such a task. Any data input/output requires prompt attention also. To cope with this, the 6502 has a built-in interrupt facility. Any piece of hardware that requires prompt attention signals an interrupt to the 6502 processor which then:

> completes its current instruction.
>
> saves all the registers on the stack, so that later it can go back to doing what it was doing. Note that this includes the Program Counter Register, so it keeps a record of where to return.
>
> fetches the address of the interrupt handler routine from a pre-ordained location,
>
> jumps to the interrupt handler routine.

The interrupt handler routine then services the interrupt and exits with a RTI (return from interrupt) instruction. The RTI causes the 6502 to restore all the registers from the stack and to continue at exactly the point it left off, with the registers containing exactly what they did prior to the interrupt. It is as though the interrupt never happened at all.

In fact the 6502 can accommodate three different types of interrupt; that is to say it has three pins through which it can receive different interrupt signals.

The first of these is known as RESET, and this interrupt occurs when you switch your microcomputer on, or when you hit BREAK on the keyboard.

The second of these is known as IRQ and is mainly used by the various ports and input/output devices to signal a fault or a request for an input/output operation. It can also be caused by software in the form of the BRK instruction.

The third of these is known as NMI (non-maskable interrupt) and its use is confined to high-speed devices such as the disk subsystem or the ECONET subsystem.

IRQ interrupts, however are maskable. This means that they can be disabled by the SEI instruction and enabled by the CLI instruction. Their respective interrupt handler routines use the SEI instruction to prevent the condition where one interrupt interrupts another. Having said that, no interrupt handler routine should run with interrupts disabled for very long. The NMI interrupt, as the name implies, cannot be disabled. The 6502 must honour all NMI interrupts, so this class of interrupt is the top priority reserved only for high speed devices. An important task for all interrupt handlers is to clear the interrupt condition at the end. This usually involves resetting an addressable, internal register in the device that caused the interrupt in the first place.

When an interrupt occurs, the 6502 is designed to fetch the interrupt handler address from ROM as follows below. Also shown are the addresses contained therein for OS 1.20.

```
NMI    fetches from &FFFA,&FFFB which contains &0D00
RESET fetches from &FFFC,&FFFD which contains &D9CD
IRQ    fetches from &FFFE,&FFFF which contains &DC1C
```

This address fetch is built into the 6502 hardware and cannot be suppressed. Of course, different versions of the OS ROM mav contain different jump addresses. The ones shown are true for OS 1.20.

Thus when an NMI interrupt occurs, the 6502 saves the current environment in anticipation of continuing later, and jumps to &0D00. If you enter TAPE and then examine location &0D00, you will see that it simply contains an RTI instruction. Any NMI interrupts are just returned straight back, as they are not expected.

The functions of the different interrupt handler routines are somewhat different. The RESET interrupt handler has first to determine the nature of the interrupt. Was it caused by switching on? If not, was it BREAK, SHIFT BREAK or CONTROL BREAK? The actions that it takes depend on the answers to these questions.

The IRQ handler has first to decide whether the interrupt was caused by a BRK instruction, or by some input/output device. If it was an input/output device it has to poll each one in turn to discover which device issued the interrupt. In doing this, it approaches the most likely devices first and the least likely last. It is possible that the interrupt was caused by some user device about which the handler has no knowledge. In this event, it must pass control to a user-supplied IRQ handler through the IRQ2V vector at &206,&207.

The NMI handler has no such problems. In a standard BBC Microcomputer, only one subsystem can be using NMI interrupts at a time. A paged ROM makes a claim for the right to handle NMI interrupts by issuing a service call (type &0C). The DFS does not have one single NMI interrupt handler; it has several. When it is about to issue a command to the 8271 FDC, it knows which of its interrupt handlers will be appropriate for the NMI interrupt that will eventually happen. Thus it downloads this code to &0D00 prior to issuing the command.

There is, of course, no need to write an NMI interrupt handler for yourself. The Acorn DFS support is truly excellent, allowing you to execute all the 8271 commands without resorting to this level of tedium. However, should you wish to write an NMI handler for yourself, maybe for purely educational purposes, you may find the following advice helpful:

> Do not try your routine on a diskette containing valuable data; at least not until you are sure your code works.
>
> Write the NMI handler to run at &D00.
>
> Do use an SEI instruction at the start of your routine to disable maskable interrupts.
>
> Do not use OSBYTE calls or any other system calls. This is one time when you are positively encouraged to peek and poke memory areas, as speed is of the essence.
>
> Do not cause another NMI interrupt. Do no disk I/Os, for example.

Do not worry about Tube compatibility. All interrupt handler code must perforce be in the I/O processor, so you can ignore the Tube altogether.

Above all - be quick and efficient! One hundred machine-code instructions should be more than enough. Try to make it less! BASIC is, of course, out of the question.

Do remember to clear the interrupt condition, by reading the 8271 FDC Result Register.

Do remember to use a CLI instruction near the end of your routine to re-enable maskable interrupts.

Do exit with an RTI instruction.

Before you download the NMI handler to &D00, you must claim the NMI space. You do this with the following code:

```
1000   LDA #&8F      \ issue service call
1010   LDX #&0C      \ type &C
1020   LDY #&FF      \ no ROM
1030   JSR &FFF4     \ claim NMI space
1040   TYA           \ save ROM number
1050   STA &70      \ that you pinched NMI space from
```

When your NMI handler has finished , you must free the NMI space as follows:

```
2000   LDY &70       \ get ROM
2010   BMI out       \ if none - don't bother
2020   LDA #&8F      \ issue service call
2030   LDX #&0B      \ type 6B
2040   JSR &FFF4     \ free NMI space
2050   .out   \ exit
```

14 Acorn DFS and BASIC

This chapter explains all of the BASIC commands that interface with the Acorn DFS. You can, however, enormously widen the scope of your disk handling by using the various low level calls to the DFS described in the previous chapters. Although these calls are primarily intended for machine-code programs, they can be used in BASIC programs and this chapter tells you how. There are three ways of processing disk files:

A whole file at a time
A group of bytes at a time
A byte at a time

A whole file can be processed by using the DFS star commands as already discussed. If the file is a BASIC program, you can also use CHAIN, LOAD and SAVE. LOAD will copy a BASIC program from disk to memory, starting at PAGE. SAVE reverses this. CHAIN is a combination of LOAD and RUN. Remember that BASIC insists that you enclose filenames in quotation marks. For example:

```
CHAIN "PROG" or
LOAD "PROG" or
SAVE "PROG"
```

For data files, BASIC programs use the commands which process a single byte, or a group of bytes. These files must be opened before they can be used.

14.1 Opening Disk Files in BASIC II

The Acorn DFS allows you to access up to five disk files at a time. Before you can start to process a file, you must first open it, using one of the three open commands (OPENIN, OPENOUT or OPENUP) specifying the filename that you want to open. The three commands that can be used in BASIC II tell the DFS how you want to use the file:

OPENIN = You only want to read the file.
OPENOUT = You only want to write the file.
OPENUP = You want to both read and write to the file.

Straightaway, the DFS works on your behalf. If you use OPENIN or OPENUP, then the file must have an existing entry in the disk catalogue. You cannot read what is not there. When you use the OPENOUT command, you are telling the DFS that you want to create a brand new file. Consequently, the DFS attempts to delete that filename from the catalogue.

There exists a small problem if you want to create a brand new file into which you intend to write new records, but from which you intend to read those records in the same run. You have to use OPENOUT to create it first, close it, and then re-open it using OPENUP.

A much larger problem also exists with OPENUP which in theory allows you to increase the file size (providing there is sufficient disk space) by making PTR# larger than EXT#. However DFS 0.90 has a serious bug (see 23.5) which means that care is needed when doing this.

OPENOUT does different things in different circumstances. If you OPENOUT a file that does not already exist, the DFS allocates 64 sectors to it initially. If you now OPENOUT another file on that diskette prior to closing the original file, the original file will retain its initial allocation of 64 sectors for ever. If, however, you close the original file prior to opening any others on the diskette, it will be allocated the number of sectors needed to contain the data as at close time. If you OPENOUT a file that already exists, the data is lost, but the new file is allocated the same space as the original file of that name. Because of the PTR# bug, many authors tell you to create a file of the correct size (using *SAVE, for example) and then use OPENOUT. For example, if you calculate that a file called 'DATA' will never exceed &4000 bytes in length, you are told to first enter:

```
*SAVE "DATA" 0 +4000
```

When your program opens this file via OPENOUT, the initial allocation of &4000 bytes will be retained. Of course, if you only write &1000 bytes of data to it, the remaining &3000 bytes allocated will be full of rubbish. This effectively renders two other BASIC commands (EXT# and EOF#) extremely dangerous in most practical circumstances. It is much better to read 23.5 and overcome the PTR# bug.

If you use either the OPENOUT or OPENUP commands you are expressing the intention of writing to that file. The DFS makes sure that the disk is not write-protected (sticky paper over the write-protect notch). It also checks that your file is not already on the disk with locked status.

Another important check by the DFS is for the number of files that you currently have open, which must not exceed five. Assuming that all is well, however, the DFS will allocate a unique number to your file, known as the file handle. DFS file handles are as follows for DFS 0.90 and 1.20, but these may change in later releases:

1st opened file handle = &11
2nd opened file handle = &12
3rd opened file handle = &13
4th opened file handle = &14
5th opened file handle = &15

If the DFS was unable to open your file, it returns a file handle of zero. The terms 'file handle' and 'channel' appear to be interchangeable. If you make a mistake with a file handle the DFS responds with the 'Channel' message. The file handle really consists of two halves. The first hexadecimal digit (always 1) defines a disk file (other filing systems have different handles). The second hexadecimal digit (1 to 5) defines whether it is the first, second, third, fourth or fifth opened file. This is important, because each file uses different memory areas for file-related tags and for buffer space into which data can be stored. The file handle allocated to each open file ensures that the DFS uses the correct memory areas for that file henceforth.

All other BASIC disk commands expect you to save the file handle that you were given at open time, and henceforth when you want to access that file, you refer to it only by that file handle. This makes sense. If you used filenames with every BASIC command, then the DFS would have to repeat a lot of tedious filename checks for each command. This would severely reduce the speed of the disk subsytem.

Thus in BASIC II, to open a file called 'DATA3', you would simply enter one of the three commands below:

```
100 handle = OPENINC'DATAS")  : REM read only
100 handle = OPENOUT("DATA3") : REM write only
100 handle = OPENUP("DATA3")  : REM read/write
```

From now on, when you want to do something with this file, you refer to it by its handle.

14.2 Opening Disk Files in BASIC I

If you bought one of the older BBC Microcomputers, your BASIC ROM may be level 1. This has only two open commands (OPENIN and OPENOUT). But does this mean you cannot open a file for both input and output? On the contrary, the first of these commands does exactly that; but for some reason they called it OPENIN (not OPENUP). You cannot use the OPENUP command in BASIC I.

Interesting things happen if you write a program in BASIC I and then load it into a microcomputer that has the BASIC II ROM. If you list the program, you find that all the OPENIN statements have changed to OPENUP. If you save this program and then load it back again into the microcomputer with the BASIC I ROM, the reverse happens. All the OPENUP statements change to OPENIN. To understand what is happening, you have to understand that the ASCII character strings 'OPENIN' and 'OPENUP are not held as such in a BASIC program. BASIC replaces each of its own commands with a single byte token. These are the tokens used for the open file commands:

BASIC command	BASIC I token	BASIC II token
OPENIN	&AD	&8E
OPENOUT	&AE	&AE
OPENUP	--	&AD

When you enter 'OPENIN' in a BASIC I program, BASIC stores it as &AD. When you list the program (listing changes tokens back to ASCII strings) in BASIC II, BASIC identifies the token of &AD as OPENUP, and lists this command accordingly.

So what are the pitfalls? A program written in BASIC II that uses the OPENIN command will not run on your BASIC I machine, because the token (&8E) has no meaning in BASIC I. You have to get someone with BASIC II to change all the OPENIN statements to OPENUP statements. If you are lucky enough to have BASIC II, but you want to give your programs to friends, avoid OPENIN in case they have BASIC I ROMs.

14.3 Writing to Disk Files

There are two ways of writing to a disk file, once it has been opened. The two available commands are called BPUT# and PRINT#. BPUT# is used to write a single byte of data onto the disk. For example:

```
100 handle = OPENOUT("DATA3")
110 BPUT# handle,5
```

This will write the number 5 to the disk file. It will not be the ASCII code for the number 5 (&35), but just the plain number 5 (&05). In fact BPUT# cannot be used directly with ASCII strings at all, only with single byte numbers. For example, the ASCII character that represents the letter 'A' is &41. To write the letter 'A' as a single byte using BPUT#, you would have to enter:

```
100 handle = OPENOUT("DATA3")
110 BPUT# handle,&41
```

If you wanted to send a string of characters, held in A$, to the disk file using BPUT#, you would have to enter:

```
100 A$    = "HARRY"
110 handle = OPENOUT("DATA3">
120 FOR I%  = 1 TO LEN(A$)
130   ascii$ = MID$(A$,I%,1) : REM get next ASCII
140  number = ASC(ascii$)  : REM convert to a number
150   BPUT# handler,number    : REM write the number
160 NEXT
```

You may feel that this is all a bit tedious. It is for this reason that a second command is provided, PRINT#.

PRINT# works on one or more variables at a time, rather than one byte at a time. It will write the contents of any variable to disk, whether the variable is an integer, a floating point number or a string. In the previous example, we could have used PRINT# as follows:

```
100 A$    = "HARRY"
110 handle = OPENOUT("DATA3")
120 PRINT# handle,A$
```

But do the two small programs produce identical results? The answer is that they do not. Using BPUT#, you would have written just five bytes to disk, the ASCII codes that spell 'HARRY'. Using PRINT#, you would have written seven bytes to disk, the last 5 of which would have been the ASCII codes that spell 'YRRAH' (note that PRINT# stores all data back to front). The two extra bytes at the front would contain &00 and &05, respectively. The zero byte identifies the start of a string variable. It is there to ensure that when you read the data back at a later date, you put it into a string variable, not an integer or a floating point variable. This stops you making mistakes. Imagine the consequences of trying to read a bank balance into your program, but mistakenly reading part of someone's name. If BASIC did not trap this error for you, it could lead to all sorts of trouble. The second extra byte contains the length of the string; in other words it says that 'YRRAH' is five characters long.

You can use the PRINT# command with all three types of variable and you can output more than one variable at a time. For example:

```
100 A = 54321.9876
110 A% = 98765
120 A$ = "HARRY"
130 handle = OPENOUT("DATA3")
140 PRINT# handle,A,A%,A$
```

Floating point numbers are stored in six bytes. The first of these contains &FF and identifies the variable as a floating point. The last five bytes are the number itself, in back-to-front floating point format (i.e. the mantissa comes last).

Integer numbers are held in five bytes. The first of these contains &40 which identifies the variable as an integer. The last four bytes are the integer itself, with the most significant byte first. This is, once again, the reverse of the normal way of storing integer variables.

If you do not understand how integers and floating point numbers are held in BBC BASIC, you may care to read 'The Advanced BASIC ROM User Guide', which covers this topic in depth.

In summary, PRINT# handles the three different variable types as follows:

> floating point = &FF + five byte number (back to front)
> integer = &40 + four byte number (back to front)
> string = &00 + length + string (back to front)

Thus the overall size of each type of variable on disk is:

> floating point = 6
> integer = 5
> string = length of string + 2

BPUT# puts you to more trouble, and is slower than PRINT#, but you retain total control over the data that is written. PRINT# gains in convenience but works to a strict formula and you sacrifice this control.

14.4 Reading from Disk Files

As you might expect, BPUT# and PRINT# have their exact counterparts when it comes to reading files (BGET# and INPUT#).

You use BGET# to read a single byte from disk. Once again, the variable into which you place this byte must not be a string variable. For example to read a single byte into A%, you simply enter:

```
100 handle = OPENUP("DATA3")
110 k%  = BGET# handle
```

INPUT# is used to read an entire variable and reverses the order of the data so that data in memory is now in the correct order. Like PRINT#, INPUT# can handle more than one variable at a time. For example:

```
100 handle = OPENUP("DATA3")
110 INPUT# handle,A,A%,AS
```

This command asks BASIC to read a floating point number, then an integer and last a string into A, A% and A$ respectively. BASIC will check the extra bytes (variable types) to ensure that the file matches your expectations. A mismatch on variable tvpe will be reported by BASIC.

14.5 Closing Disk Files

When you have completely finished with a file, you have to 'CLOSE#' it. You do this by entering:

```
100 CLOSE# handle
```

CLOSE# does a number of important things for you. The BPUT# and PRINT# commands in your program cause data to be written to a buffer: an area of memory reserved for the use of that file. This buffer gets written to disk by the DFS when it is full. CLOSE# will determine whether or not you have a partially filled buffer that has not yet been written to disk, and if so write the data out. It will also update the catalogue entries that say how big your file is. Lastly, CLOSE# will free up the file handle that you were allocated, thereby enabling you to re-use that handle if you wish to open a different file.

Note that you can close all currently opened files with a single instruction:

```
100 CLOSE# 0
```

14.6 Position in a Disk File

So far, this chapter has described disk input/output in a fairly glib manner. How does the DFS know where on the disk to read or write? The answer is that the DFS maintains a current position pointer for each opened file, which is accessible to BASIC via the PTR# command. This pointer holds the position in the file, in bytes relative to the start of the file. When you open a file the pointer is initialised to 0. You can examine the contents of the pointer at any time by entering, for example:

```
100 PRINT PTR# handle
```

Let's see what happens if we write some data to disk:

```
100 A = 54321.9876
110 A% = 98765
120 A$ = "HARRY"
130 handle = OPENOUT("DATA3")
140 PRINT "on open, PTR = ";PTR# handle
150 PRINT# handle,A
160 PRINT "after A PTR = ";PTR# handle
170 PRINT# handle,A%
180 PRINT "after A% PTR = ";PTR# handle
190 PRINT# handle,A$
200 PRINT "after A$ PTR = ";PTR# handle
210 CLOSE# handle
220 END

>RUN
on open, PTR = 0
after A PTR = 6
after A% PTR = 11
after A$ PTR =18
```

When the file is opened, PTR is reset to zero. When we write a floating point variable (six bytes long), these bytes are written to disk bytes 0 to 5 and PTR now contains 6. When we write an integer (five bytes long), these bytes are written to disk bytes 6 to 10 and PTR now contains 11. The seven byte long A$ occupies disk bytes 11 to 17 and PTR becomes 18. Thus at each stage, PTR contains the next location to be written. When reading from a file PTR contains the next location to be read.

You can process disk files from start to finish, just as you do cassette files. A file used like this is called by several different names:

Flat File
List File
Serial File
Sequential File

They all mean the same; that the file is read starting at the first byte and finishing at the last byte. The only way to access a particular bvte in the file is to access all the bytes that come before it first. When you process a disk file serially, the DFS maintains and uses the position pointer automatically. You do not need to worry about it.

With disk files, you also have the capacity to read the data in any order, starting at any point. Such files are known as 'Random Access Files' because of this. The way that you can do this in BASIC is to reset the value of the pointer to the byte that you want to read. Suppose we take the example above, where we wrote a floating point number, an integer and then a string. Suppose we want now to read the string only. We could enter:

```
100 handle = OPENUP("DATA3")
110 PTR# handle = 11 : REM point to start of string
120 INPUT# handle,A$
130 PRINT A$
>RUN
HARRY
```

As a means of demonstrating how PTR# works, this example is satisfactory. A slightly more realistic example appears in section 14.8.

14.7 Size of a Disk File and End of File

The DFS maintains, for each open file, a size flag, which points to the location following the last byte of data on the file. It uses this flag to trap program errors, such as trying to read past the end of the file. BASIC programs can access this pointer using the EXT# command, but you cannot alter its contents via BASIC. For example, to find out the size of an opened file you could enter:

```
100 PRINT EXT# handle
```

BASIC also gives you a command, EOF#, which allows you to test for an end of file condition. The EOF# statement returns the condition, TRUE, if end of file has been reached. Otherwise, it returns the condition, FALSE. The following example, which simply skips through a file from start to finish, illustrates its use:

```
100 handle = OPENUP("FILE")
110 REPEAT
120 A% = BGET# handle
130 UNTIL E0F# handle
```

14.8 Random Access Example

Suppose that you are a keen philatelist and you wish to store information about your collection of about 4000 stamps in your computer. The data that you decide to hold for each stamp comprises:

Stamp Number.
You have given the matter some thought and you realise that it would be tedious to key in 'UK Penny Black' etc. to see what data you hold for the Penny Black. There would be problems of lower case and upper case letters; how many spaces between words etc. Hence, in your album, you decide to write a unique, sequential number for each stamp, starting at stamp number 1 and going on up to stamp number 4000 or so, which you can then use as a 'key' to identify each stamp record uniquely.

Stamp Description.
You intend to put a description such as 'UK Penny Black' in here.

Value.
Every Christmas you are presented with a new catalogue and you want to enter the current value of each of your stamps.

Let's go through the various design stages that you would need in order to implement this system using only simple BASIC.

14.8.1 Sizing

The first thing you need to do is to estimate some sizes to establish if it is possible to do what you want. Each set of data that relates to an individual stamp is known as a record, and all the records together constitute the file which we shall call 'STAMPS'. Let's establish the record size first. The first field, Stamp Number, you decide to make an integer, so it will occupy 5 bytes. The second field, Stamp Description, will be a string. You think about it and decide that 20 characters will be big enough to hold the largest description that you want to enter. The maximum size of this string is therefore 22. You decide to hold the value in pence, to avoid rounding errors, so the Value field can also be integer and will therefore occupy 5 bytes. Thus the maximum size of your record is 32 bytes.

You realise that if you are to use PTR# to locate a record, then each record must have the same size. Thus you decide that for Stamp Descriptions of less than 20 characters, you will pad out the description with trailing spaces to make it exactly 20 bytes long. Thus each and every record will be 32 bytes long. Since each disk sector holds 256 bytes, you can store 8 such records per sector.

You have an 80 track disk, so the number of sectors available = 80 * 10 less 2 for the catalogue. Thus you have 798 sectors available to you which allows you to store 798 * 8 records = 6384 records in the disk file. As you only have 4000 stamps, this is all right. If you had 8000 stamps, you might have had to consider creating separate files for UK and foreign stamps.

14.8.2 Applications

The next task is to define the application programs that you will have to write; the reason for using the computer in the first place.

Update Program. This will allow you to insert new records when you buy new stamps, to amend existing records as values change (and to correct mistakes), and to delete records should you dispose of some stamps. This program will also be used for the initial take-on of data.

In reality, there would be several other programs to report your collection in a variety of ways. For this example, assume that only the update program is required.

14.8.3 Special Problems

You now start to think about the way that you are going to code the programs. Are there any special features that need to be thought out before you put finger to keyboard? There nearly always are, naturally:

When the file is first created, the program will have to use OPENOUT. Subsequently, the program will have to use OPENUP. You will see that at line 90, the program attempts to open the file for update. It then checks the file handle returned by the DFS. If this is zero, DFS could not open the file, so the program calls a PROC called 'bodgeit'. This PROC starts at line 1240. It uses OPENOUT to create a catalogue entry, closes the file immediately, and finally re-opens it using OPENUP. This is not the only way, or even the most elegant way, of handling the problem, but it does work.

When you buy a new stamp, you will need to allocate the next sequential stamp number to it. But how will you discover what was the last stamp number used? Clearly, it will be the stamp number of the last record on the file. It would be a good idea to write your program such that, for new insertions, it automatically generates the next sequential number for you. In fact the program calculates the highest stamp number in the file at line 130 for just this purpose.

What is the algorithm that will establish the start position of each record in the file? Since each record is 32 bytes long, the first record will start at zero, the second record will start at 32, the third record will start at 64 and so on. Thus you can position the pointer at the start of any record by entering:

```
100 PTR# handle = 32 * (stampno-1)
```

This appears at line 760 in the program. In the program at lines 700 and 1020, a similar command positions the file at the Description field.

Because you are using sequential numbers, you cannot simply erase a stamp entry and pack the data. This would throw the sequence out of order. You decide to make the delete routine zeroise the value and space out the description. You make a mental note to write another program at a later date that will display these blank records, so that you can re-use the stamp numbers, by amending the existing blank records so that they hold data about your most recent acquisitions.

You realise that you can save yourself a bit of code for insertions, if you insert a blank record, apart from the Stamp Number, and then pass control to the amend routine. See how short PROCinsert becomes at line 1070.

You realise that strictly speaking, you do not need the Stamp Number field at all. However, you decide to keep it so that you can check that the record accessed is the one that you wanted. This could prove a useful diagnostic aid. PROCerror, at line 1170, reports on any mismatches.

You think about writing a full screen editor for the update program, but you decide to keep things simple.

14.8.4 The Easy Bit

When you think out your application like this, all that remains is the easy bit - writing the program. If your program was more complicated, you would probably spend some time breaking it into subroutines or procedures, before starting to code. In the program:

lines 180 to 260
are the main program loop

lines 300 to 560
are the menu. In this you specify whether you want to amend, insert, delete or exit. If you specify amend or delete, you are also asked which stamp number.

lines 590 to 720
handle amendments. PROCdisplay is called to display the current contents of the record. You are asked for a new description and a new value in turn. You simply hit RETURN if you do not wish to change either of these fields. Note lines 640 and 650 which ensure that the description field remains a fixed length of 20 bytes.

lines 750 to 900
display the current record.

lines 930 to 1040
delete the current record. The record is first displayed and then you are asked if you really want to delete it. If you do, the program changes the description to spaces and the value to zero.

lines 1070 to 1140
insert a new record by creating a dummy record and calling PROCamend.

lines 1170 to 1190
report bugs that are discovered if the stamp number in a record does not match what you expected.

lines 1240 to 1280
handle the OPEN problem if you are creating your file for the first time.

This may seem like a lot of code for such a simple application. Most of the code consists of printing menus and getting replies. The code associated with disk random access is actually very brief. Also, of the 128 lines in total, 28 of these are either REM statements or blank lines. The example chosen is intentionally simple. It is designed to illustrate the rudiments of disk file handling in BASIC. Later chapters expand on this theme considerably.

