PRENTICE HALL INTERNATIONAL PERSONAL COMPUTER BOOK

DISK PROGRAMMING TECHNIQUES FOR THE BBC MICROCOMPUTER

MICHAEL COLEMAN

DISK PROGRAMMING TECHNIQUES for the BBC MICROCOMPUTER

DISK PROGRAMMING TECHNIQUES for the BBC MICROCOMPUTER

Michael Coleman

Prentice/Hall  International

Englewood Cliffs, NJ London New Delhi Rio de Janeiro
Singapore Sydney Tokyo Toronto Wellington

British Library Cataloguing in Publication Data

Coleman, Michael
Disk programming techniques for the BBC microcomputer.
1. BBC Microcomputers – Programming
2. Floppy disks (Computer storage device)
I. Title
001.64'42 QA76.8.B35

ISBN 0-13-215930-9
ISBN 0-13-216060-9 (Disk)

ISBN 0-13-215930-9

PRENTICE-HALL INTERNATIONAL, INC., London
PRENTICE-HALL OF AUSTRALIA PTY., LTD., Sydney
PRENTICE-HALL CANADA, INC., Toronto
PRENTICE-HALL OF INDIA PRIVATE LTD., New Delhi
PRENTICE-HALL OF JAPAN, INC., Tokyo
PRENTICE-HALL OF SOUTHEAST ASIA PTE., LTD., Singapore
PRENTICE-HALL INC., Englewood Cliffs, New Jersey
PRENTICE-HALL DO BRASIL LTDA., Rio de Janeiro
WHITEHALL BOOKS LIMITED., Wellington, New Zealand

Printed in the United Kingdom

10 9 8 7 6 5 4 3 2 1

To my dear wife,
THERESA
. . . probably in exchange for a paint brush.

CONTENTS

PREFACE

"They also serve who only stand and wait"
John Milton: On His Blindness

Milton was obviously a cassette tape man. This would go some way towards explaining why, even with his BBC Micro to use as a word processor, his famous poem 'Paradise Lost' took no less than seven years to produce. Had the National Union of Poets been able to negotiate an adequate productivity bonus for stanzas above a daily quota, then surely Milton would have seen the wisdom of investing in a disk drive.

The first thing that strikes any user switching to disk from tape is that, in terms of operational speed, disk is immeasurably faster. Saving and retrieving programs is carried out in seconds rather than minutes. But that should really not be the only benefit realised by the switch; the acquisition of a disk system should open up a whole new dimension to the way in which the BBC Microcomputer itself can be used.

This book aims to give the reader some idea of the tremendous potential that disk systems possess and, moreover, as much guidance as possible into the different ways in which that potential might be realised. The need to be aware of 'Disk Programming Techniques', to quote the title of the book, applies whether the user is concerned chiefly with using disks just to store programs, or intends to write programs which themselves use disks in a big way. Both categories of user should benefit from the material of Chapters 1-4, which is concerned with the facilities provided by the Disk Filing System (DFS) for 'housekeeping': creating, copying and maintaining disk files. Chapters 5-9 apply rather more to those users who intend to develop programs which use disks to hold data files; quite simply, a disk system gives the BASIC programmer unique scope for a variety of data handling activities which are just not possible with cassette tape.

Chapter 10 is devoted to 'Error Messages', and, frail humans that we are, is applicable to all. It must be said that disk systems are a rich source of errors and mishaps and the book does not attempt to gloss over this fact. However, rather than continually punctuate the

text with "if this happens then ..." the majority of references to error situations is confined to this section.

The examples in the book are all based on the version of the BBC Micro Disk Filing System supplied by the manufacturers of the computer itself: Acorn Computers. In order that owners of either a 'Watford' or 'Amcom' DFS might make full use of the book, however, Chapter 11 takes a look at the major additional features of these alternative Disk Filing Systems. It indicates how the alternative system might require changes to any given examples, and provides cross-references to the body of the text.

Proficiency in BBC BASIC is assumed from Chapter 5 onwards. Nevertheless it is the author's view, not wholly due to concern over likely royalty levels, that readers with rather more limited programming powers can still gain substantially from the book. Most points are illustrated by short example programs which should not be unduly taxing; even if this proves not to be the case, any difficulty will at least result in an increase in disk programming knowledge.

Short example programs, however, serve a useful but limited purpose in a book of this kind. Certainly they can illuminate a description of a technique or facility. But if, after digesting what has been written, the reader intends to set about the writing of a 'real' and quite possibly sizeable disk program, then short examples can be found lacking. For this reason a number of Case Studies are included in Section II. Each of these studies is intended to illustrate, within the context of a fully-fledged application program, one or more of the techniques introduced in the body of the text. In content they range from a fairly simple name/telephone number program to an extremely challenging database package. (A disk containing all of the Case Study programs, together with an example database of articles published in the 'Micro User' journal, is available separately.) The studies are not merely presented as programming listings. Each is fully described in terms of the function that it performs, and the methods that it employs. Sample dialogues are given, so that the reader can see what should happen when the program is run, and fully annotated listings provided. For the adventurous, suggestions are made as to ways in which the programs might be raised to even giddier heights of performance.

In the course of writing a book an author gains help from many quarters, of which only a few can reasonably be acknowledged in print. For my part, the few that I would like to select for especial thanks are:

Giles Wright, my editor at Prentice/Hall International, and his accomplices in the process of refereeing this book during its production, for their considerable help (they proved that the phrase 'constructive criticism' is not a contradiction in terms),
the journal 'Micro User' for their permission to use copyright material in constructing the example database,
Fr. Michael Peters, for unwittingly starting the whole thing off,

and finally, my son, Stephen - for managing to make do with only five pages of the original manuscript as padding for his football boots.

Mike Coleman
March, 1984

PROLOGUE
Taking Care of Floppy Disks

flo'ppy (adj.)- inclined to flop
flop (v.)- failure, collapse
The Oxford Dictionary

It is likely that these definitions were far from mind when its originator launched the term 'floppy disk' on an unsuspecting world. Doubtless the new term was intended to reflect the contrast between this new disk and the larger, rigid forms of disk available. The general impression given by the name was of a new, portable, and physically unbreakable medium upon which to store information for computer processing.

This is certainly the case; the floppy disk is indeed very convenient and, unless savaged by a dog or an infant, almost indestructible. It is the latter quality, however, which can become a disadvantage. The fact that a floppy disk will not physically break leads its owner into the mistaken belief that there is no need to exercise care, and that his disks, however badly treated, will not let him down. Such owners will soon discover that the above alternative definitions are most appropriate.

In this short section some golden rules on how to treat floppy disks are provided. They are really just common sense. The list is not very long, and could be summarised by one word: pamper. Pamper your floppy disks, let nothing be too good for them. Remember that they are the equivalent of a bank containing - just as money should be - the fruits of your labours. The difference is that when it comes to your floppy disks, you are banker, guard - and insurance company! Failure is costly.

1. DO NOT BEND FLOPPY DISKS

A strange restriction, perhaps, but the reason is that the 'floppiness' of the disk is a safeguard against breakage, not an invitation to fold it into four. The disk has to be read by a mechanical device which will only work reliably if the disk is completely flat all over.

2. PROTECT FLOPPY DISKS FROM DIRT AND DUST

Every disk comes with a made-to-measure protective sleeve. Use it at all times. An accumulation of dust or grime on the disk surface will inevitably shorten its life. Moreover, this dirt is also transferred to the innards of the disk drive where it can continue to play havoc. Note that this is not just a rule against floppy disks being used in a ploughed field. It also means, for instance, that users should not smoke whilst using disks.

3. NEVER TOUCH THE UNCOVERED PART OF THE DISK SURFACE

A special category of grime is that produced by humans through their fingertips. Hair cream, remnants of cheese roll, and honest-to-goodness sweat are all unwanted on the surface of a disk. We also carry small measures of static electricity around with us (not to mention magnetic personalities) and these can also damage the data on the disk.