Stamp Update Program

```
10 REM  STAMPUP
20
30 REM This is the main update program.
40 REM
50
60 REM 1) Open the file "STAMPS" for read /write
70 MODE 7
80 *DRIVE 3
90 handle = OPENUP("STAMPS")
100 IF handle = 0 PROCbodgeit
110
120 REM 2) Calculate the highest stamp number on the file.
130 highstamp% = (EXT# handle) DIV 32
140
150 REM 3) If highest stamp is zero, the user can only
insert.
160 IF highstamp% = 0 PROCinsert
170
180 REM 4) Main Program Loop
190 REPEAT
200 PROCmenu
210 IF ans$ = "A" PROCamend
220 IF ans$ = "I" PROCinsert
230 IF ans$ = "D" PROCdelete
240 UNTIL ans$ = "E"
250 CLOSE# handle
260 END
270
280 REM 5) Display the menu. Get function wanted and where
290 REM   appropriate the stamp number wanted too.
300 DEFPROCmenu
310 CLS
320 PRINT TAB(8,2);CHR$(131);CHR$(141);"Stamp Update"
330 PRINT TAB(8,3);CHR$(131);CHR$(141);"Stamp Update"
340 PRINT TAB(8,6);CHR$(134);"A) = Amend"
350 PRINT TAB(8,7);CHR$(134);"I) = Insert"
360 PRINT TAB(8,8);CHR$(134);"D) = Delete"
370 PRINT TAB(8,9);CHR$(134);"E) = Exit"
380 REPEAT
390 PRINT TAB(8,11);STRING$(30," ")
400 PRINT TAB(8,11);CHR$(131);"enter choice ";
410 INPUT ">" ans$
```

```
420 IF ans$ = "a" ans$ = "A"
430 IF ans$ = "i" ans$ = "I"
440 IF ans$ = "d" ans$ = "D"
450 IF ans$ = "e" ans$ = "E"
460 ans$ = LEFT$(ans$,1)
470 UNTIL ans$ = "A" OR ans$ = "I" OR ans$ = "D" OR ans$ = "E"
480 IF ans$ = "I" ENDPROC
490 IF ans$ = "E" ENDPROC
500 REPEAT
510 PRINT TAB(0,13);STRING$(38," ")
520 PRINT TAB(0,13);CHR$(131);"enter stamp number"
530 PRINT TAB(0,14);CHR$(131);"in range 1 to ";highstamp%;" ";
540 INPUT ">" stampno%
550 UNTIL stampno% > 0 AND stampno% < highstamp%+1
560 ENDPROC
570
580 REM 6) Display the record and get the new values.
590 DEFPROCamend
600 PROCdisplay
610 PRINT TAB(0,11);CHR$(131);"enter new description or RETURN'"
620 INPUT ">" desc$
630 IF LEN(descS) = 0 desc$ = stamp$
640 IF LEN(desc$) > 20 desc$ = LEFT$(desc$,20)
650 IF LEN(desc$) < 20 desc$ = desc$ + STRING$(20-
LEN(desc$)," ")
660 PRINT TAB(0,15);CHR$(131);"enter new value or RETURN"
670 PRINT TAB(0,16);STRING$(38," ")
680 INPUT ">" val%
690 IF val% = 0 val% = stampval%
700 PTR# handle = 32 * (stampno% - 1) + 5
710 PRINT# handle,desc$,val%
720 ENDPROC
730
740 REM 7) Display the record
750 DEFPROCdisplay
760 PTR# handle = 32 * (stampno% - 1)
770 INPUT# handle,stampgot%,stamp$,stampval%
780 IF stampgot% <> stampno% PROCerror
790 CLS
800 PRINT TAB(8,2);CHR$(131);CHR$(141);"Stamp Update"
810 PRINT TAB(8,3);CHR$(131);CHR$(141);"Stamp Update"
820 PRINT TAB(8,6);CHR$(134);"Record Contains"
830 stamp1$ = STR$(stampno%)
840 stamp1$ = STRING$(4-LEN(stamp1$)," ")+stamp1$
850 stamp2$ = STR$(stampval%)
```

```
860 stamp2$ = STRING$(7-LEN(stamp2$)," ")+stamp2$
870 PRINT TAB(0,9);"<";stamp1$;">"
880 PRINT TAB(6,9);"<";stamp$;">"
890 PRINT TAB(28,9);"<";stamp2$;">"
900 ENDPROC
910
920 REM 8) Display the record and blank out the fields.
930 DEFPROCdelete
940 PROCdisplay
950 REPEAT
960 PRINT TAB(0,11);CHR$(131);"do you want to delete it Y/N"
970 PRINT TAB(0,12);STRING$(38," ")
980 INPUT ">" ans$
990 ans$ = LEFT$(ans$,1)
1000 UNTIL ans$ = "Y" OR ans$ = "y" OR ans$ = "N" OR ans$ =
"n"
1010 IF ans$ = "N" OR ans$ = "n" ENDPROC
1020 PTR# handle = 32 * (stampno% - 1) + 5
1030 PRINT# handle,STRING$(20," "),0
1040 ENDPROC
1050
1060 REM 9) Insert a new record'.
1070 DEFPROCinsert
1080 highstamp% = highstamp% + 1
1090 stampno% = highstamp%
1100 IF stampno% > 6384 PRINT "overflow" : STOP
1110 PTR# handle = EXT# handle
1120 PRINT# handle,stampno%,STRING$(20," "),0
1130 PROCamend
1140 ENDPROC
1150
1160 REM 10) It's a bug
1170 DEFPROCerror
1180 PRINT "expected ";stampno%;" got ";stampgot%
1190 STOP
1200 ENDPROC
1210
1220 REM 11) This file has never been opened before, so we
have to
1230 REM   create a blank file first.
1240 DEFPROCbodgeit
1250 handle = OPENOUT("STAMPS")
1260 CLOSE# handle
1270 handle = OPENUP("STAMPS")
1280 ENDPROC
```

14.9 Low Level DFS Calls

As previously stated, the Acorn DFS is richly endowed with hooks into its inner primitive routines. These routines are also available to the BASIC programmer. Previous chapters described all the disk related calls to OSFILE, OSARGS, OSBGET, OSBPUT, OSGBPB, OSFIND, OSFSC and OSWORD. Their entry points in the OS 1.20 ROM are:

```
OSFILE = &FFDD
OSARGS = &FFDA
OSBGET = &FFD7
OSBPUT = &FFD4
OSGBPB = &FFD1
OSFIND = &FFCE
OSFSC = indirected through &21E,&21F
OSWORD = &FFF1
```

There are two BASIC commands that allow us to invoke these routines, CALL and USR. For both of these commands, BASIC will set up the registers for you, prior to jumping into the routine. The register settings are as follows:

A register - set to least significant byte of A%

X register - set to least significant byte of X%

Y register - set to least significant byte of Y%

P register - the carry bit is set to the least significant bit of C%

Thus by pre-loading the integer variables A%,X%,Y% and C% you can invoke these routines as though you had made the correct register assignments, as detailed in the previous chapters.

When you use CALL, the final outcome with respect to the contents of the registers is not passed back to you. USR, however, does pass back to you the contents of these registers. Thus, whenever you need to know what the DFS routine has done to the registers, you use USR. When you do not need to know, you use CALL. There are lots of instances where the DFS replies to you via a control block or a parameter block, both of which are fixed areas in memory. In these instances you will usually be unconcerned about the contents of the registers and should use CALL.

Let's have a simple example of each.

14.9.1 CALL example

You can read the title of a disk, and the boot up option, by using OSGBPB. At line 30 we specify the entry point address for OSGBPB and save it in a variable called osgbpb.

You communicate with OSGBPB via two separate memory areas. Firstly you must set up a control block 12 bytes long. The DIM statement at line 40 does this (note that DIMs are addressed from zero, so you actually specify one byte less). You must also allocate a memory area for the reply. The maximum reply will be 14 bytes long; hence the DIM statement at line 50. The control block (bytes 1 to 4) must contain the address of the area into which OSGBPB puts its reply, and line 60 achieves this: On entry to OSGBPB, the A register must equal 5 (specifies that you want the disk title), and the X and Y registers must contain the address of the control block. Lines 70 to 90 do this. At line 100 OSGBPB is invoked. When OSGBPB has finished, the reply area contains:

byte 0 - length of title = n (say)
byte 1 to n = disk title byte n + 1 =
boot-up option

The title of the disk is 'DFS Guide'. To show that this call really works, lines 110 to 140 copy the title passed by OSGBPB, byte by byte, into the string variable A$. Line 150 prints the title.

```
10 REM CALL example
20 MODE 7
30 osgbpb = &FFD1
40 DIM conblok 11
50 DIM reply 13
60 conblok!1 = reply
70 A% = 5
80 X% = conblok MOD 256
90 Y% = conblok DIV 256
```

```
100 CALL osgbpb
110 A$=""
120 FOR I%  = 1 TO ?reply
130    A$ = A$ + CHR$(reply?I%)
140 NEXT
150 PRINT A$
160 END
>RUN
DFS Guide
```

14.9.2 USR example

USR is called with statements such as:

```
100 osgbpb = &FFD1
110 R = USR(osgbpb)
```

This does exactly the same as CALL (registers are loaded from the integer variables etc.), except that on return, the variable R contains a floating point number. If you print that number in hexadecimal, using PRINT ~R, you might find a value such as &00123456.

This is really the contents of four different registers:

&00 = contents of the P register
&12 = contents of the Y register
&34 = contents of the X register
&56 = contents of the A register

You can put the contents of the registers back into integer variables as follows:

```
A% =   (R AND &FF)
X% =   (R AND &FF00)          DIV &100
Y% =   (R AND &FF0000)        DIV &10000
C% =   (R AND &FF000000)      DIV &1000000
```

Notice that C% now contains the entire contents of the status register, not just the carry bit. This consists of:

bit 7 = Negative Flag
bit 6 = Overflow Flag
bit 5 = not used

bit 4 = BRK Flag
bit 3 = Decimal Flag
bit 2 = Interrupt Disable Flag
bit 1 = Zero Flag
bit 0 = Carry Flag

You can use the function below to test for any of these bits. For example, if you have put the status register in S% and you want to know if the carry bit (bit 0) is set, you would enter:

```
100 IF FNbit(S%,0) = 1

900 DEF FNbit(byte%,bit%)
910 byte% = byte% AND (2 ^ bit%)
920 IF byte% = 0 THEN = 0 ELSE = 1
```

And so onto an example of USR. This time we shall use OSARGS to discover which filing system is currently open. To do this we enter OSARGS with the A register and Y register set to zero. On return, the A register contains a code (see the Chapter on OSARGS) which should be &04, indicating that the Disk Filing System is currently running.

```
10 osargs = &FFDA
20 A% = 0
30 Y% = 0
40 R = USR(osargs)
50 A% = (R AND &FF)
60 PRINT A%
70 END
>RUN
4
```

14.9.3 Using OSFSC from BASIC

You will probably have noticed that OSFSC is a special case as it does not have an entry point. Entry is via the FSCV vector (&21E,&21F) which simply contains the address that you want to CALL, least significant byte first.

Therefore, to obtain the entry point of OSFSC, you would enter something like this:

```
100 osfsc = ?&21E + 256*?&21F
110 CALL osfsc

Or

110 R = USR(osfsc)
```

15 Acorn DFS Error Messages

The DFS contains a substantial number of error messages to guide you when you go wrong. Naturally, each error message occupies valuable ROM space and is often terse because of this. The error messages given in this Chapter apply to DFS 0.90. Frequently, DFS 1.20 has more concise error messages (to the point of being unhelpful in many cases). Where the DFS 1.20 message differs from that in DFS 0.90, this is given as well.

An error number is associated with each error message, and this is accessible to BASIC via ERR.

This chapter deals with each error message in turn, explaining what each means. Many of the error messages can have multiple causes. For these, the most likely causes are itemised. At the end of this chapter there is some general guidance on practices and techniques which will prevent disk errors occurring.

The error messages are explained in alphabetical sequence, except for the 'Disk fault' and 'Drive fault' messages. These are special, since they are reported with an additional error number which reflects the contents of the 8271 FDC Result Register. They will be dealt with first.

15.1 'Disk fault' and 'Drive fault'

To all intents and purposes you may disregard the legends 'Disk fault' and 'Drive fault'. It is the error number which accompanies the message that identifies the error, rather than the message itself. DFS 0.90 also tells you the track number and the sector number at which the error occurred, but this desirable feature is omitted from DFS 1.20.

Both of these messages mean that the 8271 FDC returned an error status in the Result Register. The entire Result Register is printed out as a hexadecimal error code by these two messages, providing you are working in BASIC.

If you handle the 8271 yourself via OSWORD, you will not receive these error messages at all. However, the contents of the Result Register will be returned to you and you can examine these bits for yourself.

Some of the OSWORD commands do not generate error conditions. The 'Read Special Register' command is one example. The chapter on OSWORD explains this fully.

For those commands that return an error number, the Result Register contains zeroes in bit 7, bit 6 and bit 0. Normally bit 5 will also be zero, unless you are handling Deleted Data Marks in which case it may be set to 1. The error bits themselves are contained in bits 4 to 1 inclusively.

The list below is not exhaustive for the 8271. There are special error codes for the DMA mode of operation and the associated Scan commands, none of which is applicable to the BBC Microcomputer. These are the error codes that you may receive:

&00 - All is well.

This indicates that the 8271 command was processed successfully. No 'Disk fault' or 'Drive fault' messages are generated.

&08 - Clock Error.

This indicates that the 8271 found a clock bit to be missing when reading data from the disk. The likely causes are as follows:

> You are using the wrong type of diskette for your drive (e.g. 40 track diskette formatted as 80 track).
>
> Your diskette is permanently damaged in one or more places. Try overwriting the offending sector. If you get the same fault when you read the sector back, throw the diskette away.
>
> The diskette has been exposed to strong magnetic fields. Overwriting the offending sector should cure the problem, but treat the diskette as suspect for a while.

A disk interface chip is faulty, or the 8271 itself is faulty. If this is so, you will suffer repeated errors on all your diskettes and should get your microcomputer repaired.

Alas, the error means that at least one sector on your diskette is unreadable. If you have multiple files on this diskette, you can try to *COPY them one by one to a different diskette. This may allow you to salvage something from the mess.

&0C - Sector Id CRC Error.

When your disk was formatted, the 8271 calculated a two-byte CRC for each Sector Id and wrote these onto the diskette. Each time the 8271 reads a sector it re-calculates the CRC. This error simply says that, for a particular Sector Id, the result of this re-calculation does not match the original CRC written on the diskette. The most likely causes are:

You are using the wrong type of diskette for your drive (e.g. 40 track diskette formatted as 80 track).

Your diskette is permanently damaged in one or more places. Try re-formatting the diskette. If you cannot verify the diskette successfully, throw the diskette away.

The diskette has been exposed to strong magnetic fields. Re-formatting the offending sector should cure the problem, but treat the diskette as suspect for a while.

You have a hardware fault, in which case you will suffer repeated errors on all diskettes.

Alas, the error means that at least one sector on your diskette is unreadable. If you have multiple files on this diskette, you can try to *COPY them one by one to a different diskette. This may allow you to salvage something from the mess.

&0E - Data CRC Error.

When you write data to the diskette, the 8271 calculates a two-byte CRC for the entire sector of data and writes these onto the diskette. Each time the 8271 reads a sector of data it re-calculates the CRC. This error simply says that, for a particular sector of data, the result of this re-calculation does not match the original CRC written on the diskette. The most likely causes are:

> You are using the wrong type of diskette for your drive (e.g. 40 track diskette formatted as 80 track).
>
> Your diskette is permanently damaged in one or more places. Try overwriting the offending sector. If the sector still cannot be read, throw the diskette away.
>
> The diskette has been exposed to strong magnetic fields. Overwriting the offending sector should cure the problem, but treat the diskette as suspect for a while.
>
> You have a hardware fault, in which case you will suffer repeated errors on all diskettes.

Alas, the error means that at least one sector on your diskette is unreadable. If you have multiple files on this diskette, you can try to *COPY them one by one to a different diskette. This may allow you to salvage something from the mess.

&10 - Drive Not Ready.

The selected drive is not ready. The most likely causes are:

> Disk drive not powered up.
>
> No diskette in the selected drive.
>
> Disk drive not yet up to speed.
>
> Disk drive selected does not exist.
>
> Hardware fault.

The first two have obvious remedies.

The next two cannot happen if you use OSWORD; only if you command the 8271 directly. For those that want to do this, the only way of clearing the not ready status is to use the 'Read Drive Status' command. You should incorporate this into your code.

If you are left with fifth possibility try a hard reset and see if the problem goes away. If not, check that the disk cable is connected properly at both ends. If it is, it is most likely to be the disk drive that is faulty and you will have to get it repaired.

&12 - Write Protect.

You have attempted to write on a write-protected disk. BASIC programmers will never see this 'Disk fault' message, because the DFS traps this status and outputs the more informative message, 'Disk Read Only'. The likely causes for the message are:

> You have put the wrong diskette in the drive.
>
> You have forgotten to remove the sticky paper which covers the write protect notch.
>
> Hardware fault.

The first two have obvious remedies.

If you are left with the third possibility try a hard reset and see if the problem goes away. If not, and especially if all diskettes are reported as write-protected it is most likely to be the disk drive that is faulty and you will have to get it repaired. If the drive functions correctly in all other respects, it is likely to be either the LED or the photo-detector which need replacement.

&14 - Track 0 Not Found.

The 8271 has tried to 'Seek' track 0, either because you told it to do this, or because the DFS did so on your behalf. The 8271 is prepared to step outwards 255 times until the track zero microswitch trips. You have a hardware fault of some description. The most likely cause of this error is that the microswitch in the disk drive is faulty. You should get it repaired.

&16 - Write Fault.

Your disk drive has detected a fault whilst attempting to write data onto the diskette. Unless you are writing some really hairy code, you should assume that you have a hardware fault of some description. How sure are you of the cable? If you are sure the cable is sound, you should take the disk drive for repair.

&18 - Sector Not Found.

Of all the possible error messages, this must be the commonest. The error message means that your program has found a track in which a particular Sector Id cannot be found. The most likely causes are:

> You are trying to read/write a 40 track diskette in an 80 track drive (or vice-versa).
>
> You are attempting to copy (or pry into) a piece of commercial software that uses logical sectoring techniques to make it difficult for you.
>
> You are attempting to read/write an unformatted diskette.
>
> The timing parameters used by the DFS to initialise the 8271 are incorrect for your drive (see Chapter 2).

15.2 'Bad address'

error number: 252 (&FC)

You have specified an address badly, in a command that expects an address parameter. The commands concerned are *LOAD and *SAVE and you can check their respective formats elsewhere in this book. All addresses supplied by you should be hexadecimal, but you should not prefix them with the ampersand sign.

15.3 'Bad attribute[7]

error number : 207 (&CF)

You have used the * ACCESS command incorrectly. Valid parameters are 'L' to lock a file, or none at all to unlock a file. You have entered something else.

15.4 'Bad command'

error number : 254 (&FE)

You have entered a star command. The OS ROM did not recognise it, so it tried each of the paged ROMs in turn (providing they had a service entry). None of these recognised it, so the DFS was asked to look for a utility program of that name on disk. If you did not include the drive number and directory in your star command, the DFS would have selected the default machine-code library drive and directory, as specified by *LIB. The DFS was unable to find the program that you requested.

Most probably you have mis-typed the command. You may, however, simply have the wrong settings in *LIB, so that the DFS is looking at the wrong diskette, or you may have put the wrong diskette in the drive.

15.5 'Bad directory'

'Bad dir'in DFS 1.20
error number : 206 (&CE)

You have specified a bad directory parameter within a *DIR or *LIB command. Any single character other than ':', '.', '*', '#' may be used. You have probably entered more than one character,

15.6 'Bad drive'

error number : 205 (&CD)

You have specified a bad drive parameter. The drive parameter must be a number in the range 0 to 3. Note that commands which expect a drive parameter (*BACKUP, *CAT, *COMPACT, *COPY and *DRIVE) expect just this number. You may also specify a drive number as part of the filename in commands that expect a filename parameter. In these cases the drive number should be prefixed by a colon. Check the syntax of the command in Chapter 5.

15.7 'Bad filename'

'Bad name' in DFS 1.20

error number : 204 (&CC)

You have specified a bad filename in a command that expects a filename parameter. Typical filenames are:

```
:1.A.PROGl

:l.PROGl

A .PROG 1

PROG1
```

Note the use of the colon to de-limit a drive number, and the fullstop to de-limit a directory character. Be sure that you do not use wild cards(* and #) in a command that only accepts a single file specification.

15.8 'Bad option'

error number : 203 (&CB)

You have specified a bad option parameter in a *OPT command. Two parameters have to be entered, separated either by space or comma. The allowable values are:

```
*OPT1,0
*OPTl,l
*OI'T4,0
*OPT4,1
*OPT4,2
*OPT4,3
```

Several variations in style are also acceptable, (e.g. instead of *OPT1,0):

```
*OPTl 0
*OPT 1,0
*OPT 1 0
```

15.9 'Can't extend'

error number : 191 (&BF)

You are probably trying to *BUILD, *COPY, SAVE, *SAVE or *SPOOL a file, but a file of that name already exists on the diskette. So the DFS attempts to write it back in the same part of the diskette. Unfortunately, you have made the file larger than the original and it cannot squeeze it into its existing slot on the diskette. You can solve the immediate problem by saving the file with a brand new name, deleting the original file, and using *RENAME to change the filename back to the original name. You may well want to *COMPACT the diskette also to free up the space originally occupied by the file. To avoid this type of problem altogether, read the section later in this chapter on practices and techniques.

You can also get this problem with random access files, if you originally create a small file, but later try to insert new records. It is quite likely that your file now ends with a partially complete record. Problem avoidance is much easier than cure - see later.

15.10 'Catalogue full'

'Cat full' in DFS 1.20
error number : 190 (BE)

Because the catalogue size is fixed, you may only save up to 31 files per side of a diskette. This message tells you that you have tried to exceed that number. Use a different diskette to save the file.

15.11 'Channel'

error number : 222 (&DD)

You have specified a bad file handle to BGET#, BPUT#, CLOSE#, EOF#, EXT#, INPUT#, PRINT#, PTR# or to a low-level DFS routine such as OSARGS. If you are writing in BASIC, be sure to:

a) Open the file using the correct command.

b) Check for a zero file handle and abort if zero.

c) Use the file handle returned by the open routine in all other file handling commands.

d) Do not close the file until you have finished with it.

If you are writing in machine-code, be sure to:

a) Open the file using OSFIND.

b) Check for a zero handle and abort if zero.

c) Use the handle correctly with other file handling commands.

d) Do not use OSFIND to close the file until you have finished with it.

15.12 'Disk changed'

error number : 200 (&C8)

This error is most likely to occur if you have more than one drive. Suppose that you have a program, PROGA, that reads a file in drive 1 and that this program produces some interesting results which you would like to include in a memorandum using a word processor. On a bad day, you might enter:

```
*drive 0
LOAD "PROGA"
```

```
*SPOOL MEMO
RUN
*SPOOL
```

If your program contains a *DRIVE 1 command, you can thoroughly confuse the Acorn DFS and receive this error message.

There are sillier, less likely ways of getting this message, such as removing a diskette that you are currently reading or writing.

15.13 'Disk fault'

error number : 199 (&C7)

See 15.1.

15-14 'Disk full'

error number : 198 (&C6)

This happens when you are trying to save a brand new file on the diskette, but the file size exceeds the space available. Use a different diskette to save the file and then come back to the problem. Firstly, it may well be that you have exhausted the space on the diskette and can add no more data. However, it is also possible that you have deleted a lot of small files from the diskette and not used *COMPACT. In a compacted diskette, every free sector is available for use. It is also possible that you tried to *BUILD or OPENOUT a small file, forgetting that the DFS reserves 64 sectors for that file as a default. Lastly, you may have an unintentional program loop which is writing vast amounts of unwanted data to a file opened for output. Try entering *INFO. This will tell you about the files on the diskette and how big they are. You can *DELETE any large, unexpected files.

15.15 'Disk read only'

error number : 201 (&C9)

This happens when you attempt to write on a write-protected diskette (sticky paper over the write protect notch). This error is really an 8271 error, specially trapped by BASIC to give a more readable error message. See 15.1 - error &12.

15.16 'Drive fault'

error number : 197 (&C5)

See 15.1.

15.17 'EOF

error number : 223 (&DE)

You have tried to read (BGET# or INPUT#) past the end of a file. You should include some code in your program to test for end of file using EOF#. In all probability, you have a logic error in your program.

15.18 'File exists'

'Exists' in DFS 1.20
error number : 196 (&C4)

This error occurs when you use *RENAME badly. If you try to rename a file, but a file with your chosen new name already exists on the diskette, you get this message. Choose a different name or delete the offending file first.

15.19 'File locked'

'Locked' in DFS 1.20
error number : 195 (&C3)

You have tried to write on or delete a locked file. Use *ACCESS to unlock it first.

15.20 File not found'

'Not found' in DFS 1.20 error
number : 214 (&D6)

The DFS cannot find the file that you have specified. It may be just bad typing, or you may have inserted the wrong diskette into the drive. Alternatively, you could have *DIR or *LIB set wrongly. You may even have specified wild cards in a command that only accepts a single file specification. This is a common message and mostly the cause is immediately apparent. If you manage to get this message but you do not know why, remember that something you have done asked for a specific file. Think along those lines and it will come to you.

15-21 'File open'

'Open' in DFS 1.20
error number : 194 (&C2)

You have tried to open the same file twice, without closing it in between. This is normally a bad piece of logic in a program. You can also get this message if you fail to close the file in a program and subsequently reference it in a star command. Remember that CLOSE# completes the catalogue entry for a file. Until CLOSE#, there is no record of how big the file is and so it cannot be referenced in a star command. You do not need to know the handle to close the file. CLOSE#0 will close all open files, allowing you to proceed with your chosen star command.