4. DO NOT WRITE ON FLOPPY DISKS

Labels which identify the contents of a disk should always be written before being attached to the disk itself. If it is necessary to write on a label that is already in place always use a soft-tipped pen, NEVER a ball-point pen. The reason is that underneath the label and the disk cover is the disk itself. Indentations caused by the exertions of getting ink out of a ball-point pen can be very harmful. Look at the lower sheets of a writing pad to get a graphic impression of what can happen when even a small amount of pressure is exerted.

5. AVOID MAGNETIC FIELDS

The technique for writing information on a disk is based on magnetism. It is therefore possible for data to be erased by other magnetic fields. Most users should be able to avoid laying magnets on top of their disks, but they should also be aware of other sources of magnetism around the house and not leave floppy disks in those areas. One should be wary of leaving disks on top of a television set for instance, and any item of electrical equipment with a transformer in it is a particular danger.

6. KEEP FLOPPY DISKS AWAY FROM EXTREME TEMPERATURES

Typically, disks will operate satisfactorily at temperatures between 10-52 degrees Centigrade (50-125 Fahrenheit) or thereabouts. For normal domestic conditions then, no problems will exist. It is the abnormal domestic condition that has to be avoided: disks on radiators, in direct sunlight, or in the freezer will not preserve well.

7. NEVER FORCE DISKS INTO A DISK DRIVE

Even after keeping one's disks in a cocoon it is still possible to ruin everything by carelessly loading a disk into its disk drive. If the disk drive switch or door will not close, take the disk out and try again. Brute force has never yet improved the condition of either disk or drive.

DISK
PROGRAMMING TECHNIQUES
for the
BBC MICROCOMPUTER

1 BASIC PRINCIPLES

1.1 INTRODUCTION

"In the heaven, a perfect round"
Robert Browning: Abt Vogler, ix

It is surprising that, in trying to account for the vast numbers of women who now indulge in 'jogging', psychologists have completely overlooked another significant trend: the increasing number of rotary clothes lines. This is important. In the days of the conventional washing line, when hanging out the weekly wash meant travelling twice the length of the back garden, women got all the exercise they needed. Nowadays, with their rotary lines, they just have to stand still, pegging items onto the next blank slot which comes around. No movement, no exercise - perhaps jogging is the substitute?

Now, whilst the above assertion is not completely serious, its basis serves a useful purpose, for the principles of long and rotary washing lines are reasonably analogous to those of cassette tape and floppy disk storage. With a long line, data(clothes) can only be stored/retrieved in a serial fashion; it is necessary for the read/write head(Mrs. X) to scan the tape(line) from the start until the required position is reached. Given this mode of operation, any data transfer is likely to be conducted slowly. Moreover, in order to access previously scanned data, the tape has to be rewound, and the process repeated. (Our analogy falls down a little here, since it is Mrs. X that gets rewound!)

However, with a floppy disk (rotary line) the situation is rather different. Here, since the storage medium is continually rotating, it means that any item of data will repeatedly be passing under the gaze of the read/write head. For this reason the speed of access can be improved dramatically. What is more, data areas can be

accessed in any sequence with items being read/written on successive disk revolutions. Thus the constraint of linear operation disappears and the capability for random access has been provided.

There is an implication here, though, which might be best illustrated by extending our analogy a little further. Mrs. X, having hung the weekly wash on her rotary line, is taken poorly (slipped disk?) and retires to bed. When the washing has dried it is Mr. X, therefore, who is detailed to retrieve some selected items. Unfortunately, Mrs. X has always insisted on undressing in the dark, and consequently, Mr. X is somewhat at a loss when garments such as 'woolly leg-warmers' are mentioned. This problem will only be capable of resolution if Mrs. X is able to direct her husband unambiguously to specific areas on the storage device. To be able to do this, she will require

* to know exactly what is on the rotary line;
* to know precisely whereabouts on the line each item is located;
* a way of aiming Mr. X unerringly at the right spot.