15.22 'File read only'

'Read only' in DFS 1.20
error number : 193 (&C1)

You have BASIC level II. You have opened a file for input only, using OPENIN. You have subsequently tried to write to the file using BPUT# or PRINT#. Either the BPUT# / PRINT# statements are using the wrong handle, or you are using the wrong open command.

15.23 'Not enabled'

(not appropriate to DFS 1.20)
error number : 189 (&BD)

You have a 1)I'S ROM with a level less than about 1.0. You have entered *BACKUP or *DESTROY without previously entering *ENABLE. See the chapter on star commands for a more detailed explanation and the chapter on OSFSC if you want the real nitty-gritty.

15.24 'Syntax'

error number : 220 (&DB)

You have completely misused a DFS command. You should check the chapter on star commands to see which parameters you should supply, together with any rules for them. If you prefer you may enter *HELP DFS or *HELP UTIL to get a cryptic description of the command.

15.25 Too many open files'

'Too many open' in DFS 1.20
error number : 192 (&C0)

You are only allowed up to five simultaneously open files. Note that *BUILD, CHAIN, *EXEC, LOAD, *LOAD, SAVE, *SAVE and *SPOOL also open a file, however briefly. This restriction is caused because the buffer areas set aside for each file occupy fixed memory locations from &1200 to &16FF. Each file has a one-page (256 bytes) buffer. The error message tells you that you have attempted to open a sixth file.

If your application must have more than five files, you will have to consider opening and closing some of them on each occasion that vou wish to access them.

15.26 Hardware Problems

Many of the error codes, especially those that reflect the contents of the 8271 Result Register, indicate the possibility of hardware failure. In places you are advised to repair the drive, or replace the 8271 etc. These suggestions are offered as the most likely cause, not the definitive cause. In fact the hardware failure may easily be elsewhere and you can save yourself much time and expense by further diagnosis. All you need is a willing friend with a fully working BBC Microcomputer and disk interface.

You should take along both your drive and your disk cable. The following steps should be undertaken:

> Connect your drive to your friend's microcomputer, using your friend's cable. If it works, you can give your drive a clean bill of health. If it does not work, your drive is faulty.
>
> Connect your drive to your friend's microcomputer, using your own cable. Be sure to insert the cable the correct way round. (Examine your friend's cable carefully before you disconnect). If it works, your cable is also all right. If it fails, repair or replace the cable.
>
> If both a) and b) work, then you need to take your microcomputer for repair.

If you find yourself in a repair shop, tell the engineer about the tests that you have done. The bane of all repair shops is the garbled account of the nature of the fault. So often misleading and erroneous information is given, that most engineers prefer to perform their own sequence of tests, taking nothing for granted. If, however/you impress the engineer with the methodical way that you have gone about diagnosing the problem, he may well be prepared to trust you and even repair while you wait, especially if a simple chip replacement is indicated.

15.27 Practices and Techniques

Prevention is better than cure. This should be your maxim when working with disks. The few simple ideas below can save you hours of frustration. They are designed either to avoid disk errors in the first place, or to soften the blow when they occur.

When writing a new program:

Verify three diskettes before you start.

Cycle these diskettes in turn when saving your program. This is called the 'Grandfather, Father, Son' system and gives good backup, should you lose a diskette.

Store no other files on these diskettes. This will avoid all the 'Can't Extend' messages.

Create test data for your program (if necessary) on a separate diskette. Never try out an untested program on a file that you can ill afford to lose.

When you have finished testing your program and it is fully operational, you can copy it onto a diskette containing several other files.

You should make at least one further copy on another diskette.

You should lock the file against accidental erasure.

Your three working diskettes are now available for re-use when you decide to write another program.

When creating a data file:

If the data file is to be expanded at a later date, it should be the only file on the diskette. This avoids 'Can't Extend' problems.

Regular backup copies should be taken using a different diskette.

PART 2
DISK UTILITIES

16 Disk Overlay

Have you ever wished that you had more memory at your disposal? Sooner or later, most home computer users discover just how limited the memory capacity really is. This is particularly true for the BBC Microcomputer in the graphics modes. You cannot write much code before you fall foul of the dreaded 'No room' message.

All is not gloom and despondency, however. Your disk system can come to your rescue. The idea behind disk overlay is that you keep parts of your program on disk, loading them into memory only when you need them, and overwriting other parts of your program that you no longer need for a while. This technique allows you to write huge programs constrained only by the capacity of your disk system. As ever in life, luxury such as this carries a heavy premium. Disk input/output is not instantaneous. Your program must wait for disk accesses and some of the spontaneity may well be lost. You must also structure your program most carefully to use disk overlay techniques.

In spite of these drawbacks, disk overlay is applicable in a wide variety of circumstances and is well worth a chapter of its own.

16.1 Screen Overlay

One of the simplest and most effective ways of using disk overlay is for the various screen displays that a program may need. You can write a game with literally dozens of different screens all saved on disk, each ready to be loaded at the appropriate point in the program. If the screens are MODE 2 screens, each one that you keep on disk saves you 20K of memory, or at least a lot of PLOT statements etc. needed to create the screen. A MODE 7 screen occupies only IK of memory, but if you keep it on disk, think of all those PRINT statements that you are removing from your program!

16.1.1 Screen Creation

The first stage, then, is to create each screen that your program will need and to save each one on disk. For a game, you might decide to use one of the many graphics packages on the market. Alternatively, you can write a program that creates a screen by using the PLOT commands or by poking memory.

For this example, a MODE 7 screen is created and saved to disk. The screen contains a typical menu that might be used in a program. Figure 30 is similar to the required screen display (omitting colour and double height letters).

```
main menu

1) Delete a record
2  Insert a new record
3) Update a record
4) Display a record
5) Create an invoice
6) Run stock report
7) Quit

enter choice
```

Figure 30. The Required Screen.

The program that creates this screen is listed below. Note that at line 160, the menu is saved to disk. In practice, you would insert this line of code only after testing the program to ensure that it produces an acceptable display. The menu consists of:

a header (lines 80 to 100)

all the lines in the DATA statements at the end

a solicit (line 150)

```
10 REM
20 REM SCR1 - create MENU
30 REM
40
50 MODE 7
60 REM cursor off
70 VDU 23;11,0;0;0;0;
80 A$ = "main menu"
90 PRINT TAB(10,2);CHR$131;CHR$141;A$
100 PRINT TAB(10,3);CHR$131;CHR$141;A$
110 FOR I% = 1 TO 7
120    READ A$
130    PRINT TAB(4,I%+5);CHR$134;A$
140 NEXT
150 PRINT TAB(4/23);CHR$131;"enter choice "
160 *SAVE MENU 7C00 8000 7C00
170 END
180
190 DATA "1) Delete a record"
200 DATA "2) Insert a new record"
210 DATA "3) Update a record"
220 DATA "4) Display a record"
230 DATA "5) Create an invoice"
240 DATA "6) Run stock report"
250 DATA "7) Quit"
```

It is worth mentioning at this point that you should also consider *BUILD as a tool for creating menus. Such menus can be displayed using *TYPE.

16.1.2 Screen Loading

Let us now see how we can use this screen within a program. The short program below loads the screen created by the previous program (at line 30). It then gets the user's choice from keyboard entry (line 70). This process is contained in a REPEAT loop (lines 40 to 80) and is repeated until such time as the user enters a number between 1 and 7. It is important to realise just how much memory we have saved. Had we not loaded the screen from disk, our program below would have had to contain nearly all the statements that formed the screen creation program. Moreover, since the screen occupies only

IK of memory, the disk load is practically instantaneous. In practice, a program would probably have several different screens, each of which could be handled in this way.

```
10 MODE7
20 VDU23;11,0;0;0;0;0;
30 *L0AD MENU 7C00
40 REPEAT
50  PRINT TAB(17,23);STRING$(12," ")
60  PRINT TAB(17,23);CHR$131;">";
70  INPUT " " choice%
80  UNTIL choice% > 0 AND choice% < 8
90 END
```

16.1.3 The Professional Touch

In the example, none of the spontaneity of the program is lost because a MODE 7 screen occupies only four sectors of disk and this loads fast. However, it is a different story with a MODE 2 screen which occupies 80 sectors of disk and takes about 3 seconds to load. Worst of all, you can see the picture build up on the screen as it loads. The result is that the overall effect is amateurish. What subterfuge can you employ to disguise the inner workings of your program? There are many alternative strategies. Finding one to suit your game is all part of the fun. If your game is a space game, perhaps you can fiddle with the video control registers to achieve some interesting effects. You can kid the user that this is meant to simulate warp drive, while all the time you are covering up for a screen load. If you are bereft of ideas, you can use the scheme below.

You can totally disable the screen, before the *LOAD, by entering:

```
VDU 23,0,8,48,0;0;0;
```

This will prevent the user from seeing the picture build up on the screen. When the load is completed, you simply enter:

```
MODE 2
```

This will cause the display to appear almost instantaneously. Even better is to enter:

```
*FX19
MODE 2
```

This will cause the program to wait for a vertical sync pulse and remove all trace of flicker from the display.

16.2 BASIC Program Overlay

Disk overlay is not confined to screens. You can break up your program into chunks called 'modules', and save some of these onto disk. Many operating systems have in-built facilities for doing this. The technique is then known as 'virtual memory' since you are using disk space as though it was extra RAM. The BBC Microcomputer has no in-built virtual memory software. It is not difficult to achieve disk overlay, however, but you must exercise a certain amount of care and obey certain rules.

The intention is:

to write a main module which is resident in RAM all the time. This will contain all the frequently referenced code.

to split the infrequent code into a number of independent, disk-based modules, only one of which will be resident in RAM at any one time.

The rules are:

each disk-based module is loaded at the same RAM address as every other disk-based module.

disk-based modules may be of different lengths, and the maximum size of code will be the length of the main module plus the length of the largest disk-based module.

only one disk-based module can be in RAM at one time, overlaying its predecessor.

a disk-based module may call a PROC in the main module or in itself, but not in another disk-based module.

a disk-based module will consist entirely of PROCs and FNs, so that line numbers do not matter.

no part of the program will use RESTORE, GOTO or GOSUB, again ensuring that line numbers have no real significance in the program.

no PROC or FN name will be used more than once in the whole program

the main module will be responsible for recognising when a different disk-based module needs to be loaded and for achieving that load prior to invoking a PROC or FN within it.

The first step is to map out the structure of the program: which code goes in which module. For a games program you might decide to have 15 screen modules, 15 disk-based modules (one to process each screen) and a main module to control the whole thing and to provide a common set of PROCs that may be needed in more than one disk-based module.

Most people would then create the screens and code the disk-based modules. It is worth testing each disk-based module in isolation. This usually means adding some code at the front to drive each PROC/FN in turn. This test code can later be removed. Finally, when all the disk-based code is tested, you would write the main module which pulls everything together.

Let's have a simple example. We will write a program that consists of a main module called 'MAIN' and three disk-based modules called 'MODULEA', 'MODULEB' and 'MODULEC'.

16.2.1 Disk-Based Modules

The disk-based modules in this example do not achieve very much. They simply contain one PROC each which prints out an identifying message. Note that each disk-based module starts at line 9000. This is not really necessary if you follow the rules, but it does improve the clarity of the overall listing when debugging.

```
9000  REM
9010   REM MODULEA
9020   REM
9030   DEF PROCprinta
9040   PRINT TAB(10);"This is MODULEA"
9050   ENDPROC

9000   REM
9010   REM MODULEB
9020   REM
9030   DEF PROCprintb
9040   PRINT TAB(10);"This is MODULEB"
9050   ENDPROC

9000  REM
9010  REM MODULEC
9020  REM
9030  DEF PROCprintc
9040  PRINT TAB(10);"This is MODULEC"
9050  ENDPROC
```

Each of these modules is saved on disk. You should enter *INFO and write down the size (in hex bytes) of the largest module. In this case, all the modules are the same size, say &100 bytes.

16.2.2 Main Module

First it is necessary to review the way in which a BASIC program is normally stored in memory.

```
 ---------------   TOP OF RAM


   screen memory

 ---------------   HIMEM

   BASIC stack

 ---------------   VARTOP

   variables

 ---------------   L0MEM

   spare

 ---------------   TOP

   BASIC program

 ---------------   PAGE
```

Figure 31. The BASIC program

A normal BASIC program starts at PAGE. The tokenised code continues up to TOP. The variables required by the program are stored from LOMEM up to VARTOP. If you do nothing to prevent it, LOMEM will have the same value as TOP, leaving no room for any of the disk-based modules. Figure 32 shows what we are trying to achieve.

```
                  TOP OF RAM

    screen memory

----------------  HIMEM

    BASIC stack

----------------  VARTOP
    variables for
    all modules

 --------------   LOMEM
    MODULEA or
    MODULEB or
    MODULEC
 ---------------  ORIGINAL TOP

    MAIN

 --------------   PAGE
```

Figure 32. The Overlay.

Thus at line 60 in MAIN, we increase LOMEM by the size of the largest disk-based module (in our case by &100) to allow room for it. It is important to do this before referencing any variables (other than A% to Z%). The main module calls PROCs printa, printb and printc. These are in MODULEA, MODULEB and MODULEC respectively. Before calling each of these PROCs, the main module *LOADS the appropriate disk-based module into RAM. It uses PROCoverlay to do this, which has a single argument, the name of the module to be loaded. There are several interesting points to note:

> PROCoverlay uses *LOAD not LOAD. LOAD would simply overwrite the main module with the disk-based module. We want both to co-exist.

PROCoverlay uses OSCLI to achieve the *LOAD. OSCLI is only available in BASIC II, but BASIC I users can achieve the same end with the following corrections:

```
61 DIM oscli 60
160 oscli = "LOAD " + M$ + STR$~(T%-2)
161 X% = oscli MOD 256
162 Y% = oscli DIV 256
163 CALL &FFF7
```

line 50 stores the original value of TOP in T%

PROCoverlay loads each disk-based module at T%-2 (ie. TOP-2). This effectively overwrites the &0DFF which terminates the main program, so that the overlay module continues quite naturally as an extension to the main module.

```
10 REM
20 REM MAIN
30 RE
40 MODE7
50 T% = TOP
60 LOMEM = TOP + &100
70 PROCoverlay("MODULEA")
80 PROCprinta
90 PROCoverlay("MODULEB")
100 PROCprintb
110 PROCoverlay("MODULEC")
120 PROCprintc
130 END
140 DEF PROCoverlay(M$)
150 M$ = M$ + " "
160 OSCLI "LOAD " + M$ + STR$~(T%-2)
170 ENDPROC
```

Figure 33 shows the result of running this program.

```
This is MODULEA
This is MODULEB
This is MODULEC
```

Figure 33. Disk Overlay Example.

If you enter LIST after running this program you will see the following. Note how the last disk-based module loaded, MODULEC, appears nicely at the end as though really we had only written a straightforward program.

```
10 REM
20 REM MAIN
30 REM
40 MODE7
50 T% = TOP
60 L0MEM = TOP + &100
70 PROCoverlay("MODULEA")
80 PROCprinta
90 PROCoverlay("MODULEB")
100 PROCprintb
110 PROCoverlay("MODULEC")
120 PROCprintc
130 END
140 DEF PROCoverlay(M$)
150 M$ = M$ + " "
160 OSCLI "LOAD " + M$ + STR$~(T%-2)
170 ENDPROC
9000 REM
9010 REM MODULEC
9020 REM
9030 DEF PROCprintc
9040 PRINT TAB(10);"This is MODULEC"
9050 ENDPROC
```

16.2.3 Other Considerations

That was a really simple example, just to get the main points across. In practice you need to be a bit more sophisticated in your handling of disk overlays. For example, you do not want to *LOAD MODULEA each and every time you call a PROC in MODULEA, because MODULEA may well be the one that is currently loaded. You should keep arrays in the main module denoting all the PROCs and their whereabouts. You should also keep an indicator which tells you the currently loaded disk-based module. The following version of MAIN is perhaps more typical of what you might choose to write for vourself.

```
10     REM
20     REM MAIN
30     REM
40     MODE7
50     T% = TOP
60     LOMEM = TOP + &100
70     DIM proc$(3), mod$(3)
80     currmodS = "       "
90     FOR I% = 1 TO 3
100     READ proc$(I%)
110     READ mod$(I%)
120    NEXT
130    PROCcall("printa")
HO     PROCcall("printb")
150    PROCcall("printc")
160    END
170
180    DEF PROCcall(P$)
190    I% = 0
200    REPEAT
210    I% = I% + 1
220    UNTIL I% = 3 OR proc$(I%) = P$
230    IF proc$(I%) <> P$ PRINT "bad proc" : STOP
240    IF mod$(I%) <> currmod$ PROCoverlay(mod$(I%))
250    IF P$ = "printa" PROCprinta : ENDPROC
260    IF P$ = "printb" PROCprintb : ENDPROC
270    IF P$= "printc" PROCprintc
280    ENDPROC
290
```

```
300 DEF PROCoverlay(M$)
310 currmod$ = MS
320 M$ = M$ + " M
330 OSCLI "LOAD ! + M$ + STR$~(T%-2)
340 ENDPROC
350
360 DATA "printa","MODULEA"
370 DATA "printb","MODULEB"
380 DATA "printc","MODULEC"
```

16.3 Disk Overlay and Machine-Code

Disk overlay techniques are not restricted to BASIC programs. Some or all of the disk-based programs may be in machine-code. For that matter, the main module itself may also be in machine-code. In some respects machine-code overlays are much simpler. There is no need to mess about with LOMEM, for example. If your main module is in BASIC, however, you do need to ensure that the chosen load address of the machine-code modules is completely outside of the domain of your BASIC program. You can establish a safe place by entering:

```
PRINT ~(?&2 + 256*783)
```

This will give you the hex value of VARTOP. You should be safe above this and up to about HIMEM-&100.

In other respects, machine-code overlays are much tougher. Machine-code subroutines are called at an absolute address and the code is seldom relocatable. You may plan to load all your machine-code modules at &4000, for example. When you have finished the main module, you may discover that it was bigger than you anticipated and your disk-based modules will have to load at &4100 instead. The following tips may simplify the problems for you. They assume a main module in BASIC II and disk-based modules in machine-code, but similar principles equally apply if the main module is also in machine-code.

16.3.1 Disk-Based Modules.

You should keep the original BASIC version of your assembler modules, so that you can re-assemble them later at a different start address. If you have BASIC 11, this is really straightforward. You can use the extra values of OPT (add 4 to the normal values), in conjunction with O% and P% to relocate the program. The program will be actually assembled at O%, but the assembly will create addresses as though it was assembled at P%. You only have to change line 40 to the appropriate address and rerun to make the machine-code relocate at the new address. You can even incorporate automatic saving of the machine-code into the BASIC parts of the program (lines 630 to 680). The other feature to note is the inclusion of jump statements at the start of the program. There is one jump statement for each subroutine. You should write down the order of these JMP instructions as these will be used in the main module.

```
10     REM
20     REM typical disk-based module
30     REM
40     DIM code% &1000
50     start% = &4100
60     FOR pass% = 4 TO 6 STEP 2
70     P% = start%
80     O% = code%
90     [OPT pass%
100    .start JMP suba \ suba
110           JMP subb \ subb
120           JMP subc \ subc
130           JMP subd \ subd
     :
     :
200 .suba    \Start of suba
     :
     :
290 RTS      \End of suba
```

```
 300 .subb               \ start of subb
    :
390  RTS                 \ end  of subb
400 .subc                \ start of subc
    :
490  RTS                 \ end  of subc
500 .subd                \ start of subd
    :
590  RTS                 \ end  of subd
600 .end   EQUB 0
610 ]
620 NEXT pass%
630 A$ = "SAVE MODULEA"
640 A$ = A$ + " "   + STR$~code%
650 A$ = A$ + " +"  + STR$~(end-start)
660 A$ = A$ + " "   + STR$~start%
670 A$ = A$ + " "   + STR$~start%
660 OSCLI A$
690 END
```

16.3.2 Main Module.

In the main module you also store, in start%, the load address of each of the disk-based machine-code modules. This is then the only line that you need to change. All subroutine calls are coded relative to that address. For example, to call subb in MODULEA:

```
100 start% = &4100
110 *LOAD MODULEA
120 CALL (start%+3) : REM JMP is 3 bytes long
```

16.4 Disk Overlay Summary

Disk overlay is a valuable technique that allows you to increase the permissible size of your programs considerably. But the technique requires a clear head. You should expect that any errors that arise in your logic will be more complex to solve than in a simple program. For example, if you call a PROC in a simple program, but you have forgotten to write that PROC, you receive the 'Bad call' error message; not unreasonable. In an overlay system, if you call a PROC, but you forget to *LOAD the correct module first, you may not get the 'Bad call' message. BASIC may know all about this PROC if you have loaded the module previously. It does not know, however, that you have swapped the module out. It will cheerfully jump to the correct address. What happens next depends on what code is at that address. Don't take error messages too literally, therefore. Keep an open mind and if necessary insert some extra PRINT statements to help trace the logic flow. You can minimise trouble by testing each of the disk-based modules in isolation.

Another point to remember is that you will only be able to open four disk files in your program, because the fifth will be needed for the overlays.

Naturally, you must not place time-sensitive code in a disk-based module. Avoid interrupt handlers or event interrupt code, for example.

You must also be mindful of disk conflicts. Be sure that the very act of loading a disk module to call a PROC does not conflict with some disk operation that your program is trying to achieve.

Lastly, be awake to the number of disk accesses that your overlay structure will require. A program which loads an overlay module for each PROC called will be slow and tedious. You should try to minimise disk access by careful selection of PROCs in each module.

17 Combined 40/80 Track

One of the problems which confronts the Software Houses is the fact that some BBC Microcomputer owners have 40 track drives, whilst others have 80 track. They clearly do not wish to produce software in both formats. This would increase distribution costs and double the cost of stocking for the dealer. The problem is solved by creating disks which can be read equally well by both types of drive. You may also find the technique described in this chapter to be helpful. Perhaps you want to market your software commercially; or perhaps you simply want to share it with your friends.

The central principle on which the technique relies is that computers have no brains. They just do what they are told -no more and no less. To use the technique yourself, you will need a switchable 40/80 track drive.

The key to the technique lies in the formatting of the destination diskette and for this you need to be switched to 80 track mode. The 80 track version of the program is stored in physical tracks 20 to 39. This allows for up to 50K of software, ample in most cases. You will probably recall from earlier chapters that a 40 track drive steps the head by twice the distance of an 80 track drive. Thus when a 40 track drive seeks track 20, it steps to physical track 40 of an 80 track diskette. Thus, if you place another copy of the software on the even numbered tracks between 40 and 78, and give them logical track numbers of 20 to 39, the software will be accessible to both types of drive. You will have fooled the disk drive in both cases. The special 40/80 track formatter program needs to accomplish the following steps:

format track 0

create a catalogue in track 0

enter a dummy file in this catalogue with a size that implies that it occupies all the disk sectors from track 0, sector 2 to track 19, sector 9

format physical tracks 20 to 39 as logical tracks 20 to 39 format even numbered tracks from 40 to 78 as logical tracks 20 to 39

The dummy file in the catalogue ensures that when you use *COPY to store your software on the disk, it is automatically stored starting at track 20, sector 0. In fact you will want to use *COPY in 80 track mode and then repeat it in 40 track mode. Figure 34 illustrates the formatting necessary.

physical track	logical track	purpose
00	00	catalogue
01 to 39	------	not used
20 to 39	20 to 39	80 trk progs
40	20	40 trk progs
41	--------------------	not used
42	21	40 trk progs
43	– ----	not used
44	22	40 trk progs
45	– ----	not used
46	23	40 trk progs
47	-----	not used
48	24	40 trk progs
49	-----	not used
50	25	40 trk progs
51	– ----	not used
52	26	40 trk progs
53	-----	not used
54	27	40 trk progs
55	-----	not used
56	28	40 trk progs
57	-----	not used
58	29	40 trk progs
59	-----	not used
60	30	40 trk progs
61	-----	not used
62	31	40 trk progs
63	-----	not used
64	32	40 trk progs
65	-----	not used
66	33	40 trk progs

67	-------------	not used
68	34	40 trk progs
69	-----	not used
70	35	40 trk progs
71	-----	not used
72	36	40 trk progs
73	-----	not used
74	37	40 trk progs
75	-----	not used
76	38	40 trk progs
77	-----	not used
78	39	40 trk progs
79	-------------	not used

Figure 34. Combined 40/80 track format.

17.1 The Formatter Program

Lines 40 to 210 are the main controlling part of the program. Firstly 3 DIMs are used to set aside memory space for the OSWORD parameter block (param), the format sector table (sectab), and the data to be placed in the two catalogue sectors (cat).

PROCbegin handles all the boring code - finding out which drive to format, making sure that the user is ready etc.

PROCinitsec creates the sector table. It has one parameter, the logical track number to be written into each sector id.

PROCformat formats a physical track.

Thus lines 80 to 170 are responsible for formatting track 0, tracks 20 to 39 and the even numbered tracks from 40 to 78.

PROCcreatecat creates the catalogue data.

PROCwritecat writes the two catalogue sectors on disk.

This program should be very straightforward to follow. It contains helpful REM statements and all the PROCs are in alphabetical order. You can refer to the chapter on OSWORD and the memory map chapter if necessary.

Note that a special problem can occur when a disk created by this program is read by one of the older 40 track drives with a full-width head. To minimise the amount of garbage picked up by this type of head, a brand-new, unformatted disk should be used.

```
10 REM
20 REM F4080 : Combined 40/80 track formatter
30 REM
40 DIM param 12, sectab 39, cat 511
50 osword = &FFF1
60 MODE 7
70 PROCbegin      :REM get drive
80 PROCinitsec(0) :REM create sectab for track 0
90 PROCformat(0)  :REM format track 0
100 FOR T% = 20 TO 39
110    PROCinitsec(T%) : REM create sectab
120    PROCformat(T%)  : REM format
130 NEXT
140 FOR T% = 40 TO 78 STEP 2
150    PROCinitsec(T% DIV 2)
160    PROCformat(T%)
170 NEXT
180 PROCcreatecat : REM set up catalogue
190 PROCwritecat  : REM write catalogue
200 PRINT TAB(6,22);CHR$133;"formatting complete"
210 END
220
230 DEF PROCbegin
240 REPEAT
250 PROCwhichdrive
260 UNTIL ans$ = "Y" OR ans$ = "y'
270 PROCinsert
280 ENDPROC
290
300 DEF PROCcreatecat
310 FOR I% = 0 TO 511
320    cat?I% =0      : REM initialize
```

```
530 NEXT
340 $(cat+8) = "dummy $"    : REM dummy filename
350 cat?&10  =0        : REM knock off &0D
360 cat?&105 =8        : REM no. files * 8
370 cat?&106 =3        : REM sectors on disk = &320
380 cat?&107 = &20
390 cat?&10D = 198    : REM dummy is 198 sectors
400 cat?&10F = 2       : REM starting at sector 2
410 ENDPROC
420
430 DEF PROCformat(track%)
440 ?param   = drive%      : REM drive
450 param!1  = sectab      : REM sector table
460 param?5  = &05        : REM no- of parameters
470 param?6  = &63        : REM command code
480 param?7  = track%      : REM physical track
490 param?8  = 21      : REM gap3
500 param?9  = &2A        : REM 10 sectors
510 param?10 =0        : REM gap5
520 param?11 = 16      : REM gapl
530 A% = &7F
540 X% = param MOD 256
550 Y% = param DIV 256
560 CALL osword
570 PRINT TAB(12,12);CHR$130;"track ";track%
580 IF param?12 = 0 ENDPROC
590 PRINT TAB(0,23);CHR$133;"format: bad result = &";~param?12
600 STOP
610 ENDPROC
620
630 DEF PROChead
640 CLS
650 PRINT TAB(0,2);CHR$141;CHR$131;"COMBINED 40/80 TRACK FORMATTER"
660 PRINT TAB(0,3);CHR$141;CHR$131;"COMBINED 40/80 TRACK FORMATTER"
670 ENDPROC
680
690 DEF PROCinitsec(logtrack%)
700 FOR I% = 0 TO 9
710   sectab?(I%*4)   = logtrack%
720   sectab?((I%*4)+1) = 0
730   sectab?((l%*4)+2) = I%
740   sectab?((I%*4)+3) = 1
750 NEXT
```

```
760 ENDPROC
770
780 DEF PROCinsert
790 PROChead
800 PRINT TAB(0,5);CHR$134;"insert diskette in drive ";drive%
810 PRINT TAB(0,7);CHR$134;"hit space bar when ready to go"
820 REPEAT
830 UNTIL INKEY(100) = 32
840 PROChead
850 ENDPROC
860
870 DEF PROCwhichdrive
880 PROChead
890 REPEAT
900 PRINT TAB(0,5);STRING$(38," ")
910 PRINT TAB(0,5);CHR$134;"enter drive to format (0-3) >";
920 INPUT " " drive%
930 UNTIL drive% > -1 AND drive% < 4
940 PROChead
950 PRINT TAB(0,5);CHR$134;"formatting drive ";drive%
960 REPEAT
970 PRINT TAB(0,7);STRING$(38," ")
980 PRINT TAB(0,7);CHR$134;"are you sure (Y/N) >";
990 INPUT " " ans$ 1000 ans$ = LEFT$(ans$,1>
1010 UNTIL ans$ = "Y" OR ans$ = "y" OR ans$ = "N" OR ans$ = "n"
1020 ENDPROC
1030
1040 PROCwritecat
1050 ?param   = drive%     : REM drive
1060 param!1  = cat        : REM catalogue data
1070 param?5  = &03        : REM no. of parameters
1080 param?6  = &4B        : REM command code
1090 param?7  = 0      : REM logical track
1100 param?8  = 0      : REM logical sector
1110 param?9  = &22    : REM 2  sectors
1120 A% = &7F
1130 X% = param MOD 256
1140 Y% = param DIV 256
1150 CALL osword
1160 IF param?10 = 0 ENDPROC
1170 PRINT TAB(0,23);CHR$133;"writecat: bad result = &";~param?10
1180 STOP
1190 ENDPROC
```

17.2 Copying Your Program(s)

When you have run this program, if you enter *INFO for the formatted drive, you will see:

```
$.dummy   000000 000000 00C600 002
```

This confirms that you have successfully fooled the DFS into believing that the disk contains a file called 'dummy' starting at sector 2 with a size of &C600 bytes. Simple arithmetic will tell you that &C600 is equal to 198*256. Thus you have reserved 198 sectors for 'dummy' and the next file that you copy will start at sector 200 (i.e. the start of track 20). Suppose that the formatted diskette is in drive 1 and suppose that you only have one file to put on the diskette, called 'PROG', which is on a diskette in drive 0. Keeping drive 1 switched to 80 track mode, you should enter:

```
    *COPY 0 1 prog
    *INFO :1.*
```

You will see the following display:

```
    S.PROG     FF1900 FF8023 000BF5 0C8
    $.dummy    000000 000000 00C600 002
```

Your program is now saved at sector C8 (i.e. track 20). You should now turn your microcomputer/disk drive off, switch drive 1 to 40 track mode, switch on again and repeat the previous keyboard entries:

```
    *COPY 0 1 PROG
    *INFO :1.*
```

You will see the following display:

```
    $.PR0G     FF1900 FF8023 000BF5 0C8
    S.dummy    000000 000000 00C600 002
```

The DFS believes that it has overwritten 'PROG' with a fresh copy of the same size. You of course know better. Because the drive was in 40 track mode, the copy has been placed on the even numbered tracks starting at track 40. The diskette created can be accessed equally well in a 40 track drive and an 80 track drive.

17.3 Refinements

Similar techniques are used commercially by Software Houses. However, the diskette created above is far too easy to reproduce. Although the diskette cannot be copied by *BACKUP, it can easily be copied by *COPY. Thus the Software Houses normally employ variations on this theme to impede piracy. Inevitably, commercial software includes purpose built disk load routines to handle the peculiar logical sectoring used in the diskettes. Other techniques include:

> using deleted data marks instead of data marks
>
> using different sector sizes and numbers of sectors per track
>
> incorrect catalogues
>
> using partially formatted diskettes, the load routine can then check for a track which should not exist, this stops people copying onto previously formatted diskettes.

You may well decide to adopt one or more of these tactics yourself. Whatever you decide, you will need a program that can clone the diskette once you have produced one operational version. This will enable you to go into mass production with ease. The following program is a general purpose disk copier, that handles many of the non-standard disk formats that you may wish to employ. It does not handle everything, however. When you have decided on your security measures, you simply make the necessary changes to the copy program.

17.4 Disk Copier

The program is well annotated and the PROCs are in alphabetical order. The chapter on OSWORD can be used to obtain an explanation of most of the code. The program assumes ten sectors per track throughout, each of 256 bytes.

The key to this program is the main loop (lines 110 to 490). For each track to be copied, the program:

seeks track 0 of the source diskette. This establishes a known physical position and resets the appropriate track register.

reads the sector ids for the track.

checks that sector ids were present, if not skips to next track.

finds the lowest logical track number from the ten sector ids.

sets the appropriate track register to the value of the logical track in the first sector id.

reads the complete track.

tells the user to swap diskettes, if the same drive is used for the source and destination diskettes.

seeks track 0 of the destination diskette. This establishes a known physical position and resets the appropriate track register.

formats the physical track, copying the format details from the source diskette.

sets the appropriate track register to the value of the logical track in the first sector id.

writes a track of data (read from the source diskette) to the destination diskette.

tells the user to swap diskettes, if the same drive is used for the source and destination diskettes.

The function of each PROC is as follows:

PROCbegin - find out from the user the source and
destination drives to be used.

PROCdisplay - display the sector ids (and any error conditions)
of the latest track read from the source disk.

PROCformat - format a track, the sector ids to be used are held in sectab.

PROChead - display screen title.

PROChowmany - find out how many tracks are on the source diskette and place the result (minus 1) in maxtrack%.

PROCinsert - get the user to insert his diskette(s) in the appropriate drives and find out when the user is ready to start.

PROClowest - from the 10 sector ids held in sectab, find out the lowest logical sector number.

PROCread - read all ten sectors of data from the track (into data) starting at the lowest logical sector number.

PROCsectorids - read all 10 sector ids of the next track of the source, diskette.

PROCseek0 - seek track 0 for the specified drive.

PROCswopi - ask user to insert the source diskette into the drive. Only called if the user has specified the same drive number for both input and output.

PROCswopo - ask user to insert the destination diskette into the drive. Only called if the user has specified the same drive number for both input and output.

PROCtrack - change the contents of the appropriate Track Register.

PROCwrite - write a whole track of data (from data) to the destination diskette.

```
10 REM  --
20 REM CLONE : Disk Copier
30 REM  --
40 DIM param 12, sectab 39, data 256*10-1
50 osword = &FFF1
60 MODE 7
```

```
70 PROCbegin : REM get source drive and dest drive
80 PROChowmany      : REM how many tracks on source
90 PROCinsert: REM get user ready
100
110 FOR T% = 0 TO maxtrack%
120
130    REM seek track 0 on source
HO     PROCseek0(sdrive%)
150
160    REM read sector ids from source
170    PROCsectorids(T%)
180
190    REM if no sectors loop back
200    IF param?10 <> 0 THEN 490
210
220    REM establish the lowest sector no
230    PROClowest
240
250    REM set source track register to logical
260    PROCtrack(sdrive%,?sectab)
270
280    REM read the data for this track
290    PROCread(?sectab)
300
310    REM if drives are the same - swop
320    IF sdrive% = ddrive% PROCswopo
330
340    REM seek track 0 on dest
350    PROCseek0(ddrive%)
360
370    REM copy source format to dest format
380    PROCformat(T%)
390
400    REM set dest track register to logical
410    PROCtrack(ddrive%,?sectab)
420
430    REM copy source data to dest data
440    PROCwrite(?sectab)
450
460    REM if drives are the same - swop
470    IF sdrive% = ddrive% PROCswopi
480
490 NEXT
```

```
500
510 PRINT TAB(0,22);CHR$133;"     cloning complete  "
520 END
530
540 DEF PROCbegin
550 REM find out source and destination drives
560 PROChead
570 REPEAT
580 PRINT TAB(0,5);STRING$(38," ")
590 PRINT TAB(0,5);CHR$134;"enter source    drive (0-3)
>";
600 INPUT " " sdrive%
610 UNTIL sdrive% > -1 AND sdrive% < 4
620 REPEAT
630 PRINT TAB(0,7);STRING$(38," ")
640 PRINT TAB(0,7);CHR$134;"enter destination drive (0-
3) >";
650 INPUT " " ddrive%
660 UNTIL ddrive% > -1 AND ddrive% < 4
670 ENDPROC
680
690 DEF PROCdisplay
700 REM display the track being processed
710 PROChead
720 PRINT TAB(0,5);CHR$134;" drive";CHR$130;sdrive%;
730 PRINT CHR$134;" physical track ";CHR$130;T%
740 IF param?10 <> 0 ENDPROC
750 PRINT TAB(00,7);CHR$134;"phys"
760 PRINT TAB(00,8);CHR$134;"sect"
770 PRINT TAB(06,7);CHR$134;"log "
780 PRINT TAB(06,8);CHR$134;"trck"
790 PRINT TAB(12,7);CHR$134;"head"
800 PRINT TAB(18,7);CHR$134;"log "
810 PRINT TAB(18,8);CHR$134;"sect"
820 PRINT TAB(24,7);CHR$134;"size"
830 @% = 3
840 FOR I% = 0 TO 9
850    J%=I%+10
860    PRINT TAB(02,J%);CHR$130;I%
870    PRINT TA8(08,J%);sectab?(4*I%)
880    PRINT TAB(14,J%);sectab?((4*I%)+1)
890    PRINT TAB(20,J%);sectab?((4*I%)+2)
900    PRINT TAB(26,J%);sectab?((4*I%)+3)
910 NEXT
920 ENDPROC
```

```
930
940 DEF PROCformat(track%)
950 REM format dest disk to match source
960 ?param  = &FF        : REM dest drive number
970 param!1 = sectab     : REM sector table
980 param?5 = &05        : REM number of parameters
990 param?6 = &63        : REM format command code
1000 param?7 = track%    : REM physical track number
1010 param?8 = 21        : REM gap3 size
1020 param?9 = &2A       : REM sector size/number of sectors
1030 param?10 =0         : REM gap5 size
1040 param?11 = 16       : REM gapl size
1050 A% = &7F
1060 X% = param       MOD 256
1070 Y% = param       DIV 256
1080 CALL osword
1090 IF param?12 = 0 ENDPROC
1100 PRINT TAB(4,22);CHR$133;"format: bad result = &";~param?12
1110 STOP
1120 ENDPROC
1130
1140 DEF PROChead
1150 REM screen header
1160 CLS
1170 PRINT TAB(8,2);CHR$141;CHR$131;"DISK CLONER"
1180 PRINT TAB(8,3);CHR$141;CHR$131;"DISK CLONER"
1190 ENDPROC
1200
1210 DEF PROChowmany
1220 REM find out how many tracks on source
1230 REM *DRIVE sdrive%
1240 OSCLI "DRIVE " + STR$sdrive%
1250 REM get no of sectors in &70
1260 A% = &7E
1270 X% = &70
1280 Y% = 0
1290 CALL osword
1300 maxtrack% = ((?&71+?&72*256) DIV 10)-1
1310 ENDPROC
1320
1330 DEF PROCinsert
1340 REM get disks in drive and wait for user
1350 IF sdrive% = ddrive% PROCswopi : ENDPROC
```

```
1360 PROChead
1370 PRINT TAB(0,5);CHR$134;
1373 PRINT "insert source disk in drive ";
1376 PRINT sdrive%
1380 PRINT TAB(0,6);CHR$134;
1383 PRINT "insert destination disk in drive ";
1386 PRINT ddrive%
1390 PRINT TAB(3,8);CHR$133;"hit space bar when ready to go"
1400 REPEAT
1410 UNTIL INKEY(100) = 32
1420 PROChead
1430 ENDPROC
1440
1450 DEF PROClowest
1460 REM find out lowest logical sector
1470 lowsec% = sectab?2
1480 FOR I% = 1 TO 9
1490   IF sectab?((4*I%)+2) < lowsec% lowsec% = sectab?((4*I%)+2)
1500 NEXT
1510 ENDPROC
1520
1530 DEF PROCread(track%)
1540 REM read a track from source
1550 ?param = &FF       : REM source drive number
1560 param!1 = data     : REM data buffer
1570 param?5 = &03      : REM number of parameters
1580 param?6 = &57      : REM read code
1590 param?7 = track%   : REM logical track number
1600 param?8 = lowsec%  : REM logical sector
1610 param?9 = &2A      : REM sector size/number of sectors
1620 A% = &7F
1630 X% = param MOD 256
1640 Y% = param DIV 256
1650 CALL osword
1660 del% = param?10 AND &20 : REM get delete bit
1670 res%  = param?10 AND &1E : REM mask off delete bit
1680 del$ = "ordinary data marks"
1690 IF del% <> 0 del$ = "deleted data marks"
1700 PRINT TAB(4,21);CHR$130;del$
1710 IF res% = 0 ENDPROC
1720 PRINT TAB(4,22);CHR$(133);"read : bad result = &";~param?10
1730 STOP
1740 ENDPROC
```

```
1750
1760 DEF PROCsectorids(track%)
1770 REM read  sector ids off source track
1780 ?param = &FF      : REM source drive number
1790 param!1 = sectab  : REM sector table
1800 param?5 = &03     : REM number of parameters
1810 param?6 = &5B     : REM read sector ids code
1820 param?7 = track%   : REM physical track number
1830 param?8 = 0
1840 param?9 = 10       : REM number of sectors
1850 A% = &7F
1860 X% = param NOD 256
1870 Y%  = param DIV 256
1880 CALL osword
1890 PROCdisplay
1900 IF param?10 = 0 ENDPROC
1910 PRINT TAB(4,22);CHR$(133);"ids : bad result = &";~param?10
1920 ENDPROC
1930
1940 DEF PROCseek0(drive%)
1950 REM seek track 0
1960 ?param = drive%    : REM drive
1970 param!1 = &FFFF    : REM no address
1980 param?5 =1       : REM number of parameters
1990 param?6 = &69    : REM seek command code
2000 param?7 =0       : REM track 0
2010 A% = &7F
2020 X% = param MOD 256
2030 Y% = param DIV 256
2040 CALL osword
2050 IF param?8 = 0 ENDPROC
2060 PRINT TAB(2,22);CHR$(133);"seek0: bad result = &";~param?8
2070 STOP
2080 ENDPROC
2090
2100 DEF PROCswopi
2110 REM swop back to source diskette
2120 PROChead
2130 PRINT TAB(0,5);CHR$134;"insert source disk in drive ";sdrive%
2140 PRINT TAB(0,8);CHR$134;"hit space bar when ready to go"
2150 REPEAT
2160 UNTIL INKEY(100) = 32
2170 PROChead
```

```
2180 ENDPROC
2190
2200 DEF PROCswopo
2210 REM swap back to destination diskette
2220 PROChead
2230 PRINT TAB(0,5);CHR$134;
2233 PRINT "insert destination disk in drive ";
2236 PRINT ddrive%
2240 PRINT TAB(0,8);CHR$134;"hit space bar when ready to go"
2250 REPEAT
2260 UNTIL INKEY(100) = 32
2270 PROChead
2280 ENDPROC
2290
2300 DEF PROCtrack(drive%,to%)
2310 REM alter track register
2320 ?param = &FF     : REM drive number
2330 param!1 = &FFFF  : REM dummy address
2340 param?5 = &02    : REM number of parameters
2350 param?6 = &7A    : REM write track reg code
2360 REM pick the correct track register.
2370 REM for drive 0/2 register = &12
2380 REM for drive 1/3 register = &1A
2390 param?7 = &12+8*(drive% MOD 2)
2400 param?8 = to%     :  REM new track number
2410 A% = &7F
2420 X% = param MOD 256
2430 Y% = param DIV 256
2440 CALL osword
2450 IF param?9 = 0 ENDPROC
2460 PRINT TAB(4,22);CHR$133;"track: bad result = &";~param?9
2470 STOP
2480 ENDPROC
2490
2500 DEF PROCwrite(track%)
2510 REM copy data from source track to dest track
2520 ?param  = &FF      : REM dest drive number
2530 param!1 = data     : REM data buffer
2540 param?5 = &03      : REM number of parameters
2550 REM pick correct write code
2560 REM if not deleted data use &4B
2570 REM if deleted data use &4F
2580 param?6 = &4B      : REM write code
```

```
2590 IF del% <> 0 param?6 = &4F
2600 param?7 = track%   : REM logical track number
2610 param?8 = lowsec%  : REM logical sector
2620 param?9 = &2A      : REM sector size/number of sectors
2630 A% = &7F
2640 X% = param MOD 256
2650 Y% = param DIV 256
2660 CALL osword
2670 res% = param?10 AND &1E : REM mask off delete bit
2680 IF res% = 0 ENDPROC
2690 PRINT TAB(4,22);CHR$(133);"write  : bad result = &";~param?10
2700 STOP
2710 ENDPROC
```

18 Disk Recovery

You can recover a diskette, on which you have inadvertently deleted one or more files, providing you have not compacted that diskette or saved new files onto it. This is possible because the *DELETE, *DESTROY and *WIPE commands only remove catalogue entries, not the files themselves. The process of recovering a diskette necessarily involves several steps, but the first and most difficult step can be accomplished by a computer program. Suppose that you have five BASIC programs on a diskette, the contents of which (as displayed by *INFO) are:

```
S.PROGA    FF1900 FF8023 000282 02E
S.PROGB    FF1900 FF8023 000171 029
S.PR0GC    FF1900 FF8023 000171 027
S.PR0GD    FF1900 FF8023 0012EB 014
S.PR0GE    FF1900 FF8023 0011A1 002
```

Figure 35. Diskette before the mishap.

Remember that *INFO displays files in the reverse order to their storage on the diskette. Suppose that you intend to delete PROGC but you unintentionally enter:

```
•DELETE PROGD
```

If you repeat the *INFO command, you will see:

```
S.PR0GA   FF1900  FF8023 000282 02E
S.PROGB   FF1900  FF8023 000171 029
S.PR0GC   FF1900  FF8023 000171 027
S.PR0GE   FF1900  FF8023 0011A1 002
```

Figure 36. Diskette after the mishap.

No program can recover the catalogue exactly for you, because knowledge of the missing file name and its size is lost forever. A program can, however, detect that there is a gap between 'PROGC and PROGE'. PROGE' starts at sector 2 and has a length of &11Al bytes. Therefore, it occupies a little more than &11 sectors (i.e. &12 sectors). Since all files start on a fresh sector, the next available sector after PROGE' = 2 + &12 = &14. But the next file, PROGC actually starts at &27. Therefore sectors &14 to &26 must contain one or more files which have been deleted. What is required is a program that, having detected the gap between PROGC' and PROGE', modifies the catalogue as follows:

```
S.PROGA   FF1900 FF8023 000382 02E
$.PR0GB   FF1900 FF8023 000171 029
$.PROGC   FF1900 FF8023 000171 027
S.DUMMY01 FF1900 000000 001300 014
S.PROGE   FF1900 FF8023 0011A1 002
```

Figure 37. Diskette partially recovered.

The program must give the missing file, a dummy name (in this case 'DUMMY01'). Note that the program cannot recognise if the gap between PROGC' and PROGE' originally contained more than one file and so it assumes that only one file has been deleted. It ascribes to 'DUMMY01' a length equal to the entire gap, because it has no knowledge of the size of the deleted file. It also ascribes a load address of &1900 (quite arbitrarily) and an execute address of zero. Since the deleted file was a BASIC program, all that remains for you to do is:

```
*RENAME DUMMY01 PR0GD
LOAD "PR0GD"
SAVE "PR0GD"
```

The BASIC command, LOAD, automatically stops when it encounters the end of the BASIC program. Note, however, that it is possible for LOAD to overwrite the screen area of memory and you should use Mode 7 to minimise this risk. Thus LOAD and SAVE suffice to correct the execute address and the length fields in the catalogue. If you repeat the *INFO command, you will see that you have fully recovered your deleted program:

```
S.PROGA   FF1900 FF8023 000282 02E
S.PR06B   FF1900 FF8023 000171 029
S.PR0GC   FF1900 FF8023 000171 027
$.PR0GD   FF1900 FF8023 0012EB 014
S.PR0GE   FF1900 FF8023 0011A1 002
```

Figure 38. Diskette fully recovered.

That was just about the simplest case of a disk recovery that can occur. Unfortunately, there are several complications that can arise, which we need to cover.

Suppose, for example, that you accidentally delete 'PROGA'. There is now no recognisable gap for the program to detect. The program can however automatically insert a dummy file entry covering all of the free space area (i.e. following the last used sector up to the last available sector on the diskette). If indeed this represents a deleted BASIC program, you can proceed as before. If no file has been deleted in the free space area, you should simply delete the dummy file created by the program.

Suppose that you delete more than one BASIC program. If the deleted files were not next to each other on the diskette, you should simply use *RENAME, LOAD and SAVE for each deleted file. But suppose that you inadvertently deleted both 'PROGC' and 'PROGD'. When you run the disk recovery program, followed by *INFO you would see:

```
S.DUMMY01    FF1900 000000 02C900 057
$.PROGA      FF1900 FF8023 000282 02E
S.PROGB      FF1900 FF8023 000171 029
$.DUMMY02    FF1900 000000 001500 014
S.PROGE      FF1900 FF8023 0011A1 002
```

Figure 39. Two contiguous deleted files.

You should now proceed as before, remembering to delete the dummy file that represents free space:

```
*DELETE DUMMY01
*RENAME DUMMY02 PROGD
LOAD "PROGD"
SAVE "PROGD"
```

You can now re-run the disk recovery program and a further *INFO will reveal:

```
$.DUMMY01 FF1900 000000 02C900 057
S.PR0GA   FF1900 FF8023 000282 02E
S.PR0GB   FF1900 FF8023 000171 029
S.DUMMY02 FF1900 000000 000200 027
$.PROGD   FF1900 FF8023 0012EB 014
S.PROGE   FF1900 FF8023 0011A1 002
```

Figure 40. Diskette after re-run.

To recover the diskette completely, all that remains is to enter:

```
*DELETE DUMMY01
*RENAME DUMMY02 PROGC
LOAD "PROGC"
SAVE "PROGC"
```

Of course there will be occasions when you mistakenly delete files which are not BASIC programs. For these you cannot use LOAD and SAVE. What you have to do is to find the end of the deleted file. Techniques include:

Searching for a known terminating sequence (as with BASIC files).

Searching for a character outside a given range (as with text files).

Searching for virgin, formatted sectors which are filled with bytes of &E5.

Using a special feature of a word-processor, such as the READ command in 'VIEW' (text files).

If all else fails, you can look at the file on the screen using *DUMP, but it can be tedious. Suppose that you have deleted a data file called 'DATA01'. You should run the disk recovery program, then enter:

```
*DELETE DUMMY01
*RENAME DUMMY02 DATA01
VDU14
*DUMP DATA01
```

This will allow you to view 'DATA01' on the screen in page mode. Scroll through the file until you find the end, and make a note of the relative byte address. Suppose that this is &2400. You should now enter:

```
VDU15
*LOAD "DATA01" 1900
*SAVE "DATA01" 1900+2400
```

Naturally you can substitute other load and execute addresses if appropriate (if the file was a machine-code program, say).

18.1 Disk Recovery Program

The program listed in this section will generate dummy entries in the catalogue for:

the free space area

each gap in the catalogue

Dummy filenames used start at 'DUMMY01' and are incremented as necessary. The program is quite straightforward to follow, with PROCs in alphabetical order and frequent REM statements. The memory map chapter provides the necessary information to understand this program.

The main loop of the program occurs from lines 130 to 210.

PROCinit is called to find out which drive to recover.

PROCreadcat reads the catalogue into a 512 byte area called 'cat'.