Here then Mrs. X is being asked to perform a task analogous to that of a Disk Filing System (abbreviated DFS) in its communications with a disk drive (Mr. X) in order to read information from a disk (rotary line). The solution requires the DFS to be able to produce an 'address' as a key part of its disk handling instruction. This, in its turn, means that a floppy disk must be so organised as to be 'addressable': it must have what is known as a 'format'.

1.2 DISK FORMATS

The format of the rotary line owned by Mrs. X might well be that indicated in Fig. 1-1: six rows of line with three distinct sections. Her instructions to Mr. X then would have been something like, "Woolly leg-warmers are hanging on row 2 of section 3". The format of a floppy disk is not really that dissimilar. The recording surface, Fig. 1-2, is organised as a number of concentric 'tracks', each track having a number of 'sectors'. Thus a disk 'address' will be specified as a combination of track and sector numbers.

This format has, itself, to be written on to the disk. If a particular disk is said to be 'hard-sectored' then it means that its format was imposed on the recording surface during manufacture (as with Mrs. X's rotary line) and cannot be altered. It can therefore only be used with filing systems which are prepared to accept disks having such a format.

The alternative is for a disk to be 'soft-sectored'. That is, the disk is manufactured totally blank and is subsequently formatted as required by a computer program. This enables the same disk to be used with any DFS since it can always be re-formatted to suit.

The BBC Microcomputer DFS will handle disks which have been

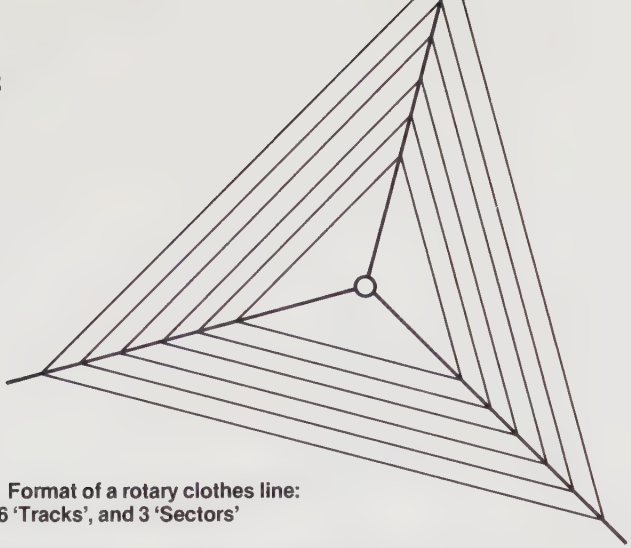


Fig. 1-1 Format of a rotary clothes line: 6 'Tracks', and 3 'Sectors'

soft-sectored; Fig. 1-2 shows the format employed. The recording surface is divided into either 40 tracks (numbered 0-39) or 80 tracks(0-79), each track comprising 10 sectors(0-9). Track 0 is the outermost track on the disk. Since each sector can hold 256 bytes(characters) of information, the disk capacity of each would be

40 track	-	40 x 10 x 256	=	102,400 bytes	(100K)
80 track	-	80 x 10 x 256	=	204,800 bytes	(200K)

The disk format, then, provides the means of locating any particular spot on the disk by giving the DFS a way of nominating a sector and track as the start position for a data transfer. But, referring to our analogy again, generating the right 'rotary line address' required Mrs. X to know precisely where every item was stored on the line. So it is with a disk; the DFS can only specify a disk ADDRESS if it knows whereabouts on the disk the data items are held. It does this by referring to what is known as a 'disk catalogue'.

1.3 THE DISK CATALOGUE

It is the function of the disk catalogue to hold information about data that is held on the disk. In particular, three facts contained in the catalogue are:

1. Data (or File) Name - a name which the DFS relates to an area of disk space. Such an area is termed a file, and the 'file name' is quoted whenever the user wishes to access the information in this file area.
2. Start Address - the track/sector location of the first byte of the file.
3. Extent - the length, in bytes, of the area occupied by the file. This value, rounded up to the next multiple of 256, gives the number of sectors used.

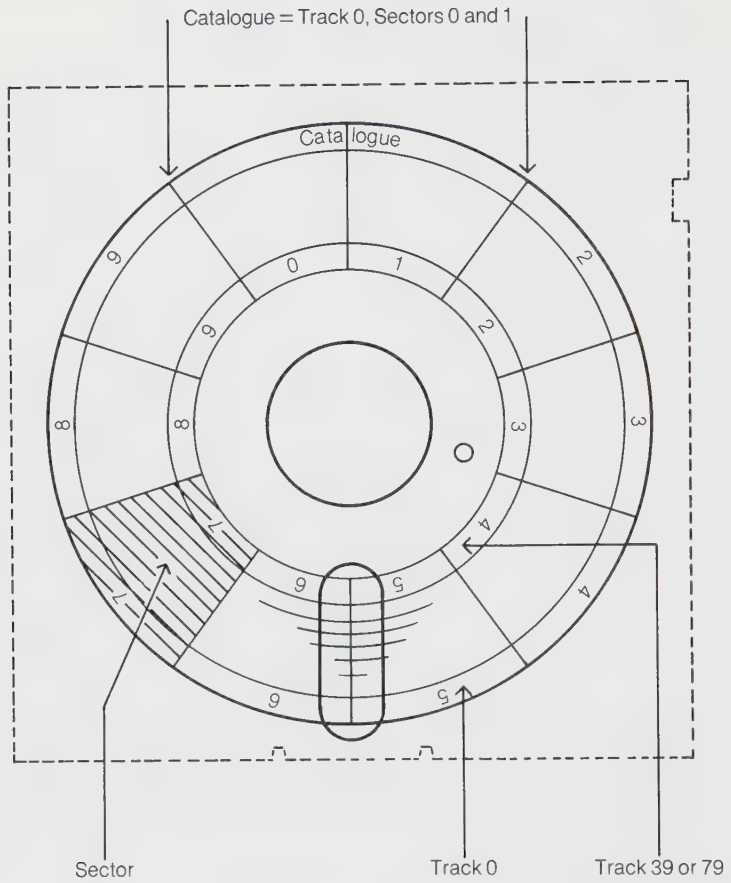


Fig. 1-2 Format of a disk

Given this information, the DFS, when asked to load a program for instance, can tell the disk drive where to look on the disk and how much data to transfer into the computer memory.

As an example, the following dialogue shows the file names recorded in the catalogue of the disk supplied by Acorn with their BBC disk drives. It also shows, for one of these files, the sort of information which is held in the catalogue for every file.
(NOTE: In this, and every dialogue in the book, characters which are to be typed by the user are underlined.)

```
>*CAT
WELCOME(29)
Drive 0                 Option 3 (EXEC)
Directory :0.$          Library :0.$

    !B                      !BOOT
    DCONV   L               FORM40   L
    FORM80  L               VERIFY   L
    content

  W.ALPHA                 W.BATBALL
  W.BIO                   W.BIORTHM
  W.BPART2                W.CALC
  W.CLOCK                 W.HELP
  W.INDEX                 W.KEYBD
  W.KINGDOM               W.MESSAGE
  W.MUSIC                 W.PATTERN
  W.PHONE                 W.PHOTO
  W.POEM                  W.SKETCH
  W.WELCOME

>*INFO W.INDEX

W.INDEX        001900 00801F 000A99 135
```

Catalogue heading

Disk filenames

Filename

Extent

Start address

The values for start address and extent are given in hexadecimal notation. That is, the numbers are written using a base of 16 rather than the base 10 of a decimal number. In this notation, 10 is represented by A, 11 by B, 12 by C, 13 by D, 14 by E and 15 by F. In BBC BASIC hexadecimal numbers may be used; they are prefaced by the symbol '&'. For instance,

```
&10  =  16 decimal (1x16)
&1A  =  26 decimal (1x16 + 10)
&108 = 264 decimal (1x16x16 + 0x16 + 8)
```

Thus the information for file W.INDEX indicates that the file starts at address &135 (309 decimal). Since the disk has a format of 10 sectors to a track this means that the file begins at sector 9 in track 30. The amount of room that is occupied by the file is given as &A99 (2713 decimal) bytes. As space on disk is allocated by the DFS in units of

whole sectors, this means that the file occupies

2713/256 = 10, remainder 153 = 11 sectors.