PROCextract extracts certain catalogue information. The number of files on the diskette is stored in fnum% and the number of sectors in snum%. Actual file information is transferred into arrays as follows:

> fnam$ = array for filenames
> load% = array for load addresses
> exec% = array for execute addresses
> leng% = array for file lengths
> strt% = array for start sectors
> fini% = array for end sectors

PROCeof adjusts these arrays to generate a dummy file which covers all the free space on the diskette.

PROCnewfiles looks for gaps in the arrays and generates a dummy file entry for each gap.

PROCprintcat uses the arrays to display the new catalogue. Note that original files on the dikette are displayed in green, whilst program-generated dummy files are displayed in red. For each file, the display consists of:

> filename (including directory)
> lock flag
> load address (in hex)
> execute address (in hex)
> start sector (in hex)
> end sector (in hex)
> length in bytes (in hex)

PROCnewcat uses the arrays to update the copy of the catalogue in the memory area, 'cat'.

PROCwritecat writes this new catalogue back to disk.

The functions of other PROCs in this program are as follows:

PROChead diplays a screen title.

PROCinsert inserts a dummy file into the arrays.

PROCmakeroom shifts the arrays up 1 to make room for a dummy file to be inserted.

PROCoscli executes a call to OSCLI.

PROCosword executes a call to OSWORD.

```
2  REM RESCUE : disk recovery
3  REM-----------------
10 M0DE7
20 DIM osc 60      :REM OSCLI  area
30 DIM cat 511     :REM catalogue
40 DIM param 13    :REM OSWORD parameter block
50 DIM fnam$(31)   :REM filenames
60 DIM load%(31)   :REM  load addresses
70 DIM exec%(31)   :REM exec addresses
80 DIM strt%(31)   :REM start sectors
90 DIM leng%(31)   :REM  lengths
100 DIM fini%(31)  :REM end sectors
110 DIM colr$(31)  :REM colours
120
130 PROCinit       :REM which drive etc.
140 PROCreadcat    :REM read catalogue
150 PROCextract    :REM extract from cat
160 PROCeof        :REM check EOF
170 PROCnewfiles   :REM revise files
180 PROCprintcat   :REM print catalogue
190 PROCnewcat     :REM make new catalogue
200 PROCwritecat   :REM write catalogue
210 END
220
230 DEF PROCeof
240 IF fini%(1) = snum%-1 ENDPROC
250 PROCinsert(1)
260 fini%(1) = snum% - 1
270 leng%(1) = fini%(1) - strt%(1) + 1
```

```
280 leng%(1) = 256*leng%(1)
290 ENDPROC
300
310 DEF PROCextract
320 REM extract fields from the catalogue
330 REM get number of files
340 fnum% = cat?&105 DIV 8
350 REM get number of sectors
360 snum% = cat?&107 + 256*(cat?&106 AND 3)
370 REM if no files - leave
380 IF fnum% = 0 ENDPROC
390 REM get file details
400 FOR I% = 1 TO fnum%
410   fnst% = cat + 8*I%  : REM start of filename
420   adst% = fnst% + &100 : REM start of addresses
430   REM get file name
440    FOR J% = 0 TO 7
450       fnam$(I%) = fnam$(I%) + CHR$(fnst%?J%)
460    NEXT
470   REM get load address
480    load%(I%) = ?adst%
490    load%(I%) = load%(I%) + &100*adst%?1
500    load%(I%) = load%(I%) + &10000*((adst%?6 AND &C) DIV 4)
510    REM get exec address
520   exec%(I%) = adst%?2
530   exec%(I%) = exec%(I%) + &100*adst%?3
540   exec%(I%) = exec%(I%) + &10000*((adst%?6 AND &C0) DIV 64)
550   REM get length
560   leng%(I%) = adst?4
570    leng%(I%) = leng%(I%) + &100*adst%?5
580    leng%(I%) = leng%(I%) +  &10000*((adst%?6 AND &30) DIV 16)
590   REM get start sector
600   strt%(I%) = adst%?7
610   strt%(I%) = strt%(I%) + &100*(adst%?6 AND 3)
620   REM calculate end sector
630   fini%(I%) = strt%(I%) + leng%(I%) DIV 256
640 NEXT
650 ENDPROC
660
670 DEF PROChead
680 REM screen title
690 CLS
700 PRINT TAB(11,2);CHR$141;CHR$131;"RESCUE"
710 PRINT TAB(11,3);CHR$141;CHR$131;"RESCUE"
```

```
720 ENDPROC
730
740 DEF PROCinit
750 REM initialise no. of dummy files
760 dumm% = 0
770 REM initialise DIMS.
780 FOR I% = 1 TO 31
790    colr$(I%) = CHR$130 : REM green
800    fnam$(I%) = ""     : REM initialise filenames
810 NEXT
820 REM which drive :-
830 PROChead
840 REPEAT
850 PRINT TAB(4,8);STRING$(35," ")
860 PRINT TAB(4,8);CHR$134;"which drive (0-3) ";
870 INPUT "> " drive%
880 UNTIL drive% > -1 AND drive% < 4
890 $osc = "DRIVE " + STR$(drive%)
900 PROCoscli
910 REM get user ready
920 PROChead
930 PRINT TAB(4,8);CHR$134;"insert diskette in drive ";drive%
940 PRINT TAB(4,9);CHR$134;"hit space bar when ready to go"
950 REPEAT UNTIL INKEY(100) = 32
960 ENDPROC
970
980 DEF PROCinsert(J%)
990 PROCmakeroom(J%+1)
1000 dumm% = dumm% + 1
1010 dumm$ = STR$(dumm%)
1020 IF dumm% < 10 dumm$ = "0" + dumm$
1030 fnam$(J%) = "DUMMY" + dumm$ + "$"
1040 fini%(J%) = strt%(J%-1) - 1
1050 strt%(J%) = fini%(J%+1) + 1
1060 leng%(J%) = fini%(J%) - strt%(J%) + 1
1070 leng%(J%) = 256*leng%(J%)
1080 colr$(J%) = CHR$129
1090 load%(J%) = &31900
1100 exec%(J%) = 0
1110 ENDPROC
1120
1130 DEF PROCnewcat
1140 cat?&105 = fnum% * 8    : REM number of files * 8
1150 FOR I%  = 1 TO fnum%
```

```
1160    fnst% = cat + 8*I%  : REM start of filename
1170    adst% = fnst%+&100  : REM start of addresses
1180    FOR J% = 1 TO LEN(fnam$(I%))
1190       fnst%?(J%-1) = ASC(MID$(fnam$(I%),J%,1))
1200    NEXT
1210    !adst% = load%(I%)
1220    adst%!2 = exec%(I%)
1230    adst%!4 = leng%(I%)
1240    adst%?7 = strt%(I%)
1250    ex%    = exec%(I%) DIV &10000
1260    le%    = leng%(I%) DIV &10000
1270    lo%    = load%(I%) DIV &10000
1280    st%    = strt%(I%) DIV &100
1290    adst%?6 = 64*ex% + 16*le% + 4*lo% + st%
1300 NEXT
1310 ENDPROC
1320
1330 DEF PROCnewfiles
1340 IF fnum% = 0 ENDPROC
1350 I% = 1
1360 REPEAT
1370 I% = I% + 1
1380 gap% = strt%(I%-1) - fini%(I%) - 1
1390 IF gap% <> 0 PROCinsert(I%)
1400 UNTIL I% = fnum%
1410 ENDPROC
1420
1430 DEF PROCmakeroom(J%)
1440 REM shift filenames up to make room
1450 fnum% = fnum% + 1
1460 FOR K% = fnum% TO J% STEP -1
1470    fnam$(K%) = fnam$(K%-1)
1480    load%(K%) = load%(K%-1)
1490    exec%(K%) = exec%(K%-1)
1500    strt%(K%) = strt%(K%-1)
1510    leng%(K%) = leng%(K%-1)
1520    fini%(K%) = fini%(K%-1)
1530    colr$(K%) = colr$(K%-1)
1540 NEXT
1550 ENDPROC
1560
1570 DEF PROCoscli
1580 X% = osc MOD 256
1590 Y% = osc DIV 256
```

```
1600 CALL &FFF7
1610 ENDPROC
1620
1630 DEF PROCosword
1640 A% = &7F
1650 X% = param MOD 256
1660 Y% = param DIV 256
1670 CALL &FFF1
1680 ENDPROC
1690
1700 DEF PROCprintcat
1710 CLS
1720 PROChead
1730 PRINT
1740 IF fnum% = 0 ENDPROC
1750 FOR I% = 1 TO fnum%
1760   PRINT colr$(I%);
1770   dir$ = RIGHT$(fnam$(I%),1)
1780   dir% = ASC(dirS) AND &7F
1790   lok% = ASC(dir$) AND &80
1800   dir$ = CHR$(dir%)
1810   lok$ = "L"
1820   IF lok% = 0 lok$ = " "
1830   nam$ = LEFT$(fnam$(I%),7)
1840   PRINT dirS;".";
1850   PRINT nam$;" ";
1860   PRINT lok$;" ";
1870   a% = 5
1880   PRINT ~load%(I%);" ";
1890   PRINT ~exec%(I%);" ";
1900   @% = 3
1910   PRINT ~strt%(I%);" - ";
1920   PRINT ~fini%(I%);" ";
1930   PRINT ~leng%(I%)
1940 NEXT
1950 ENDPROC
1960
1970 DEF PROCreadcat
1980 REM read 2 catalogue sectors into cat
1990 ?param = drive% : REM drive number
2000 param!1 = cat   : REM buffer address
2010 param?5 = 3     : REM no. of params
2020 param?6 = &53   : REM read command
2030 param?7 =0      : REM track number
```

```
2040 param?8 = 0     : REM sector number
2050 param?9 = &2 2  : REM two sectors
2060 PROCosword      : REM read the cat
2070 IF param?10 = 0 ENDPROC
2080 PRINT TAB(4,10);CHR$129;
2090 PRINT "error reading catalogue = &";
2100 PRINT ~param?10
2110 STOP
2120 ENDPROC
2130
2140 DEF PROCwritecat
2150 REM update 2 catalogue sectors
2160 ?param = drive%
2170 param!1 = cat
2180 param?5 = 3
2190 param?6 = &4B
2200 param?7 = 0
2210 param?8 = 0
2220 param?9 = &22
2230 PROCosword
2240 IF param?10 = 0 ENDPROC
2250 PRINT TAB(4,10);CHR$129;
2260 PRINT "error writing catalogue = &";
2270 PRINT ~param?10
2280 STOP
2290 ENDPROC
```

18.2 Use of the Program

The way in which this program should be used has already been described at the start of this chapter. However, there are a few more points about its use that can save you some heartache.

Firstly, it is a good idea to avoid the need to use this program at all. You cannot safeguard against making mistakes at the keyboard: everybody does on occasions. But you can make regular backup copies of your important data and programs.

Secondly, before you use the disk recovery program, it is a good idea to make a backup copy of the entire diskette which you have corrupted. You must use *BACKUP, (not *COPY). If the diskette that you are trying to recover uses non-standard logical sectoring techniques, you can make a backup copy using the CLONE program outlined in the previous chapter. If you make a further mistake in the recovery attempt, you have a backup to fall back on. You certainly need a clear head (and peace and quiet) when recovering a diskette. Mistakes are easily made.

If you want to identify the end of an ASCII text file which is not too large, and you have a word processor ROM, you might find it easier to load the file into the word processor, rather than use *DUMP. You can simply delete the unwanted bytes at the end of the file and then save it again.

Machine-code programs are often most easily resolved with a disassembler, if you have one at your disposal.

The hardest files to recover are databases, especially if you use a commercial package. There is virtually no prospect of recovering these unless you are intimately acquainted with the physical structure of the database.

Lastly, there are some files which you cannot *LOAD and *SAVE because they are too big. Fortunately, there are many alternative ways to adjust the length field in the catalogue. You could use one of the many disk sector editors published in magazines, for example, or indeed the one published in the next chapter.

Editor's note: The disk editor provided in the next section may be effective enough, but 'sideways' ROMs' are widely available on the internet which provide this type of facility and more besides. Eg Disk Doctor.

19 Disk Editor

The previous chapter outlined one usage of a disk editor program. In fact the uses are many and various. A disk editor can be used simply to view a disk file on the screen. It can also be used to patch up a disk file which has become corrupted in some way. The program listed in this chapter has several features not found in most of the editors published in magazines:

It works in MODE 7. This makes the display clearer, especially if you do not own a monitor. However, it means that only half a sector can be displayed on the screen at a time. Swapping to the other half is simply achieved by pressing function key 0.

It handles logical sectoring, in a manner similar to the CLONE routine, previously described in this book.

The screen display uses colours. The relative address of the start of each line is displayed in green. Eight hexadecimal bytes are then displayed in yellow. Finally, the ASCII representation of those bytes is displayed in red. The choice of red may not be ideal for those with monochrome monitors. You can change the colour to cyan, simply by substituting CHR$134 for CHR$129 throughout the program.

You may edit either the hex part or the ASCII part, or both. You use the normal cursor keys to position the cursor.

The editor ensures that the cursor is always positioned over a hex or ASCII character that can be amended. The program provides full wrap around of the cursor top to bottom and side to side.

The program keeps you informed of your position on the diskette. Both physical and logical track numbers are displayed.

When the programs asks you which sector to display, it specifics (and will only accept) the logical sector numbers found on the disk.

Errors returned from OSWORD are reported, but the program allows you to continue anyway.

If you are editing a sector with deleted data marks, the program writes it back with deleted data marks.

All-in-all, you should find that this editor handles most of your needs. If you have built extra security features into your own diskettes, it should be a straightforward matter to amend this program in line with your features. The program is well annotated and all the PROCs are in alphabetical order. Thus it should be fairly simple to follow. These notes describe some of the features used in the program with which you may not be familiar.

The program takes complete control of the cursor. At line 480, *FX4,1 disables normal cursor editing and makes the cursor keys return:

LEFT	&88
RIGHT	&89
DOWN	&8A
UP	&8B

The program uses the function keys (f0 to f3) for other functions. Lines 200 to 230 make these keys return:

f0	&8C	display other half of sector
f1	&8D	return to main menu
f2	&8E	update this sector on disk
f3	&8F	quit the program

Let us look at line 200 in some detail to see how these values are derived. The sequence |!causes the OS to add &80 to the rest of the sequence. The sequence |L means 'subtract &40 from the ASCII value for L' Since L has a hexadecimal value of &4C, |L has a value of &C. Thus the whole value is calculated as:

value = &80 + &4C - &40 = &8C (just what we wanted)

Throughout the program, part% is used to determine which half of the sector is displayed. It is initialised to zero, meaning that the first 128 bytes are displayed. When it is set to 1, the last 128 bytes are displayed. At line 1700, part% is toggled by the statement:

```
      part% = part% EOR 1
```

This changes 0 to 1 or 1 to 0, depending on the current value of part%.

Line 330 uses OSCLI to perform a *DRIVE. This is a BASIC II command. See Chapter 16 for an explanation of how to use OSCLI with BASIC I.

If you decide to key this program in for yourself, you may ignore all REM statements. There are no GOTO or GOSUB statements in the program and line numbers have no significance.

```
10 REM
20 REM EDIT : disk editor
30 REM
40
50 DIM param 12, sectab 39, data 255
60 osword = &FFF1
70 MODE 7
80 PROCbegin          : REM get drive
90 REPEAT             : REM repeat
100 PROCmain          : REM main loop
110 MODE 7            : REM normal cursor
120 UNTIL get% = &8F  : REM until user finished
130 END
140
150
160 DEF PROCbegin
170 VDU23;11,0;0;0;0;
180 REM set KEY0 to &8C KEY1 to &8D
190 REM set KEY2 to &8E KEY3 to &8F
200 *KEY0 |!|L
210 *KEY1 |!|M
220 *KEY2 |!|N
230 *KEY3 |!|O
```

```
240 REM display header
250 PROChead
260 REM get drive number
270 REPEAT
280 PRINT TAB(3,5);STRING$(38," ")
290 PRINT TAB(3,5);CHR$130;"enter drive (0-3)";
300 INPUT " > " drive%
310 UNTIL drive% > -1 AND drive% < 4
320 REM *DRIVE for this drive number
330 OSCLI "DRIVE " + STR$(drive%)
340 REM get user to insert diskette
350 PRINT TAB(3,7);CHR$130;"insert disk in drive ";drive%
360 PRINT TAB(3,8);CHR$130;"hit space bar when ready to go"
370 REPEAT UNTIL INKEY(100) = 32
380 REM find out how many tracks
390 A% = &7E
400 X% = &70
410 Y% = 0
420 CALL osword
430 maxtrack% = ((?&71+?&72*256) DIV 10)-1
440 ENDPROC
450
460 DEF PROCchoices
470 REM disable cursor editing
480 *FX 4,1
490 REM display choices
500 PRINT TAB(0,21);CHR$133;"you may edit the";
510 PRINT CHR$131;"hex";
520 PRINT CHR$133;"or the";
530 PRINT CHR$129;"ASCII"
540 PRINT TAB(0,22);"(f0 = other half : f1 = main menu  )"
550 PRINT TAB(0,23);"(f2 = update disk : f3 = quit program)"
560 ENDPROC
570
580 DEF PROCcoord
590 REM get cursor coordinates
600 x% = POS
610 y% = VPOS
620 ENDPROC
630
640 DEF PROCdispdata
650 REM display header
660 PROChead
```

```
670 REM display a 16 by 8 matrix of data
680 FOR I% = 1 TO 16
690   PROCdispline(I%)
700 NEXT
710 REM display choices
720 PROCchoices
730 REM position cursor at 1st byte
740 VDU31,5,4
750 ENDPROC
760
770 DEF PROCdispline(W%)
780 REM display a line of data
790 REM set J% = relative byte number
800 J% = part%*128 + 8*(W%-1)
810 REM set start% to address of 1st byte of data
820 start% = data + J%
830 REM set str$ to 2 byte hex string of J%
840 PROCstring(J%)
850 REM print it in green and set up rest in yellow
860 PRINT TAB(0,W%+3);CHR$130;str$;CHR$131;" ";
870 REM initialise ASCII representation of data
880 A$ = ""
890 REM for each of 8 bytes of data
900 FOR Z% = 0 TO 7
910   REM set C% = next byte
920   C% = start%?Z%
930   REM store ASCII representation in A$
940   IF C%<32 OR C%>122 A$ = A$ + "." ELSE A$ = A$ + CHR$(C%)
950   REM set str$ = 2 byte hex string of C%
960   PROCstring(C%)
970   REM print it
980   PRINT str$;" ";
990 NEXT
1000 REM print ASCII representation in red
1010 PRINT CHR$129;A$
1020 ENDPROC
1030
1040 DEF PROCdown
1050 REM move cursor down 1 line
1060 VDU 10
1070 REM get cursor coordinates
1080 PROCcoord
1090 REM if it's fallen off the bottom, wrap round to top
```

```
1100 IF y% > 19 VDU31,x%,4
1110 ENDPROC
1120
1130 DEF PROCedit
1140 REM get cursor coordinates
1150 PROCcoord
1160 REM if user editing ASCII do edit2
1170 IF x% > 29 PROCedit2 : ENDPROC
1180 REM user is editing hex, so change key-in from ASCII to hex
1190 get% = get% - &30
1200 IF get% > 9  get% = get% - 7
1210 REM make sure the key-in is valid
1220 IF get% < 0  ENDPROC
1230 IF get% > &15 ENDPROC
1240 REM calculate relative memory address of byte being edited
1250 mem% = part%*128 + 8*(y%-4) + (x%-5) DIV 3
1260 REM find existing value of left nibble
1270 left% = data?mem% DIV 16
1280 REM find existing value of right nibble
1290 right% = data?mem% MOD 16
1300 REM if user is editing left nibble, overwrite left nibble
1310 REM else overwrite right nibble
1320 IF (x%-5> MOD 3 = 0 left% = get% ELSE right* = get%
1330 REM put new value back in memory
1340 data?mem% = 16*left% + right%
1350 REM re-display whole line
1360 PROCdispline(y%-3)
1370 REM restore cursor position
1380 VDU31,x%,y%
1390 REM move cursor right
1400 PROCright
1410 ENDPROC
1420
1430 DEF PROCedit2
1440 REM get relative address of byte being edited
1450 mem% = part%*128 + 8*(y%-4) + x% - 30
1460 REM overwrite this byte with key-in value
1470 data?mem% = get%
1480 REM re-display whole line
1490 PROCdispline(y%-3)
1500 REM restore cursor position
1510 VDU31,x%,y%
1520 REM move cursor right
1530 PROCright
```

```
1540 ENDPROC
1550
1560 DEF PROCerr
1570 REM handle a bad result from OSWORD
1580 PROChead
1590 PRINT TAB(0,5);CHR$129;bad$
1600 PRINT TAB(0,6);CHR$129;"result = &";STR$~(res%)
1610 PRINT TAB(0,8);CHR$133;"hit space bar for main menu"
1620 REPEAT UNTIL INKEY(100) = 32
1630 ENDPROC
1640
1650 DEF PROCget
1660 REM get keyboard input
1670 REPEAT
1680 get% = GET
1690 REM if user entered f0, toggle next half of sector display
1700 IF get% = &8C part% = part% EOR 1 : PROCdispdata
1710 REM if user entered f2, update the disk
1720 IF get% = &8E PROCwrite
1730 REM if cursor key, move cursor
1740 IF get% = &88 PROCleft
1750 IF get% = &89 PROCright
1760 IF get% = &8A PROCdown
1770 IF get% = &8B PROCup
1780 REM if editing a byte, do edit
1790 IF get% > &1F AND get% < &80 PROCedit
1800 REM repeat all this until user wants to
1810 REM return to main menu, update disk or quit
1820 UNTIL get% > &8C
1830 ENDPROC
1840
1850 DEF PROChead
1860 REM display screen header
1870 CLS
1880 PRINT TAB(10,0);CHR$141;CHR$132;"DISK EDITOR"
1890 PRINT TAB(10,1);CHR$141;CHR$132;"DISK EDITOR"
1900 ENDPROC
1910
1920 DEF PROCleft
1930 REM move cursor left
1940 VDU 8
1950 REM get cursor coordinates
1960 PROCcoord
1970 REM if it's fallen off at the left, wrap round to the right
```

```
1980 IF x% < 5 VDU31,37,y% : ENDPROC
1990 REM if it's at X = 28 or 29, move it to X = 30
2000 IF x% > 29 ENDPROC
2010 IF x% = 28 VDU8  : ENDPROC
2020 IF x% = 29 VDU8,8 : ENDPROC
2030 REM if it's in a gap, move it left
2040 IF (x%-4) MOD 3=0 VDU8
2050 ENDPROC
2060
2070 DEF PROCmain
2080 VDU23;11,0;0;0;0;
2090 REM main program loop
2100 REM find out which track
2110 PROCmenu
2120 REM seek track 0. reset track register
2130 PROCseek(0)
2140 REM read sector ids off wanted track
2150 PROCsectorids
2160 REM any problems, return to main
2170 IF res% <> 0 ENDPROC
2180 REM find out which sector
2190 PROCwhichsec
2200 REM set track register
2210 PROCtrack(?sectab)
2220 REM read the sector
2230 PROCread
2240 REM any problems, return to main
2250 IF res% <> 0 ENDPROC
2260 REM initialise display to 1st 128 bytes of sector
2270 part% = 0
2280 REM special editing cursor
2290 VDU23;10,0;0;0;0;
2300 VDU23;11,255;0;0;0;
2310 REM display 128 bytes of data
2320 PROCdispdata
2330 REM start reading from keyboard
2340 PROCget
2350 ENDPROC
2360
2370 DEF PROCmenu
2380 REM find out which track
2390 @% = 2
2400 PROChead
2410 REPEAT
```

```
2420 PRINT TAB(0,5);STRING$(38," ")
2430 PRINT TAB(0,5);CHR$130;"enter track number (0 -
";maxtrack%;")";
2440 INPUT " > " track%
2450 UNTIL track% > -1 AND track% < maxtrack%+1
2460 ENDPROC
2470
2480 DEF PROCow(R%)
2490 REM call OSWORD and set res% to result register.
2500 REM on entry, R% = index into param for result register.
2510 A% = &7F
2520 X% = param MOD 256
2530 Y% = param DIV 256
2540 CALL osword
2550 res% = param?R%
2560 ENDPROC
2570
2580 DEF PROCread
2590 REM read a sector
2600 ?param = drive%   : REM drive
2610 param!1 = data    : REM address
2620 param?5 = 3       : REM no. of parameters
2630 param?6 = &57     : REM read
2640 param?7 = ?sectab : REM track
2650 param?8 = sector% : REM sector
2660 param?9 = &21     : REM one sector
2670 PROCow(10)
2680 REM save delete bit from result register
2690 del% = res% AND &20
2700 REM mask off the deleted data bit
2710 res% = res% AND &1E
2720 REM report any problems
2730 IF res% = 0 ENDPROC
2740 bad$ = "sector " + STR$(sector%) + " bad read"
2750 PROCerr
2760 ENDPROC
2770
2780 DEF PROCright
2790 REM move cursor right
2800 VDU 9
2810 REM get cursor coordinates
2820 PROCcoord
2830 REM if cursor drops off right, wrap round to the left
2840 IF x% >37 VDU31,5,y% : ENDPROC
2850 IF x% > 29 ENDPROC
```

```
2860 REM don't leave cursor in a gap
2870 IF x% = 28 VDU9,9 : ENDPROC
2880 IF x% = 29 VDU9 : ENDPROC
2890 IF  (x%-4> MOD 3 = 0 VDU9
2900 ENDPROC
2910
2920 DEF PROCsectorids
2930 REM read sector ids
2940 ?param  = drive% : REM drive
2950 param!l = sectab : REM address
2960 param?5 = 3      : REM no. of  parameters
2970 param?6 =&5B     : REM sectorids
2980 param?7 = track% : REM physical track
2990 param?8 = 0      : REM dummy
3000 param?9 = 10     : REM no.  of sectors
3010 PROCow(10)
3020 REM report any problems
3030 IF  res% = 0 ENDPROC
3040 bad$ = "track " + STR$(track%) +" missing sector ids"
3050 PROCerr
3060 ENDPROC
3070
3080 DEF PROCseek(T%)
3090 REM seek a track
3100 ?param = drive%  : REM drive
3110 param!1 = &FFFF  : REM no address
3120 param?5 = 1      : REM no. parameters
3130 param?6 = &69    : REM Seek
3140 param?7 = T%     : REM physical track
3150 PROCow(8)
3160 IF res% = 0 ENDPROC
3170 REM report any problems
3180 bad$ = "track " + STR$(track%) + "  missing"
3190 PROCerr
3200 ENDPROC
3210
3220 DEF PROCstring(num%)
3230 REM make a 2 byte hex string (str$) of a number
3240 str$ = STR$~num%
3250 REPEAT
3260 IF LEN(str$) < 2 str$ = "0" + str$
3270 UNTIL LEN(str$) = 2
3280 ENDPROC
3290 REM Editor's note: Easier to use str$=RIGHT$("0"+STR$~num%,2)
```

```
3300 DEF PROCtrack(T%)
3310 REM adjust track register
3320 ?param = drive%   : REM drive
3330 param!1 = &FFFF   : REM address
3340 param?5 = 2       : REM no. of parameters
3350 param?6 = &7A     : REM write spec reg
3360 REM pick correct track register
3370 IF drive% MOD 2 = 1 reg% = &1A ELSE reg% = &12
3380 param?7 = reg%    : REM reg
3390 param?8 = T%     : REM value
3400 PROCow(9)
3410 ENDPROC
3420
3430 DEF PROCup
3440 REM move cursor up 1 line
3450 VDU 11
3460 REM get cursor coordinates
3470 PROCcoord
3480 REM if cursor falls off the top, wrap round to bottom
3490 IF y% < 4 VDU 31,x%,19
3500 ENDPROC
3510
3520 DEF PROCwhichsec
3530 REM find out lowest sector id
3540 lowsec% = sectab?2
3550 FOR I% = 1 TO 9
3560   IF sectab?((4*I%)+2)<lowsec% lowsec% = sectab?((4*I%)+2)
3570 NEXT
3580 REM display header
3590 PROChead
3600 REM display track details
3610 @% = 2
3620 PRINT TAB(5,5);CHR$131;"you are positioned on"
3630 PRINT TAB(8,7);CHR$133;"physical track ";track%
3640 PRINT TAB(8,8);CHR$133;" logical track ";?sectab
3650 REM get a valid sector id
3660 REPEAT
3670 PRINT TAB(0,10);STRING$(38," ")
3680 PRINT TAB(0,10);CHR$130;"enter sector number (";
3685 PRINT lowsec%;" - ";lowsec%+9;")";
3690 INPUT " > " sector%
3700 UNTIL sector% > lowsec%-1 AND sector% < lowsec%+10
3710 ENDPROC
3720
```

```
3730 DEF PROCwrite
3740 REM write a sector back to disk
3750 ?param  = drive%  : REM drive
3760 param!l = data    : REM address
3770 param?5 = 3       : REM no. of parameters
3780 REM if deleted data read, write back deleted data
3790 IF del% <> 0 code% = &4F ELSE code% = &4B
3800 param?6 = code%    : REM write
3810 param?7 = ?sectab  : REM track
3820 param?8 = sector%  : REM sector
3830 param?9 = &21      : REM one sector
3840 PROCow(10)
3850 REM report any problems
3860 res% = res% AND &1E : REM mask off delete bit
3870 IF res% = 0 ENDPROC
3880 bad$ = "sector " + STR$(sector%) + " bad write"
3890 PROCerr
3900 ENDPROC
```

PART 3
FILES

20 Access Methods

Chapter 14 introduced you briefly to the possibilities of randomly accessing a file. In this chapter, the theme is considerably expanded. There are an enormous number of techniques available to you and we shall try to cover the main ones here.

First, let us recall some of the buzz-words used to describe filing systems. A file consists of a number of records. Although it is possible to create a file with different record types, normally there is only one and so each record has an identical format. Each record consists of a number of fields and each field contains some basic piece of information. At least one of the fields, known as a 'key field', is more important than the others, since it uniquely defines the record. Some records require more than one key field to define them uniquely. Put another way, key fields are the fields that the user would enter at the keyboard in order to access a particular record. The less important fields simply add more information.

Let's use a telephone directory as an example. Each record in the file contains information about one person's address and telephone number. The key fields are the surname and the initials of the telephone subscriber. The address and telephone number represent the other fields. Thus each record consists of:

surname	main key field
initials	secondary key field
address	field
telephone number	field

20.1 Fixed Length/Variable Length

You should realise that the data itself can be stored in one of two ways - as 'fixed length records' or as 'variable length records'. In a fixed length record we assign to each of the fields the maximum length of that field. We then ensure that each occurrence of that field is padded out (possibly with trailing spaces) to its maximum length. For example, we could define maximum lengths as follows:

surname	36 bytes
initials	8 bytes
address	200 bytes
telephone number	20 bytes
total record length	264 bytes

Thus each record would occupy exactly 264 bytes. This has the advantage that we can use PTR# to locate the start of each subsequent record, simply by adding 264 to it. It has one big disadvantage, however, in that it wastes a lot of disk space with trailing pads. In general, if your fields are either all small, or if each occurrence of your fields does not show much variation in length from the others, you will elect to use fixed length records. Apart from this, fixed length records are a luxury confined to large mainframes, where large amounts of disk space are available and the simplification of program code is valued much more highly than the cost of disk space.

In a variable length record, one or more of the fields varies in size from one record to the next. The field is always just the right size for the data it has to hold. In order to process this data, we have to invent an extra field, the 'length attribute' for each variable field. The actual length of the field is stored in the length attribute. If we made all the fields variable length in our telephone directory file, we would define the record as:

length of surname	length attribute
surname	main key field
length of initials	length attribute
initials	secondary key field
length of address	length attribute
address	field
length of 'phone no	length attribute
telephone number	field

You will probably recall that BASIC itself uses just this technique for strings (see INPUT# and PRINT#), but treats integers and floating point numbers as fixed length fields. In the main, the disk space available on a BBC Microcomputer is restricted and variable length records will be preferred.

20.2 Serial/Sequential Access

A serial file is stored in chronological order, whereas a sequential file is stored in true key order. Both of these files share the same weakness. To locate a particular record, it is necessary to read records one-by-one, starting at the beginning of the file, until the correct record is located. In a file of 5000 records it may, in the worst case, require 5000 reads to locate one given record. On average, you can locate a record in half that number of reads. Put more generally, to locate a record in a file of N records takes:

N/2 reads (average)
N reads (worst case)

Neither serial, nor sequential, files have any true potential for random access, unless they are so small that they can be contained entirely within memory. Nevertheless, sequential files are still commonly used in commercial systems so long as the information contained in them is not required interactively. In the traditional system, such files are not amended directly. A separate file, known as the 'Transaction File' is created which contains details of all the updates, insertions and deletions required. This file is sorted sequentially (into key order). It can then be matched, record-by-record against the data file, and a new data file produced which incorporates the amendments. The old data file can be scrapped. The new data file becomes the input file for the next batch of amendments. This is called 'Batch Processing'.

20.3 Fixed Length Binary Chop

This technique can be used with fixed length records in sequential order. Suppose that your private telephone directory contains 600 fixed length records (you would have to reduce the field sizes for this) and the file is in sequential (key field) order. That is to say, the record for Brown comes before the record for Smith etc. Suppose you wish to locate the record for Smith. If Smith is the 461st record in the file, then you would have to read 461 records to obtain it by sequential access. You can improve on this considerably, with a technique known as 'binary chop". In your program you locate the record half-way through the file (the 300th record in this case). You examine its key. If the key is less than the key you want, you split the difference again (300 + (600-300/2)) and examine record number 450. Had the key been larger than the key that you wanted, you would have gone to record number (0 + (600-300/2) = 150. This process is repeated until you find the required record, or until you discover that the record is missing. This is exactly the kind of process that you do instinctively when you look up a word in a dictionary. You know that the words are in alphabetical order. You open the dictionary at a page and in a few large jumps you locate roughly the correct area of the dictionary. Now you turn the pages in successively smaller leaps until you locate the word that you want. Throughout the exercise you are comparing the words in the dictionary with the word that you want.

Let us see how to locate Smith by binary chop when we do not know that it is the 461st record:

1) record number = 0 + (600-0)/2 = 300 record number 300 has a key less than Smith therefore key must be between 300 and 600

2) record number = 300 + (600-300)/2 = 450 record number 450 has a key less than Smith therefore key must be between 450 and 600

3) record number = 450 + (600-450)/2 = 525 record number 525 has a key greater than Smith therefore key must be between 450 and 525

4) record number = 450 + (525-450)/2 = 488 (rounded up) record number 488 has a key greater than Smith therefore key must be between 450 and 488

5) record number = 450 + (488-450)/2 = 469 record number 469 has a key greater than Smith therefore key must be between 450 and 469

6) record number = 450 + (469~450)/2 = 460 (rounded up) record number 460 has a key less than Smith therefore key must be between 460 and 469

7) record number = 460 + (469-460)/2 = 465 (rounded up)
record number 465 has a key greater than Smith
therefore key must be between 460 and 465

8) record number = 460 + (465-460)/2 = 463 rounded up)
record number 463 has a key greater than Smith
therefore key must be between 460 and 463

9) record number = 460 + (463-460)/2 = 462 (rounded up)
record number 462 has a key greater than Smith
therefore key must be between 460 and 462

10) record number = 460 + (462-460)/2 = 461
record number 461 - keys match

Thus, in this example, it was possible to locate a given record in only ten reads. In a fixed-length, sequential file of N records, the worst case number of reads required to locate a particular record is given by the expression:

$$\log_2(N + 1)$$

Figure 41 shows the worst case number of accesses for various sizes of file. It will be seen that the binary chop performs better as the number of records increases.

number of records	worst case reads
1	1
2	2
4	3
8	4
16	5
32	6
64	7
128	8
256	9
512	10
1024	11
2048	12
4096	13
8192	14
16384	15
32768	16
65536	17
131072	18
262144	19
524288	20
1048576	21

Figure 41. Binary Chop Worst Cases.

The file must be sorted in key order. This in itself introduces a huge overhead when amending the file, since amendments must keep to the correct order and this can cause a great deal of record shuffling. Consequently the technique, though simple to code, is seldom used directly on files. It does have a use, however, as we shall see later on.

20.4 Variable Length Sequential Access

To read the next record in a fixed length record file, you have only to step PTR# by the length of the record which is a constant. To read the next record in a variable length record file you have to calculate the amount by which to step PTR#. If all four fields in the telephone directory are variable length, then this amount is calculated from:

amount = 4 4- length of surname + length of initials + length of address 4- length of telephone number.

The extra 4 bytes cover the length attributes themselves. This then allows you to read variable length records serially or sequentially.

20.5 Index Files

Consider how you find your way around a book. To locate a particular reference, you probably use the index. You can extend this principle to disk files. The telephone directory could be implemented as one file of key fields (known as the 'index file') the records of which point to the whereabouts on disk of the rest of the data. For the moment we shall say that the surname field and the initials field are made fixed length (surname 20 bytes, initials 8 bytes). The other fields are held as variable length in a separate file. The index file consists of records of:

surname	20 bytes
initial	8 bytes
ptr	4 bytes

ptr contains the value of PTR# which will locate the start of the rest of the record in the separate data file.

Thus a typical index file could have entries such as:

```
surname             initial value of PTR# for
                            rest of data

ADAMS               C.J.    0
ARNOLD              G.      156
BRAITHWAITE-SMYTHE  O.X.V.  298
```

etc.

The data file itself, consists of records of:

length of address
address
length of telephone number
telephone number

Note that the index file is kept in strict key order, but there is no need to keep the data file in order. We can insert new data file entries at the end of the file, but we must put the index record in the correct place, including the pointer to the correct data record.

To access the file sequentially, you simply access the index file sequentially, picking up the rest of the data from the appropriate part of the data file using ptr. This type of access is known as 'sequential through the index'.

To access all the data for one person, you first have to find the correct index entry. You then use the value of ptr, to locate the address and telephone number in the separate file. But how do you find the correct index? There are several possible techniques.

Firstly, because the overall size of the index file is very much less than the total size of the original file, it may be possible to read the entire index file into memory at the start of the program. Finding the correct index is then extremely fast and can be made even quicker by using the binary chop technique on memory. Moreover, handling insertions and deletions is made fast also, simply because all the data is in memory and no disk accesses are required. However, this method has a serious drawback. Because you are not writing the indices back to disk until the end of the run, the data file and disk copy of the index file are inconsistent during the run. Should you suffer power-loss during a run, you will have lost all your data. There are several ways around this. The first and most obvious is to take a back-up copy of the disk before you start. A power failure would still lose you all your amendments, however. The second possibility is to duplicate the key fields in the data file. Now you can write a special program to re-create the index file from scratch, though, of course, at the expense of the wasted disk space caused by storing key fields twice. You can mitigate against this to some extent by holding the duplicated key fields as variable length fields within the data file.

If the index file is too big to hold in memory, you can go to the other extreme and use the binary chop technique on disk to access each index record as required. Note that all insertions and deletions will necessitate that you re-shuffle the index file on disk. The user may have to wait an eternity whilst this takes place. You do have the advantage that the index file and data file are kept consistent with each other. An alternative is to create a separate index file known as the 'foot index'. All newly inserted records are placed in this foot index file, rather than into the main index file. Deleted index records are flagged and not actually deleted at run time. A separate routine physically deletes flagged index records and sorts the foot index back into the main index at the end of the run. This solves the problem of inconsistency quite nicely. However, any routine that locates a particular index record (including the update routine, itself) must check the foot index if the record cannot be found in the main index file. This can affect response times badly. You need to run the sort fairly frequently to minimise this.

Often a compromise between these two extremes proves valuable. Suppose that you have enough memory to hold one half of the index file. All the time that the user is accessing records in the correct half, the program resembles the first technique. As soon as a record is requested which lies in the other half of the index file, the first half is written back to disk, and the second half is read into memory.

Whichever technique you employ, there are additional problems that require your attention. Deletions are easily handled. You simply delete the index record. Over a period of time, however, the data still on the data file which represents deleted data can consume significant amounts of disk space. You need a tidy up routine to free this disk space, rather analagous to *COMPACT. You also need to keep track of the free disk space, so that you can insert new records there. Moreover, you will have some problems with updated records, if the new records are larger than the originals. Usually you just move the updated record to the free space area and adjust ptr accordingly. Note that this leaves the space occupied by the original record as unusable and your tidy up routine must deal with this also. You can minimise this affect to some extent by assigning slightly more disk space to each record than it actually requires (say 10% more). If you do this, you are in effect electing to waste disk space in small amounts for each record, in the hope of avoiding the need to waste large amounts of disk space when updating records.

If by now you are forming the distinct impression that access methods are a hotch-potch of problems and compromise, then you are correct. And we have only just started too! If you can persevere to the end of this chapter, the summary will help to put things into perspective for you.

20.6 Partial Index Files

If you know much about BASIC, another possibility will probably already have occurred to you. You can use the same technique that BASIC uses to store and retrieve its variables. In this technique, the index file consists of the first letter of the alphabet only. The index record for 'A' points to the first data record that starts with 'A'. The index record for 'B' points to the first data record that starts with 'B' and so on. The data record contains all eight fields, four length attributes plus the surname, initials, address and telephone number. You may if you wish omit the first letter of the surname, since this is implied by the index record. There is also an additional field in each record which points to the start of the next record in sequence that begins with this letter. This is known as an 'embedded pointer', and the records strung together in this fashion are known as an 'embedded pointer chain'. The last record in the chain has an embedded pointer of zero to indicate end of chain.

Let's see an example. Suppose we have three people in the telephone directory whose surnames begin with the letter 'A'. Suppose also that the index record for letter 'A' tells us that records starting with 'A' begin at byte 56. Then the data file may well look something like this:

```
byte  56 = 1350          (2 byte pointer to next in chain)
byte  58 = 5             (length of surname)
byte  59 = "ADAMS"       (surname)
byte  64 = 4             (Length of initials)
byte  65 = "C.J."        (initials)
byte  69 = 12            (length of address)
byte   70 = "15, HILLSIDE" ' (address)
byte  82 = 11            (length of telephone number)
byte  83 = "01-999-9999" (telephone number)
byte 1350 = 1460           (2 byte pointer to next in chain)
byte 1352 = 7              (length of surname)
byte 1353 = "ARNOLD"       (surname)
byte 1360 = 2              (Length of initials)
byte 1360 = "G."           (initials)
byte 1362 = 11             (length of address)
byte 1363 = "9, ASH ROAD"  (address)
byte 1374 = 11             (length of telephone number)
byte 1375 = "01-998-8888"  (telephone number)

byte 1460 = 0000           (2 byte end of chain pointer)
byte 1462 = 9               (length of  surname)
byte 1463 = "ARPINGTON"     (surname)
byte 1472 = 2               (length of  initials)
byte 1473 = "H."            (initials)
byte 1475 = 11              (length of  address)
byte 1476 = "1, ASH ROAD"   (address)
byte 1487 =11               (length of  telephone number)
byte 1488 = "01-998-7777"   (telephone  number)
```

The physical order of the records in the data file is of no importance. So long as the embedded pointers form a chain which is in key order, the technique works. In this technique, the index file is only 26 records long and will not need to be updated frequently. You would probably choose to hold it in memory and write it back to disk if it changed. Note that it can only change if you:

move the first record in a chain

delete the first record in a chain

insert a new first record in a chain

The technique is a combination of random access and pseudo-sequential access. To locate a record you examine the appropriate index record and then start reading records in the chain in sequence, using the embedded pointers, until you find the one that you want.

Note also that to delete a record you simply make the embedded pointer of the record before the deleted record point to the record in the chain which follows the deleted record. For example, to delete the second record in the 'A' chain above, you simply write 1460 at byte 56. However, as earlier, this leaves you with wasted space in the middle of the data file and a tidy up routine is a desirable feature. To insert a new record, you can place a record anywhere in free space and break the pointer chain at the appropriate place, to include the new record. But if this technique scores well on ease of coding, how well does it perform? The answer obviously depends on the number of records in the chain. If you want to access a record for 'ZORRO', you will probably find the technique lightning fast, because surnames beginning with 'Z' are rare. In other instances, when chains become very large; access is woefully slow. Commercial programs that are based on this theme often try to offset the problem by reading more than one record at a time into a large memory buffer. For example, you could decide to read a whole track at a time. In this way, you hope to find the whole chain in memory, which you can then process quickly. If needs be, another track may have to be read. Clearly if you are intending to use this technique, the tidy up routine is crucial. If, over time, your file becomes completely higgledy-piggledy, then the chances of finding the rest of the chain in your buffer are remote.

You do need to keep a record of free space with this technique. This is often accomplished by having a 27th index record which simply points to the next free byte on the disk.

20.7 Variations on Partial Index Files

In the case of a telephone directory, the previous technique is unlikely to be very good from a performance point of view, because some of the chains will be very long. You can extend the idea by creating index records for two letters (for example 'AA', 'AB', 'AC' ....'ZX', 'ZY', 'ZZ'). You now have 26*26 index records (676 records) which you may well be able to hold in memory. Now, however, the chains should be much shorter, allowing much better performance. It is true that the 'SM' chain (with all its Smiths) is likely to be a good deal longer than the 'ZZ' chain. This highlights one problem with the technique - many of the index records will be wasted as there will be no corresponding chains. Each wasted index is not only a drain on valuable disk space, but also on memory.

20.8 Index Range Files

Yet another variation is to use ranges in the index file. For example:

```
"AARON"   to   "ARKWRIGHT"   chain   pointer
"ASKEW"   to   "BOLLINGER"   chain   pointer
"BUNYARD" to "CRUFF"          chain pointer
etc.
```

An extension to this theme appears in hierarchical index files. Broad ranges appear in the top level index, each record of which points to a second level index which further refines the range. Successive index levels gradually narrow the range still further until the bottom level index. The bottom level index either contains individual keys, or is itself a range pointing to a chain of data records that lie within that range. The number of levels used in a hierarchical index is a design feature determined by the application. The example below shows a 3 level index.

```
INDEX 1st LEVEL

   ------------------------------
   Surname     surname    PTR# to
   start of    end of     2nd level
   range       range      index
   ------------------------------
   ABRAHAMS    LORD       4000
   MAKEPEACE   ROGERS     6214
   RUTLEDGE    ZIMBOBO    7345
   ------------------------------

   INDEX 2nd LEVEL

   ------------------------------
   surname     surname    PTR# to
   start of    end of     3rd level
   range       range      index
   ------------------------------
   ABRAHAMS    FINLAY     8976
   FORD        HARRIS     9763
   JORDAN      LORD       10345
   ------------------------------
```

```
       INDEX 3rd LEVEL

       ---------------------------
Disk   surname   surname   PTR# to
addr.  start of  end of    data
       range     range     chain
       ----------------------------

8976   ABRAHAMS  ARKWRIGHT 12654
       ASTON     BARLOW    14321
       BRIDGES   CARLESS   17543
       CLOUGH    DUNFORD   18065
       EVANS     FINLAY    20123
```

The data records for ABRAHAMS to ARKWRIGHT then form one embedded pointer chain. Another chain exists for ASTON to BARLOW and so on.

20.9 Alternate Access

The use of index files is not confined to the main record keys. For example, if a telephone company computerised its telephone directory, there would also be a requirement to be able to access the file using telephone number as a key. This requirement arises in the maintenance department. Having established that a circuit is faulty, they need to know the address of the subscriber so that they can repair it. Thus it is possible to have more than one index file for a given data file, each giving a different way to process the file.

In yet another type of application, separate index files are used to string together records of a similar type. Suppose, for example, that the telephone directory data file was extended to include the type of telephone equipment installed at th£ subscriber's premises. It may be a requirement of the system to process all telephones of a given type. This can be done by separate index files and the technique is known as 'Inverted Index'. An inverted idex file contains several records for the same key field, each record pointing to the next occurrence on the data file. For example:

phone type	PTR#
type 1	0543
type 1	1068
type 1	1096
type 1	1345
type 1	1654
type 2	0345
type 2	0678
type 2	0765
type 2	0867
type 3	0987
type 3	1432
type 3	1700

Equally, index inversion can be achieved by chaining together all the telephones of one type using embedded pointers. As with index files, you are not restricted to just one chain. You can invent chains for specific purposes.

You must always bear in mind that the time taken to update a record is dependent on the complexity of your index file/ chain structures. The more complex structures accelerate the location of a given record but retard its amendment.

20.10 Hashing Algorithms

A 'hashing algorithm' is a mathematical routine which, when applied to a key field, computes the disk address for that record. We have already seen one example of this in the Stamp Collection file in Chapter 14. We saw that we could locate any given record from the algorithm:

record size * (key field - 1)

This type of access is known as 'Direct Access' rather than 'Random Access' because the hashing algorithm leads directly to the required disk address.

In practice, hashing algorithms are a good deal more complicated than this. There are many algorithms that can be used. Some work best with numeric keys; others favour ASCII keys. It should be stressed that an ASCII key is treated as a number for the purposes of hashing. Let us take an example of a hashing algorithm. This example requires that the data be held in fixed length records and that we estimate the maximum size of the file. Suppose we have a file with a numeric key, three bytes long. The key fields can thus be in the range 000 to 999. However, suppose that we do not expect that all the allowable key values will exist. We expect only 200 records at maximum, say. The hashing algorithm to be used is known as the 'modulo hash'. In this algorithm, you simply divide the key number by the maximum number of records, taking the remainder as the record number at which to locate the record. In our example, to locate the record with a key of 347:

```
347 MOD 200 = 147
```

Therefore, this record will be the 147th record in the file. Knowing the fixed record length, we can locate that record directly by computing the value of PTR# to the start of that record. In the general case, a record with a key of K can be located in a file with total number of records T by computing the value:

```
K MOD T
```

What a hashing algorithm attempts to do is to generate a random number in the range 0 to T based on the number K. It does this by scrambling the number K in some way; making a complete hash of it, whence comes the name of the technique.

Of course, no algorithm can be absolutely perfect. Inevitably two records with different keys will hash to the same address. The records are said to have 'collided' when this happens. There are several ways of handling collision. Probably the most effective (and most popular) is to provide a special overflow area on the disk into which to insert records that have collided. It is necessary to include some extra bytes in each data record. One of these bytes determines whether or not a collision of keys has occurred. If it has, the record at the hashed address will not contain any data. Instead it will point to the start of a chain in the overflow area of all those records whose key fields hash to the same address.

The invention of hashing algorithms is a science in its own right with firm foundations in mathematics and published algorithms are many and varied. Clearly, the less collisions that occur, the better the performance. You may wish to experiment with your own hashing algorithms, or read a book on the subject. You will find that the following hints will help you to reduce collisions:

avoid techniques that use addition or multiplication.

add 40% to the maximum number of expected records.

make the maximum number a prime number.

Thus the modulo algorithm would work better if we added 40% to 200 giving 280. If we now look for the next prime number higher than 280 (281) and use this as our value for T, we will minimise the number of collisions.

It should be clear that any chosen algorithm should compute the hash address fast, because it is used to locate each record. An algorithm that takes five seconds to run imposes an overhead of that time for every random read or write.

Because of the need for fixed record lengths, hashing algorithms are frequently used on index files rather than data files, in microcomputers that have limited amounts of disk space. In this way the main part of the data may be held as variable length.

20.11 Implementation Features

Clearly there exists a wide choice of techniques with which you may access a file randomly. On top of this, there are a number of implementation features that you need to consider.

If you are using index files, will they be located on the same side of the diskette as the data files, or possibly on a completely separate diskette?

Will you allow the data file to extend over more than one diskette?

Are you going to use big buffers and read in more than the required amount of data at one time?

Answers to questions like this help to determine the overall performance and scope of your system. They also determine the amount of work that is involved. For example, big buffers are not an option with the DFS. If you desire them, you can abandon all hope of using OPENOUT, INPUT# and PRINT# etc. You are really down to the OSWORD level, controlling all input/output at the sector level.