The reader might care to use the examples above to examine in the same way the contents of the catalogue for any disk that they happen to have available.

In so doing, it will be spotted by the alert that sectors 0 and 1 appear never to be used. This is not the case. In fact these two sectors contain the catalogue itself for, since the catalogue must reflect the contents of one particular disk, it is a fairly obvious strategy to hold a disk's catalogue on the disk itself. In passing, it should be self-evident that this sort of arrangement means that a disk which suffers some grievous corruption to its first two sectors is rendered totally unusable.

1.4 SINGLE AND DUAL-DISK DRIVES

A cursory glance at the advertisements in any popular computing journal will unearth disk drives for sale, their description in dealer's shorthand being along the lines of

"100k/200k/400k/800k Disk Drives for BBC Micro . . "

It has already been mentioned that 100K is the capacity of a 40-track disk, and 200K that of an 80-track disk. Additional capacity can be obtained in one of two ways:

1. Dual-disk drives: quite simply, connecting together two separate disk drives in series to the BBC Micro (see Fig. 1-3), each being capable of individual identification. In this way, 200K and 400k capacities can be achieved.

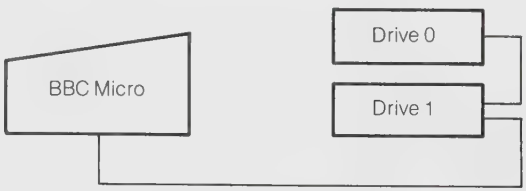


Fig. 1-3 Dual-disk drives

2. 'Double-sided disks': some makes of disk drive can handle information which is written on both sides of what is termed a double-sided (as opposed to a single-sided) disk. Thus the capacity of any one disk is doubled, giving the possibility of 200K and 400K single-disk drives. In such cases, the BBC Disk Filing System regards each side of the disk as being a separate 'drive', and

requires it to be identified as such. That is, a double-sided, dual-disk drive would be treated as though its 800K capacity was made up by 4x200K 'drives', numbered 0,1,2 and 3. In the unlikely event of all this not being crystal clear, Fig. 1-4 attempts to summarise matters.

Capacity	Organisation			Drive numbers
	Tracks	Sides	Drives	
100K	40	Single	Single	0
200K	40	Single	Dual	0,1
	40	Double	Single	0,1
	80	Single	Single	0
400K	40	Double	Dual	0,1,2,3
	80	Single	Dual	0,1
	80	Double	Single	0,1
800K	80	Double	Dual	0,1,2,3

Fig. 1-4 Disk drive configurations

In this book all examples and listings are based on a 40-track, single-disk system. Where users of an alternative configuration, e.g. dual drives, might operate in a significantly different way to that which has been described, additional information is given.

1.5 THE BBC DISK FILING SYSTEM

We have compared (in 1.1) the role of Mrs. X to that of a Disk Filing System, in that the DFS is responsible for maintaining the contents of the disk catalogue. Assume that Mrs. X, having the only rotary line in the road, had decided to go into business by taking in washing from her neighbours. She would now be in regular demand to provide information to her customers who will ask, 'What items of mine are on your line? Is there room for any more? Can I remove something and put something else in its place?'

This sort of facility is a key element as far as customers of the DFS are concerned, since they will need to know similar things about the state of a disk, and want to be able to ask the DFS to carry out certain functions on their behalf. With the BBC Micro, the user can invoke one of a number of DFS commands simply by typing its name at the keyboard. Each of these commands carries out a particular operation concerned with disk handling.

*CAT (abbreviation *.)

Two such commands, *CAT and *INFO, have already been introduced, albeit in passing. The command *CAT can be used to obtain a listing of

the catalogue entries for a currently loaded disk. The example of Fig. 1-3, for the BBC disk, shows the sort of output that is produced. The *INFO command is dealt with fully in Chapter 4.