20.12 Access Method Summary

The art of access methods is that of compromise between good performance and ease of coding. Techniques are legion. However, there are only three fundamental techniques by which you can achieve random access:

hashing algorithm

index file

embedded pointer

You may permutate each of these techniques at will, but it still boils down to these three. These techniques are also the foundation stones for databases (see next chapter).

Note that questions such as 'Which technique is the best?' are meaningless. Some techniques suit some applications best, but are positively rotten for others. Each application needs to be assessed on its own merits. Each application will vary from another in terms of:

size of key type of key size of
record fixed/variable record
number of records amount of
disk available

20.13 Commercial Packages

A wide range of access method software exists for the BBC Microcomputer. This greatly simplifies the task for you. However, your problem now is to ensure that you get value for money, as some of the packages will be better than others.

The first rule is to treat all claims in advertisements with suspicion. Be wary of claims such as:

> maximum number of records 6000
> maximum record size 6000

Your own commonsense should tell you that 36,000,000 bytes of disk data is not possible with an ordinary BBC Microcomputer. The claim means (but does not say) that you can have 6000 very small records or a small number of records, each up to 6000 bytes long. It is generally a mistake to AND together manufacturer's claims. Use OR; it is much more likely to be true. As a general rule you should try to find out as much as possible about the design features of access method software, before you part with your money. Knowing how the package is designed should give you a qualitative feel for how well it will perform with your application(s).

If you intend to write your own access method software - good luck to you! There are few more challenging, or more rewarding ventures in Computing. Hopefully this chapter will have sparked off some ideas in your mind and highlighted some difficulties which have to be overcome. The rest is up to you. Your only limitation will be your own skill and ingenuity. One last word of advice seems appropriate, however. Do not be in too much of a rush to code your software. This is an area where you need to solve all the problems before you start.

21 Database

21.1 Entities and Relationships

In a typical computer application, more than one file may be involved. For example, an on-line invoicing system might randomly access separate Customer and Product files and produce a new file, the Invoice file. You may well have several files yourself. You might decide to keep more extensive data on your friends and relatives, such as their birthdays, the names of their children and their birthdays also. Suppose you have already designed and written the telephone directory system and you decide to invent a new file, the Birthdays file, to hold the extra information. You now find that with your new requirement, you will have to enter all those surnames again.

Holding the same data twice is known as 'data redundancy'. The obvious problem of wasted disk space is probably the least important. If one of your lady friends marries, you will have to change her surname twice, once in the original Telephone file and once in the Birthdays File. You may decide to get address information from the existing Telephone file by using the surname common to both files as a link between them. But are they really common? Whenever the same data is entered more than once, their exists a real opportunity for discrepancy. You may enter your lady friend's new name as Brown in one file, but Browne in the other. It might even be Briwn, due to miskeying the data.

What we are seeing is a case where two files contain data that is related. Where a database differs from a conventional file is that it is specifically designed to handle relationships between one piece of data and another. Of course, you could decide to simply extend your Telephone file; to incorporate the extra birthday data fields. Note though that you still have a relationship problem, the relationship between parents and their children.

Successful database design requires a new process, 'Data Analysis'. Data Analysis is simply a logical (or even conceptual) analysis of the data in order to unearth all the buried relationships prior to the physical design of the data structure. The physical design concerns itself with such mailers as hashing algorithms, index files and embedded pointer chains. The logical design of data is predominantly concerned with relationships and is a necessary precursor to the physical design.

Data may be inter-related in one of three ways. In the examples to be given, all the relationships are human relationships, but database relationships are not confined to these of course.

Simple relationships are known as one-to-one relationships. In a monogamous marriage, the relationship between a husband and a wife is one-to-one. Each husband has one wife and each wife has one husband. In this example, husband and wife are both 'entities'. A database is just a collection of entities and their inter-relationships. The outcome of logical analysis of data is normally depicted in a diagram. Each entity is written in a box, and the related boxes are joined by a line (often accompanied by a description of the nature of the relationship). Thus our simple one-to-one relationship could be represented diagramatically as:

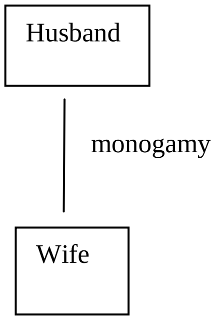


Figure 42. One-to-one relationship.

In some arab countries, the marriage relationship is one-to-many. A husband may have many wives, but a wife may only have one husband. The diagram is drawn similarly, but an arrow is included at the 'many-end' of the relationship.

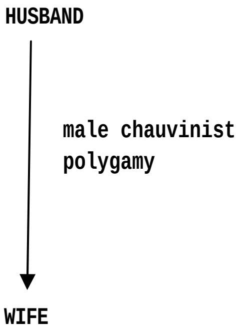


Figure 43. One-to-many relationship.

The most complex relationship is a many-to-many relationship, such as that which exists between parents and children. Each parent may have more than one child and each child may have more than one parent. The relationship is drawn with an arrow at both ends.

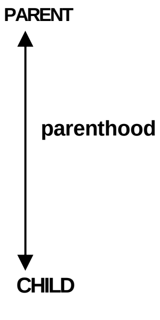


Figure 44. Many-to-many relationship.

Many-to-many relationships are extremely common. The human brain is quite capable of coping with them. Unfortunately/ they are just about impossible to implement directly in a database. Normally an extra entity is invented and through it, the many-to-many relationship is divided into two one-to-many relationships.

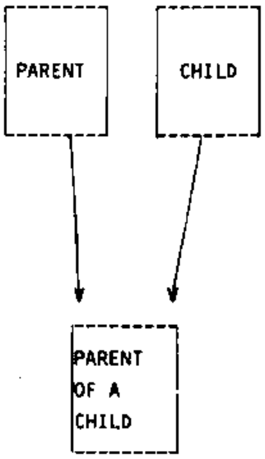


Figure 45. Many-to-many resolved.

Diagrams such as the examples above are called 'Bachman Diagrams' after their inventor. The entities in the diagrams are much more than labels in a box. An entity is a person, place, thing or event. It is roughly equivalent to a record in a conventional file. It has key fields for example, just like a record. The key field of 'Parent' is likely to be the name of the parent. Associated data fields, such as age and sex may also exist. An entity is thus a set of data, uniquely identified by one or more key-fields. The key fields of the 'parent of a child' entity in Figure 45 are parent name / child name. It represents a single event of parenthood and simplifies the relationship between parent and child.

21.2 Normalised Data

In the jargon of the data analyst, the object of Data Analysis is to end up with a fully 'normalised' logical model of the data in the system. What this means is that all the entities must be correctly identified and their keys established. The remaining data fields must be allocated to their correct entities and the relationships between entities must all be exposed. Achieving this is much harder than it sounds. There is a great temptation to concentrate on the application in hand and to bend the logical diagram towards this. The normal consequence of doing this is to design an inflexible database, incapable of supporting future (as yet unknown) applications. Having said that, it has to be admitted that some databases are more susceptible to good logical design than others. Where the data in a database is predominantly tangible (people, places and things), good data analysis is easier. Intangible entities (events) such as Orders, Sales and Purchases are much more difficult to analyse. This is because intangible data only arises out of one or more applications. It is thus much harder to separate Data Analysis and System Analysis from one another. When two people analyse the same data, they frequently produce different results which suggests that there is probably more art than science in Data Analysis. In this book, we shall try to explain the process of Data Analysis as a series of steps. A methodical approach such as this is less likely to lead to nonsensical results.

21.3 The Example

For an example, we shall return to the stamp collection described in Chapter 14. However, the scope of the system will be enlarged to include details of when and from whom each stamp was bought. It will also include the country of origin of the stamp and whether or not the stamp has been franked. We will dispense with stamp number and use stamp name only.

21.4 First Step

The first step is to write down a list of all the fields in your system. This should not prove too difficult. In the example we have chosen you may well end up with a list as follows:

```
stamp-name
country
frank-indicator
value
date-of-vaLuation
supplier-name
date-bought
price-paid
```

21.5 Second Step

The second step is to make a first stab at identifying the entities in your data.

Are there any people ? The answer is yes - the Supplier.

Are there any places ? No.

Are there any things ? Yes - stamps.

Are there any events ? Yes - the purchase of a stamp is an event, as is its valuation.

In a simple example like this you would probably expect to identify all the entities without much difficulty. You will see that it is not too important if you miss a few at this stage, as the method is self-correcting.

Let's write down the entities that we have discovered:

```
Stamp
Supplier
Purchase
Valuation
```

21.6 Third Step

In the third step we decide key fields for each of the entities discovered in the previous step. Effectively, we are asking the question, 'What makes each occurrence of the entity unique from all other occurrences ?'

Assuming that you never buy the same stamp twice, each stamp has a unique name and stamp-name is the key field.

Similarly each supplier has a unique name and supplier-name is the key field.

When you purchase a stamp, you purchase a particular stamp from a particular supplier on a particular date. Thus these are the three key-fields, the fields which together make each purchase unique from all others.

When you value a stamp, you do so for a particular stamp on a particular date and these are therefore the key-fields.

Let's list the entities and key-fields:

```
Stamp Entity
------------
Key fields  : stamp-name

Supplier Entity
---------------
Key fields  : supplier-name

Purchase Entity
---------------
Key fields  : stamp-name/supplier-name/date-bought

Valuation Entity
----------------
Key fields  : stamp-name/date-of-valuation
```

21.7 Fourth Step

In this step you examine the remaining fields in your original list. You should find that they are all 'Other Fields' and have no difficulty in allocating them to their correct entities. You are asking the question, 'About which entity does this field give me some extra information ?' If you come across a field which does not belong in any of the entities in your model, then you have probably missed an entity in step 2. You should identify the entity and go back to step 2.

In our example, we only have four fields left over:

```
country
frank-indicator
price-paid value
```

The first two add extra information about a stamp and so belong with the stamp entity. The price-paid belongs with the purchase entity. The value belongs with the valuation entity. We can now list all the data in our logical model as follows:

```
Stamp Entity

Key fields  : stamp-name
Other fields : country, frank-indicator


Supplier Entity

Key fields  : supplier-name


Purchase Entity

Key fields  : stamp-name/supplier-name/date-bought
Other fields : price-paid


Valuation Entity

Key fields  : stamp-name/date-of-valuation
Other fields : value
```

21.8 Fifth Step

You should now draw a Bachman diagram of your entity model. You are effectively defining the relationships between each entity. If you have a many-to-many relationship, you should split it into two one-to-many relationships at this stage. To discover a relationship, you frequently need to do no more than examine the key-fields of each entity. Two entities which share one or more key-fields must be related. It is a good idea to include the key fields in your entity diagram.

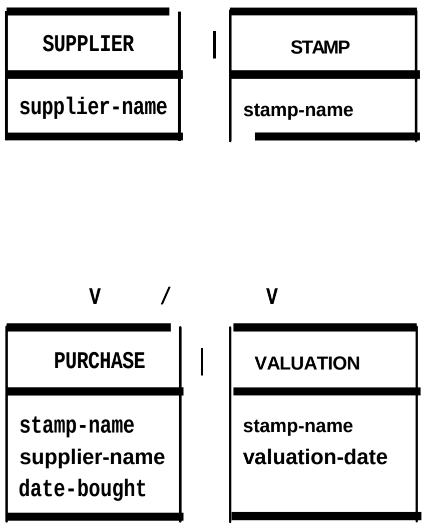


Figure 46. The Logical Model.

21.9 Sixth Step

Why did we bother to analyse our data in the first place ? What have we achieved ? We have in fact achieved a great deal. The logical model drawn above is by no means the only model possible, but hopefully it is the best. By divorcing Data Analysis from the applications to be written, we have produced a logical model of the data which reflects reality rather than purpose. If all has gone well, this logical model should support all of the known applications, together with all future (as yet unknown) applications. In practice, life is never quite that Utopian. There will always be problems, not least because Data Analysis relies as much on judgment as on method. It is probably true to say, however, that the chances of going astray are greatly increased if you try to sidestep the Data Analysis steps.

In the sixth step, you test your model against all the known applications: to see if your model supports them all. It should do. If it does not, then your judgement is probably suspect. Knowing the area(s) of difficulty, you should go back and repeat all the Data Analysis steps.

21.10 Physical Design

The logical data model is the corner-stone on which the database is physically built. Your progress from here depends mainly on whether or not you own a general purpose database package. Database packages are frequently known as DBMS (Database Management System). If you own one of these, you simply define your logical model to the program and it» handles the physical implementation for you. You may well be presented with some choices by the DBMS, however, and your selections will influence the overall responsiveness of your applications.

If you do not own a DBMS package, then you will have to write the database software for yourself. As with simpler files, your basic building blocks will be hashing algorithms, index files and embedded pointer chains. Many databases hash to the top-level entities (Supplier and Stamp in our example) and use embedded pointer chains for the lower-level entities (Purchase and Valuation in our example). Essentially there are only two ways to access a top-level entity. You may either use a hashing algorithm or an index file. For the lower-level entities your choice is between an embedded pointer chain or an index file.

It is possible that you have several database applications in mind and that you would prefer to write your own DBMS software. Read the reviews of existing, commercial packages for ideas on how to implement a DBMS. The following section should help you whether you are in the market for a DBMS or want to write your own.

21.11 DBMS Packages

DBMS packages consist of several different components. The following is a check-list. Most packages will not include all the features in the check-list.

File Definition. This software allows you to define the logical model of your data at the keyboard. It then translates this into a physical design.

File Handling. This software provides you with the capability of randomly or sequentially accessing the data.

Screen Definition. This software allows you to define the screen layout and screen sequences for updating your database.

Query Language. This software allows you to produce reports from your data, by entering a few, simple, very high-level commands. The software should allow you to specify report titles, headers and so forth. It should allow you to report on screen or at the printer.

Sorting. This software allows you to sort your data into a given order, possibly as part of the Query Language, but possibly also to tidy up the file.

Tidy Up Routine. This software frees up wasted space that may be left behind after record deletion for example.

Database Recovery. This software allows you to recover your database to some known, consistent point following an untimely power failure for example.

Database Re-vamp. This software allows you to modify the data structure (add an extra field say) and automatically re-vamps the database to allow for this change.

Data Vetting. This software allows you to specify elementary vetting checks for each field. This can be a range check, a list of valid codes, or a date check. The DBMS update software should then refuse to accept any data that you enter at the keyboard which disobeys your vetting rules.

On top of this check-list, each DBMS will specify certain operating constraints inherent in the design. These include:

> Disk based or cassette based.
>
> Single or multiple entity. Many so called DBMS packages are not DBMS packages at all. They are simple indexed sequential access methods that support only one record type.
>
> Maximum database size in bytes.
>
> Maximum entity size in bytes.
>
> Maximum number of key fields per entity.
>
> Maximum size of each key field.
>
> Alternate indices allowed? - yes/no.
>
> Partial Key searches allowed? - yes/no.
>
> Can data be sorted in both ascending and descending order - yes/no?
>
> Can the data be accessed from within a BASIC program or are you restricted to a query language?
>
> How many conditions can be nested together in a single query language search?
>
> Can conditions be both ANDed and ORed?

Lastly you will want to consider the performance of the DBMS. How quickly can you access a single, specified entity? How long do reports take? For this you need to read reviews in magazines. Generally the reviews will also give some indication of the design of the DBMS and you should be able to deduce those techniques which lead to acceptable performance. If you want to buy a DBMS, look to the reviews also for an indication of ease of use, documentation etc.

22 Sorting

The need to sort a disk file is an everyday occurrence. Sometimes you want to convert a serial file into a sequential file. Sometimes you just want your data in a different order so that you can produce a different report. Above all, the sort is a component part of many database systems. This chapter is confined to sort techniques. There is a problem inasmuch as there are so many different filing methods that it is impossible to demonstrate the various sort techniques for all disk files. Thus throughout this Chapter it is assumed that your data is in a memory array, A$, which has A% elements within it. In this way each sort technique can be explained very simply, with demonstration code. The application of the technique to your own particular disk files should not prove difficult, but this is left to you.

22.1 Bubble Sort

This is one of the commonest techniques used in computer programs, though its performance is poor for large arrays. It is a simple technique to understand, and for this reason is a good one to start with. Suppose that we have nine numbers that we wish to sort into numeric sequence:

```
1
5
2
6
9
8
3
4
7
```

A bubble sort achieves its end by going through these numbers in several passes. In the first pass it compares each number in turn with its neighbour. If they are in the wrong order, it swops them; otherwise it leaves them alone. So starting at the beginning:

Compare 1 and 5. 1 is less than 5 so leave well alone.

Compare 5 and 2. 5 is greater than 2 so swop them.

Compare 5 and *6*. 5 is less than 6 so leave well alone.

Compare 6 and 9. 6 is less than 9 so leave well alone.

Compare 9 and 8. 9 is greater than 8 so swop them.

Compare 9 and 3. 9 is greater than 3 so swop them.

Compare 9 and 4. 9 is greater than 4 so swop them.

Compare 9 and 7. 9 is greater than 7 so swop them.

So, at the end of the first pass, the new sequence of the numbers is:

```
1
2
5
6
8
3
4
7
9
```

The process is repeated with another pass and then another, until eventually a complete pass through all the numbers leads to no swopping of numbers. When this happens, the numbers are in sequence.

Clearly the time taken by this sort technique depends on how well ordered the data was in the first place. As a rule ef thumb though, the number of comparisons that will on average have to be made depends on the square of the number of elements in the array. Thus the bubble sort performs satisfactorily for a small array, but gets dramatically worse as the array increases in size.

Let us now see how we can code the bubble sort for our array, A$, with A% elements in the array. There are just two points to be made about this code.

Firstly, in order to swop two numbers we need a temporary holding area for one of these numbers (temp$). Secondly, at the start of each pass we set to 1 an indicator (fin%). If we have to swop two numbers, fin% is zeroised. At the end of the pass, if fin% is still set to 1, then we swopped no numbers and so the sort must be finished.

```
10 REM
20 REM Bubble Sort of A$ array with A% elements
30 REM –
40 REPEAT
50 fin% = 1
60 FOR I% = 1 TO A%-1
70    IF A$(I%) > A$(I%+1) PROCswop
80 NEXT
90 UNTIL fin% = 1
100 END
110
120 DEF PROCswop
130 fin% = 0
140 temp$= A$(I%)
150 A$(I%) = A$(I%+1)
160 A$(I%+1) = temp$
170 ENDPROC
```

22.2 Shell Sort

This sort, named after its inventor, Donald Shell, is a modification of the bubble sort. In fact Shell described a whole range of sort algorithms, each slightly different from the other, and each with optimal performance characteristics for different types of arrays. We have already stated that the basic bubble sort performs well for small arrays, but poorly for large arrays. To put that statement into perspective, the program listed above will take several hours to sort just 5000 elements.

Once again, the easiest way to explain a Shell sort is by example. Consider the nine numbers in the first example:

1
5
2
6

9
8
3
4
7

Instead of comparing each number in the array with its neighbour, we shall compare each with the number three places away. This is called a 3-sort.

Compare 1 and 6. 1 is less than 6 so leave well alone.

Compare 5 and 9. 5 is less than 9 so leave well alone.

Compare 2 and 8. 2 is less than 8 so leave well alone.

Compare 6 and 3. 6 is greater than 3 so swop them.

Compare 9 and 4. 9 is greater than 4 so swop them.

Compare 8 and 7. 8 is greater than 7 so swop them.

At the end of the first pass of the 3-sort, the numbers are in the following sequence. No further passes at the 3-sort level change the sequence of numbers.

1
5
2
3
4
7
6
9
8

The whole process is now repeated, but this time each number in the

array is compared with the one two places away (a 2-sort).

Compare 1 and 2. 1 is less than 2 so leave well alone.

Compare 5 and 3. 5 is greater than 3 so swop them.

Compare 2 and 4. 2 is less than 4 so leave well alone.

Compare 5 and 7. 5 is less than 7 so leave well alone.

Compare 4 and 6. 4 is less than 6 so leave well alone.

Compare 7 and 9. 7 is less than 9 so leave well alone.
Compare 6 and 8. 6 is less than 8 so leave well alone.

At the end of the first pass of the 2-sort, the numbers are in the following sequence. No further passes at the 2-sort level change the sequence of numbers.

1
3
2
5
4
7
6
9
8

Finally the array is subjected to an ordinary bubble sort (a 1-sort). It will be seen that in the example above only 1 pass is needed to sort the numbers into sequence. In fact, to be strictly accurate, two passes at each of the 3-sort, 2-sort and 1-sort levels were needed. The second pass in each case merely discovered that no more numbers needed to be swopped.

All Shell sorts follow these basic principles. They vary from one another in the intervals over which comparisons are done and in the way that these intervals are calculated. Clearly you would not normally expect a 3-sort, followed by a 2-sort followed by a 1-sort. Equally clearly, the last sort must always be a 1-sort. As a general rule, intervals that consist of powers of 2 do not perform well.

Let's take a simple example. If there were 1000 elements in the array, typical values for the intervals might be 333, 111, 37, 12, 4 and 1. To make the sort routine more general, it uses an algorithm to calculate the intervals, based on the number of elements in the array. It is this algorithm that varies from one Shell sort to the next. In the example program to be listed here, we shall start off at A% DIV 3 and then divide successively by 3 until we reach the number 1. There are better algorithms than this, all rather mathematical, but this technique will produce much better sort times for large arrays than the straightforward bubble sort. In this program, K% represents the interval over which comparisons are to be made.

```
10 REM
20 REM Shell Sort of A$ array with A% elements
30 REM
31 K% = A%
32 REPEAT
33 K% = K% DIV 3
34 IF K% = 0 K% = 1
40 REPEAT
50 fin% = 1
60 FOR I% = 1 TO A% - K%
70    IF A$(I%) > A$(I%+K%) PROCswop
80 NEXT
90 UNTIL fin% = 1
100 UNTIL K% = 1
100 END
110
120 DEF PROCswop
130 fin% = 0
140 temp$ = A$(I%)
150 A$(I%)= A$(I%+K%)
160 A$(H+K%) = temp$
170 ENDPROC
```

Shell sorts perform well for all types of sorting and should normally be your first choice. Most of the published algorithms involve exponentiation in some way or another. An algorithm that works particularly well calculates the interval from the expression:

interval = (3^N-l)/2

Here are some typical intervals derived from this expression:

N = 7 interval = 3280
N = 6 interval = 364
N = 5 interval = 121
N = 4 interval = 40
N = 3 interval = 13
N = 2 interval = 4
N = 1 interval = 1

For the first sort, N is an integer chosen such that the interval is the highest possible integer which does not exceed one third of the number of elements in the array. N is then decremented by 1 each time, until it reaches one eventually. Thus for a 10000 element array, you would start at N = 7, equivalent to an interval of 3280 and then decrement N by 1 each time.

22.3 Hoare Sort

Sometimes known as Quicksort, this technique was invented by C.A.R.Hoare. It performs very well if the array to be sorted is truly random in nature. However, if there is any order existing already in the array, it performs quite badly. There are many variations on the basic idea. We shall confine ourself to the basic idea, by using a similar example as before. Suppose we want to sort the numbers below:

5
1
2
6
9
8
3
4
7

We make a note of the first number in the array, 5. This is called the pivot. We now proceed down the array until we find a number greater than this pivot. In this case it is the number 6. Now we start at the bottom of the array proceeding upwards until we find a number less than the pivot. In the example it is the number 4. So we swop these two numbers, 6 and 4, giving:

```
5
1
2
4
9
8
3
6
7
```

Repeating the previous exercise, we identify 9 and 3 as the next numbers to swop, giving:

```
5
1
2
4
3
8
9
6
7
```

Repeating once again, we note that 8 and 3 are the next numbers that should be swopped, but we also notice that the downward scan has crossed over the upward scan. When this happens, we take the lower of the two swop numbers and swop it with the pivot instead. In this example, we simply swop 5 and 3, and 3 becomes the new pivot.

```
3
1
2
4
5
8
9
6
7
```

Effectively we have located the correct position for the number 5. The whole process now starts again, with 3 as the new pivot. Eventually all the numbers will be sorted into sequence. The following code will Quicksort our A$ array.

```
10 REM
20 REM Quick Sort of A$ array with kX  elements
30 REM
40 REPEAT
50 PROCdown
60 PROCup
70 PROCtest
80 UNTIL J% > A% OR K% = 0
90
300 DEF PROCdown
310 J% = 0
320 REPEAT
330 J% = J% + 1
340 UNTIL J% > A% OR A$(J%) > A$(1)
350 ENDPROC
360
400 DEF PROCup
410 K% = A%+1
420 REPEAT
430 K% = K% - 1
440 UNTIL K% = 0 OR A$(K%) < A$(1)
450 ENDPROC
460
500 DEF PROCtest
510 IF J% > A% ENDPROC
520 IF K% = 0 ENDPROC
530 IF J% < K% PROCswop(J%,K%) : ENDPROC
540 IF A$(J%) < A$(K%) PROCswop(1,J%) ELSE PROCswop(1,K%)
550 ENDPROC
560
600 DEF PROCswop(L%,M%)
610 temp$ = A$(L%)
620 A$(L%)  = A$(M%)
630 A$(M%) = temp$
640 ENDPROC
```

22.4 Sort Summary

When you write sort routines, your natural choice will favour a Shell sort. However, disk I/O is a time consuming affair. You should try to arrange that as far as possible, the data to be sorted is held in memory. For example, suppose that you wish to sort a serial file, with 200 records each having an alphanumeric key 10 bytes long. You should write a program that serially reads all the keys into an array together with the value of PTR# for each record as well. You should then sort this array in memory and use the sorted array to shuffle the records into the correct sequence.

If your file is too big to sort in this way, you should generally regard it as a number of smaller files, sorting each smaller file in the way described above. You then need a merge program to merge the sorted smaller files together. Remember that each of the smaller files will be sorted into sequential order in the first step. A merge program opens each of these smaller files for input and compares the key fields on each file. The record with the lowest key is copied to an output file and the appropriate input file is read again. The key comparison is now repeated. The whole process continues until all the input files have been completely read. At this point, the output file consists of the original input file in sorted sequence.

PART 4
PROBLEMS

23 Acorn DFS Weaknesses

The Acorn DFS has several fundamental design features that detract from its value as a Disk Filing System. Before we discuss these weaknesses in depth, it is only fair to point out that you have to spend very much more on a microcomputer before you get a DFS which works much better.

Each of these weaknesses presents the user with a problem. Most of the problems can be overcome at the expense of convenience, providing you are aware of them at the outset.

23.1 Contiguous Files

Perhaps the worst feature of the DFS is the way in which files and the catalogue are stored on the diskette. The catalogue occupies the first two logical sectors of physical track zero and permits the storage of up to 31 files on that side of the diskette. Each file has to be stored in a single, contiguous extent on the diskette and may not extend to the other side. Suppose that you have two files on a diskette:

filename	starts at	ends at
FILE1	track 0,sector 2	track 3, sector 5
FILE2	track 3,sector 6	track 8, sector 2

If you load FILEl into memory and then increase its size, you can no longer save it back to the diskette as FILEl. The Acorn DFS will try to overwrite the original version of FILEl until it gets to the end of track 3, sector 5. It then displays the 'No room' error message. Clearly there is plenty of room on the diskette, but the DFS works in a rather crude and unhelpful manner. You can circumvent this problem at the time that it happens, by saving the updated version of FILEl with a different name. For example:

```
SAVE "FILE3"
*DELETE FILE1
*RENAME FILE3 FILE1
*COMPACT
```

This sequence of commands will allow you to save the updated file as FILE1, although F1LE2 and FILE1 will now be stored in reverse order on the diskette.

It is much better to prepare in advance for this problem by ensuring that any data or program file, which you expect to increase in size, is the only file on that side of the diskette. You should set aside three diskettes for program development work. A newly written program should be the only file on the diskette. Each of the diskettes is cycled in turn, ensuring that you have adequate back-up should something go wrong. This is known as the 'Grandfather/Father/Son' system. When your program is fully tested and working, you should copy it to a utilities diskette which can contain more than one program. This frees your development diskettes for future development work. Of course, this method assumes that you can afford to buy diskettes in the necessary numbers in the first place. If a tight budget prevents this, you will have to depend on the first method. It is advisable to take a regular back-up copy of any diskette which you expect to *COMPACT. If anything goes wrong during *COMPACT, you are likely to lose all the files on that side of the diskette.

A closely related set of problems occurs with the BASIC command, OPENOUT. The size of a new file is determined when the file is first created by OPENOUT. The following flowchart shows how the file size is determined when you use OPENOUT:

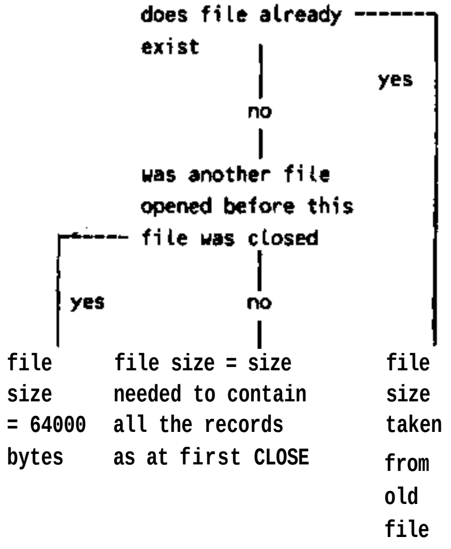

does file already exist
yes
no
was another file opened before this file was closed
yes
no
file size = 64000 bytes
file size = size needed to contain all the records as at first CLOSE
file size taken from old file

Figure 47. File Size Allocation.

23.2 Number Constraints

Because the catalogue is fixed, the maximum number of files permitted on each side of a diskette is restricted to 31. This prevents you from creating a PROCs disk, with each of your commonly used PROCs held as a separate file, ready to be included in future programs. Alas, you have to cluster your PROCs together, so that several appear in one file. Similarly, a diskette of letters that you have written will soon reach the 31 maximum, even though there may be plenty of free space on the diskette. If this is your main application, then I am afraid you have not chosen your microcomputer wisely.

At any one time, the number of files opened must not exceed five, this is because the buffer allocation for files is fixed. Nothing prevents you from closing an opened file temporarily whilst you open another. This effectively gives you a technique for handling more than five files in a program.

Filenames, disk titles etc. are all fixed in length as well.

23.3 Cassette/Disk Problems

How much better it would have been if PAGE had been fixed at say &1300 for all systems. The buffer space needed for open files could easily have been dynamically allocated, rather like having a hidden DIM statement. Instead of this the default values for PAGE (without ECONET) are:

cassette only	PAGE = &0E00
disk	PAGE = &1900

So when you buy your DFS, how do you transfer all those cassette programs onto a diskette? In many cases, there is no simple answer. Let's go through the various problems starting with the simplest.

Suppose you have a single BASIC program on cassette that you wish to transfer to disk. Suppose it is called PROG. Then you simply enter:

```
*TAPE
PAGE = &1900
LOAD "PROG"
*DISK
SAVE "PROG"
```

Note that this technique also works for assembler programs which still have their BASIC appendages, because these are really BASIC programs also.

Disk versions of purely machine-code programs can be created by substituting *LOAD and *SAVE in place of LOAD and SAVE respectively. The technique is rarely sufficient for purely machine-code programs, however, because these are not normally relocatable. You can often, however, *LOAD a machine-code program from disk at &1900 and then shift it to its correct load address. Note that if the correct load address is below &1900 any disk access performed by the machine-code program may not work. The following program *LOADs a machine-code program (size = &1234 bytes) at &1900 and then downloads it to its correct load address (&F00) and executes it at its correct execute address (&1000). To use this program for other machine-code routines, simply substitute the correct values for filename, old load address, new load address, execute address and size. Note that the routine will not function properly if you upload the program in memory, if the old and new addresses overlap. The program is written in assembler so that large programs download quickly. The assembly occurs at &7000 in this example, but you may need to alter this depending on circumstances.

```
10 REM
20 REM DLOAD : program downLoader
30 REM
40 FOR pass% = 0 TO 2 STEP 2
50 P% = &7000
60  [OPT pass%
70  .file    EQUS "PROG"
80           EQUB &D
90  .blok    EQUW file
100 .oldload EQUD &1900
110 .execadd EQUD &1000
120 .size    EQUD &1234
130 .dummy   EQUD 0
140 .newload EQUD &F00
150 .blokad  EQUW blok
160 .move
170  LDX blokad   \ load PROG
180  LDY blokad+1 \ at old
```

```
190    LDA #&FF       \ Load
200    JSR &FFDD      \ address.
210    CLC            \ for add.
220    PHP            \ save carry.
230    LDX #0         \  loop count.
240    .init
250    LDA newload,X   \ save new load
260    STA &80,X      \ in &80 - &83.
270    LDA oldload,X   \ save oldload
280    STA &70,X      \ in &70 - &73.
290    PLP            \ get carry.
300    ADC size,X     \ save end addr
310    STA &7C,X      \ in &7C - &7F.
320    PHP            \ save carry.
330    INX            \ bump count.
340    CPX #4         \ init done?
350    BCC init       \ no - back
360    PLP            \ tidy stack.
400    LDY #0         \ zero count
410    .loop
420    LDA (&70),Y    \ move old
430    STA (&80),Y    \ to new.
440    INC &70        \ bump old.
450    BNE bumpnew    \ if overflow
460    INC &71        \ bump next
470    BNE bumpnew    \ if overflow
480    INC &72        \ bump next
490    BNE bumpnew    \ if overflow
500    INC &73        \ bump next
510    .bumpnew
520    INC &80        \ bump new.
530    BNE test       \ if overflow
540    INC &81        \ bump next
550    BNE test        \ if overflow
560    INC &82        \ bump next
570    BNE test       \ if overflow
580    INC &83        \ bump next
590    .test
600    LDX #3 \ count
610    .tloop
620    LDA &70,X      \ test
630    CMP &7C,X      \ all done ?
640    BNE loop       \ no - back
650    DEX            \ next count
```

```
660   BPL tloop    \ try again
670   LDX #&FF     \ clear
680   TXS          \ the stack.
690   JMP execadd  \ start PROG
700 ]
710  NEXT pass%
720 CALL move
730 END
```

The real problem arises when you wish to make a disk copy of software that consists of more than one program. Once you have downloaded one of these programs into the DFS memory space (&E00 to &1900) you can no longer use the DFS to fetch in the next program in the chain. If the program is a machine-code program, you have no choice but to relocate it at a new load address and this involves re-writing the program. Now you might have two further problems. Perhaps the program is too big to load at a higher load address. In this case, you will have to chop out some of the code to make it squeeze in. Alternatively, perhaps it is a program that you have bought. The copyright laws forbid you to modify other people's software. A good Software House will offer you a reasonable trade-in price for your cassette and supply you with the disk version in its place. I advise you to take this offer. When the Software House offers no such trade-in, or worse still does not even produce a disk version, it is not unknown for users to flaunt the copyright laws and modify the original software. They obtain a disassembled listing of the machine-code and alter the appropriate bytes to make the program locate at the new address. This is a tedious process, involving a fair amount of expertise. Care has to be taken that the new version of the program interfaces correctly with the other programs in the suite. Whereas it might be expected that a task such as this would be time-consuming, but not mind-bending, in fact the security features built into most programs normally increase the complexity of the task out of all proportion. Frequently the program is not executable machine-code at all and only becomes so at run time when it is EORed with some bit pattern. Usually ESCAPE is disabled and BREAK is set to clear the entire program, so it is difficult to trap the code after it has been re-constituted at run time. Normally, RTS instructions can be inserted into strategic parts of the code to allow you to obtain a disassembly. This is often an iterative process, however. It is clear that people do find ways around all these problems. You really need the hacker's mentality to programming, in which the challenge of breaking the security code far outweighs any pleasure derived from the software once it has been broken. If you have a flair for this sort of thing, you are probably breaking into software all the time, in which case you need no help from me and you will probably choose to ignore all cautions regarding the legality of what you are doing. If you do not have the hacker's instincts, then even if I devoted a whole book to the subject of software security, you probably would not manage to crack any but the simplest programs. Remember, it is not just know-how that you need - you also need a great deal of time and patience. Without this, your only hope is to pressure the Software Houses. It is perhaps surprising that the Software Houses have not reproduced some of the old favourite games, ten at a time on a single diskette, for about the price of a single game. Sales of these games must be rather limited by now and there is a clear market on which the Software Houses could capitalise, without any extra development costs normally associated with games. Maybe a barrage of letters from users is just the stimulus needed to trigger such action.

23.4 Performance Problems

DFS 0.90 has no true OSGBPB function and this has to be regarded as a serious weakness. DFS 1.20 does have a true OSGBPB function and is also about three times quicker in handling OSBGET and OSBPUT calls.

Some of the commercial word processor packages available for the BBC Microcomputer are exceedingly slow at loading and saving text files. This is because they examine each individual byte (to handle soft carriage returns, for example) before transferring the byte via OSBGET or OSBPUT. If you look at the DFS 0.90 code for OSBPUT you will see that among the routine tasks performed are:

```
check file handle
check directory
check drive number
check if disk changed
check if disk write protected
check if file locked
check 8271 command register is free
check for end of file
check drive ready
check catalogue for room on the diskette
put byte into buffer
if buffer full, write sector
```

This means that each OSBPUT call incurs a sizeable overhead in code, which is repeated for each and every byte to be written to the diskette. Consequently, saving a large text file can easily take a minute or so, whereas *SAVE can save a file of similar size in just a few seconds.

If, like me, you use a word processor extensively, whether it uses OSBGET/OSBPUT or OSGBPB, you might consider buying DFS 1.20.

23-5 Bugs

DFS 0.90 has a number of bugs. The most important of these are:

Checks for a valid catalogue are not properly made. For example, if you do the following quickly enough, you can end up with the first diskette's catalogue copied onto the second diskette:

```
LOAD "PROG"
then switch diskettes
SAVE "PROG"
```

When a file is extended by increasing the value of PTR# past the end of file, the DFS should pad the file with zeroes from old end of file to new end of file. Instead of this, DFS 0.90 pads from old value of PTR# to new value of PTR# and this can lose you a lot of data. The way to avoid this is to set PTR# to EXT# before going past end of file.

When using CLOSE#0 to close all open files, things can go badly wrong if the open files are on different surfaces. It is possible to end up with the catalogues copied back onto the wrong surfaces.

If you generate a zero length file and then try to copy it, the result can be rather disastrous in DFS 0.90. Avoid zero length files.

24 Alternative DFS

If you wonder why a chapter on alternative DFS software should be included in the Problems Section, then I have to tell you that in my experience, some of these packages are presenting BBC Microcomputer users with some truly nasty headaches. I would also add that the letters columns in various magazines support this contention.

Let's start at the beginning, for those considering a disk upgrade. The most important question that you should ask yourself is, 'Do I need to be able to read or write diskettes compatible with the Acorn DFS?' The answer to this question will be 'yes', for example, if you intend to buy games or other commercial software. If the answer to this question is 'no', then you do not have to worry about DFS compatibility with the Acorn DFS. You are free to choose any of the DFS packages on the market. You are also free to explore quite different disk systems such as CP/M of which there are several offerings. Your choice will be based on the facilities on offer and the answer to the second important question, 'Does my chosen DFS actually work, or is it riddled with bugs?' Consult magazines, particularly the letters page, to answer that one. One last question may arise if you own a Paged Rom Board. You have to ensure that your chosen DFS does not consist of a board which cannot co-habit with your rom board.

If, however, compatibility with the Acorn DFS is important to you, then you have to research your prospective DFS most carefully. The range is large already and increases everyday. Moreover, existing packages appear in new versions regularly. For this reason no attempt is made to list the features of individual packages in this book. To be truly compatible with the Acorn DFS, a DFS must:

> Be able to read and write in single density (FM) format. Double density (MFM) format is completely incompatible, though most double density controllers have an optional single density mode and this is rarely a problem.
>
> Be able to handle all the DFS star commands in the same way as the Acorn DFS.

Be able to support all the DFS OSF1LE, OSARGS, OSBGET, OSBPUT, OSGBPB, OSFIND, OSFSC and OSWORD calls.

Be able to return error statuses in a manner identical to the Acorn DFS.

Use the same memory areas as the Acorn DFS and for the same purposes.

In practice, minor discrepancies with the star commands are seldom of great importance. The most important factor is the support for all the OSWORD calls. This is because most Software Houses use logical sectoring techniques, deleted data marks etc. to protect their products from piracy. This means that their products use low-level OSWORD calls when loading.

Some products also peek and poke memory addresses directly. Deplorable though this practice is, it means that compatibility in memory allocation can be important also.

How will you discover if a particular DFS is truly compatible or not? My experience leads me to believe that virtually all of them have had problems with compatibility at some stage in their histories. It is difficult enough to prove that software works at all. Extensive testing cannot hope to test all the pathways through a complicated program. It is even harder to prove that a piece of software works identically to another. Therefore, you would be wise to regard alternative DFS systems as a risk. The risk is clearly greatest when the DFS is based on a FDC chip other than the 8271. How do you emulate the read deleted data command of the 8271 if the chip has no similar feature? You must remember that the manufacturers of alternative DFS packages are trying to attract you to their extra features. A true Acorn DFS emulation with some of the available FDC chips would require so much code that there would scarcely be room for these extra features. Nevertheless, I am reluctant to name names. The Acorn DFS certainly leaves room for improvement and alternative DFS packages appear with ever increasing version numbers more or less daily. You should simply be aware that you are taking a chance with alternative DFS packages. Only you can balance the risk against the benefits to be derived from the alternate DFS. If incompatibility is likely to distress you, you should at least ask the dealer to demonstrate the DFS with one of the best protected games. Make sure it is the standard version of the game and not just a 'special' that works with the particular DFS. Though not foolproof, this may give you some degree of confidence.

Perhaps though, you already own an alternative DFS. Maybe you believed the sales pitch which promised 'full Acorn compatibility'. Or possibly the shortage of 8271 chips in the UK forced you to seek an alternative. By now you may be one of the many who has bought at least one diskette which will not run with your DFS. The dispute which then ensues may involve no less than five parties.

Firstly, there is the dealer who sold you the software. He disclaims all liability on the grounds that the software was only intended for a standard BBC Microcomputer.

Secondly there is Acorn who clearly has no responsibility for alternative DFS roms.

Thirdly there is the Software House who wrote your software. They disclaim responsibility on the same grounds as the dealer.

Fourthly there is the manufacturer of the DFS rom. He tends to blame the Software House for writing stupid code; he may even accuse them of dodgy practices, such as direct memory poking.

Lastly there is you, the innocent party in all this, quite unable to get any satisfaction. It always seems grossly unfair that it is the innocent party who has to suffer. To redress that imbalance, I would offer the following advice.

For all future purchases of software, you should ask the dealer if the software will run with your DFS. You should not part with your money unless the dealer is prepared to give you a written undertaking that he will refund your money should it fail to work with the specified DFS package.

Next you should test all the low-level calls on your DFS. Appendix 4 provides a check list of these calls and you can use the demonstration programs in this book. Note carefully each discrepancy from the standard Acorn DFS. Repeat this exercise for the star commands. You should then write to the manufacturer of the DFS pointing out each discrepancy. If the manufacturer advertises the DFS as 'Acorn compatible', you should ask for your money back. If he refuses, you might try contacting the magazine that carried the advertisement and your local council's trading standards department.

In addition to alternate DFS packages, there are also alternate diskette packages. One of the popular ones is the 3" diskette. This format has several likeable features. Firstly, the read/write window is protected by a spring-loaded metal plate which slides away when the diskette is inserted into the drive. This, combined with the rigidity of the diskette, makes it much less susceptible to damage: ideal where children are concerned. Secondly, 3" diskettes are double-sided. They can be turned over in a single drive, rather like a gramophone record. The 3" drives can be driven by the standard Acorn DFS and present no compatibility problems under these circumstances. They are also sometimes offered as part of a package with an alternate DFS system and here the caution given before applies.

APPENDICES

Appendix 1. Disk Upgrade

The Acorn DFS disk upgrade requires OS 1.0 or greater. The main components are:

DFS Rom (IC88)

Intel 8271 floppy disk controller (IC78)

In addition there is a disk interface kit which consists of:

2 x 7438 (IC79 and IC80)
2 x 74LS393 (IC81 and IC86)
1 x 74LS10 (IC82)
1 x CD4013B (IC83 and IC84)
1 x CD4020B (IC85)
1 x 74LS123 (IC87)

IC86 is a clock divider, deriving a 31.25 KHz clock signal from the 8 MHz system clock. IC83, IC84 and IC85 detect index pulses from the drive.

Disk drive logic signals are buffered by IC79 and IC80. IC87 conditions the incoming data signal from the drive, ensuring fixed pulse width, which is then fed into IC81 and IC82, the data separator circuitry.

The following steps should be undertaken:

1) Switch off the microcomputer and unplug from the power supply.

2) Remove the lid by unscrewing the four screws marked 'fix'. Two of these are on the rear panel and two on the base.

3) Gently disconnect the keyboard ribbon cable from its connector.

4) Unfix the keyboard sub-assembly by undoing the two (or 3) securing nuts and bolts.

5) Carefully remove the keyboard, disconnecting the speaker lead as this becomes exposed.

6) Examine the printed circuit board to discover the Issue message.

7) If it is Issue 1 or 2, connect the two pads of S8 with a wire link.

8) If it is Issue 1, 2 or 3, carefully cut the leg of pin 9 of IC27. Sever the track between this pin and IC89. Solder a piece of insulated wire between the cut leg of IC27 and the east pad of S9.

9) If it is Issue 4 or greater, remove the connector from S9.

10) Ensure the following links are made:

S18 North
S19 East
S20 North
S21 East-West twice
S22 North
S32 West
S33 West

11) Insert all the chips in the correct holders, ensuring that the notches conform (point towards the rear).

12) Reassemble in reverse order.

13) If the disk drive uses the BBC Microcomputer power supply, attach the power lead to the socket provided.

14) Attach disk drive ribbon cable to the disk port.

Appendix 2.   Disk Cable.

The following signals are present on the disk drive ribbon cable:

pin 1	ground
pin 2	not connected
pin 3	ground
pin 4	not connected
pin 5	ground
pin 6	not connected
pin 7	ground
pin 8	index pulse
pin 9	ground
pin 10	drive select 0
pin 11	ground
pin 12	drive select 1
pin 13	ground
pin 14	not connected
pin 15	ground
pin 16	load head
pin 17	ground
pin 18	direction
pin 19	ground
pin 20	step
pin 21	ground
pin 22	write data
pin 23	ground
pin 24	write enable
pin 25	ground
pin 26	track zero
pin 27	ground
pin 28	write protect
pin 29	ground
pin 30	read data
pin 31	ground
pin 32	side select
pin 33	ground
pin 34	not connected

Appendix 3. Disk Connectors.

Disk Drive PCB Connector.

```
                          TOP

---------------------------------------------------
34 32 30 28 26 24 22 20 18 16 14 12 10  8  6  4  2
---------------------------------------------------
33 31 29 27 25 23 21 19 17 15 13 11  9  7  5  3  1
---------------------------------------------------

                         BOTTOM
```

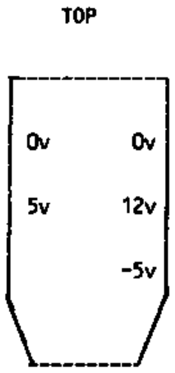
Socket to connect to BBC Port.

```
                          TOP


               keyway --+
                        |
                        |
                        |
                       +--+
-----------------------+  +-------------------------
 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33
----------------------------------------------------
 2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34
----------------------------------------------------

                         BOTTOM
```

Power Socket on BBC Micro.



The +5v connection is 1.25 amps.

The +12v connection is 1.25 amps.

The –5v connection is 75 milliamps.

Appendix 4. DFS Call Summary.

OSFILE

A = 0	Save memory block
A = 1	Update catalogue
A = 2	Update load address
A = 3	Update execute address
A = 4	Update lock status
A = 5	Read catalogue entry
A = 6	Delete file
A = &FF	Load file

OSARGS

A = 0	Y = 0	Read Filing System
A = 1	Y = 0	Read address of *RUN parameters
A = &FF	Y = 0	Update all files
A = 0	Y = handle	Read PTR#
A = 1	Y = handle	Write PTR#
A = 2	Y = handle	Read EXT#
A = &FF	Y = handle	Update a file

OSBGET

Y = handle	Read byte into A

OSBPUT

Y = handle	Write byte from A

OSGBPB

A = 1	Write block at user pointer
A = 2	Write block at PTR#
A = 3	Read block at user pointer
A = 4	Read block at PTR#
A = 5	Read title + boot-up option
A = 6	Read *DIR drive + directory
A = 7	Read *LIB drive + directory
A = 8	Read filenames

OSFIND

```
A = &40 Y = handle OPENIN
A = &80 Y = handle OPENOUT
A = &C0 Y = handle OPENUP
A = 0  Y = handle CLOSE# a file
A = 0  Y = 0    CLOSE# all files
```

OSFSC

```
A = 0 *OPT handler
A = 1 EOF* handler
A = 2 */ handler
A = 3 * handler
A = 4 *RUN handler
A = 5 *CAT handler
A = 6 Shutdown files
A = 7 Read handle ranges
A = 8 *ENABLE handler
```

OSWORD

```
A = &7D     read disk cycles
A = &7E     read number of sectors
A = &7F
command = &4A write data 128 bytes
command = B4B write data muIti-sector
command = &4E write deleted data 128 bytes
command = &4F write deleted data multi-sector
command = &52 read data 128 bytes
command = &53 read data multi-sector
command = &56 read data + deleted data 128 bytes
command = &57 read data + deleted data multi-sector
command = &5B read sector ids
command = &5E verify 128 bytes
command = &5F verify multi-sector
command = &63 format
command = &69 seek
command = &6C read drive status
command = &75 initialise 8271/load bad tracks
command = &7A write special register
command = &7D read special register
```

Appendix 5.
Disk Error Messages.

Disk Fault/Drive Fault

```
&08 Clock Error
&0C Sector Id CRC Error
&0E Data CRC Error
&10 Drive Not Ready
&12 Write Protect
&14 Track 0 Not Found
&16 Write Fault
&18 Sector Not Found
```

BASIC

```
189 (&BD) Not enabled
190 (&BE) Catalogue full
191 (&BF) Can't extend
192 (SCO) Too many open files
193 (&C1) File read only
194 (&C2) File open
195 (&C3) File locked
196 (&C4) File exists
197 (&C5) Drive fault
198 (&C6) Disk full
199 (&C7) Disk fault
200 (&C8) Disk changed
201 (&C9) Disk read only
203 (&CB) Bad option
204 (SCO Bad filename
205 (&CD) Bad drive
206 (&CE) Bad directory
207 (&CF) Bad attribute
214 (&D6) File not found
220 (&DB) Syntax
222 (&DD) Channel
223 C&DE) Eof
252 (&FC) Bad address
254 (&FE) Bad command
```

Appendix 6. DFS 0.90/1.20 Differences.

1) DFS 1.20 does not allow characters with the most significant bit set to be entered as filenames or disk titles. This prevents you from effecting coloured catalogue displays in Mode 7.

2) DFS 1.20 does not distinguish between upper case and lower case directory codes.

3) *DIR works differently. Consider the sequence:

    *DIR A
    *DIR:1 In DFS 0.90, you end up with a default general purpose directory of '$'. In DFS 1.20 the second command does not overwrite the default general purpose directory which stays as 'A'.

4) DFS 1.20 does not close files on soft BREAK. Neither does it reset the values of *OPT, *DIR and *LIB etc.

5) *ENABLE is superfluous in DFS 1.20. If not used, the command prompts for confirmation.

6) In DFS 1.20, *BUILD and *LIST have line numbers with leading zeroes.

7) DFS 1.20 has several different error messages, most of which are shorter and less helpful than those in DFS 0.90.

8) DFS 1.20 uses 'n:' instead of 'drive n' in messages such as produced by *COMPACT.

9) DFS 1.20 manages the sectors per disk value differently, such that *BACKUP between 40 and 80 track drives produces slightly different results.

10) DFS 1.20 has fixes for lots of the bugs found in DFS 0.90.

11) DFS 1.20 will not *COPY a zero length file.

Index

Other Books in the Series