As with BASIC keywords, every disk command can be abbreviated in some way; '*.'for instance, is the abbreviation for *CAT. As new disk commands are introduced they will be given in both full and abbreviated form.

Additionally, commands might also be provided with information to work with. For instance, *CAT can have a disk drive number supplied, so that

*CAT 1 (or *. 1)

produces a catalogue listing for the disk currently loaded in drive number 1. If the drive number is omitted it is assumed to be zero (termed the drive number 'default' value), so that

*CAT is equivalent to *CAT 0

***HELP (abbreviation *H.)**

To obtain a complete list of the available commands, just type

*HELP DFS

```
>*HELP DFS

DFS 0.90 <---- DFS Version Number
  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>

OS 1.20 <---- Operating System Version
```

List of DFS Commands

The form in which this list appears might benefit from a little explanation. It is presented in a way which gives not only the command

names, but also an indication of what additional information has to be supplied when any particular command is used. Basically, an item within angled brackets <> indicates that a certain item of data must be supplied for the command to work with; items within round brackets () are optional. For instance,

RENAME <old fsp> <new fsp>

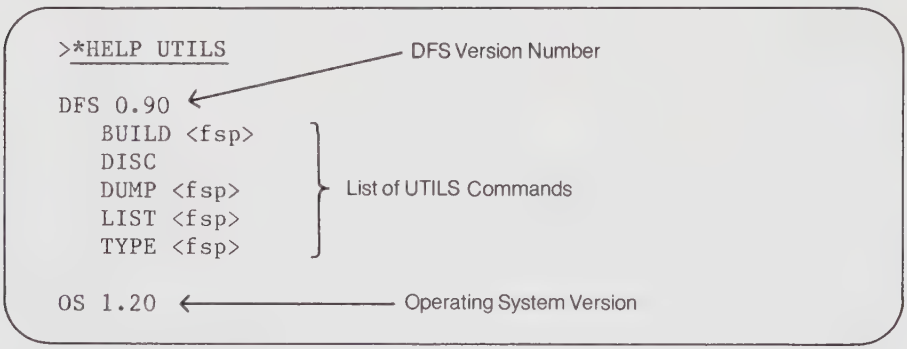
indicates that the command to change the name of a file needs two items of information to work: the current, or 'old' file name, and the name which is to replace it, the 'new' name. However, rather than mention the meaning of the many variants such as <afsp>, <fsp> etc. at this stage, each will be fully explained in their rightful context as the different commands are introduced.

In addition to the DFS commands, various other features of the BBC system can be used with the disk system. These might be grouped under three headings:

(a) UTILS Commands
(b) Machine Operating System (MOS) Statements
(c) BBC BASIC Statements

(a) UTILS Commands

This comprises a set of commands which are primarily concerned with operations on data files. This list is also accessible by using the *HELP command, thus:



(b) MOS Statements

These are functions which, because they are applicable to both tape and disk filing systems, are grouped into the controlling operating system of the BBC Micro. Of the complete list, given on pp. 416-7 of the User Guide, we will be looking at the following:

*CAT (<drv>)

```
*DISC
*DISK
*EXEC <fsp>
*HELP <keyword>
*LOAD <fsp> <load address>
*OPT <function> <option>
*RUN <fsp> (<parameters>)
*SAVE <fsp> <start addr> <finish addr> (<exec addr>)
                                              (<reload addr>)
*SPOOL (<fsp>)
*TAPE
```

(c) BBC BASIC Statements

Some features are embedded into the BASIC language interpreter, so as to allow programs to be written which utilise disks. Again, these are all mentioned in the User Guide, since many are equally applicable to either tape or disk handling. Those to which a disk user will refer are

```
CHAIN <fsp>   }
LOAD  <fsp>   }  for program handling
SAVE  <fsp>   }

BPUT#         }
BGET#         }
CLOSE#        }
EOF#          }
EXT#          }
INPUT#        }  for data handling
OPENIN        }
OPENOUT       }
OPENUP        }
PRINT#        }
PTR#          }
```

All of these different commands and statements will be introduced, hopefully in a logical fashion, as the book progresses. Readers lacking sufficient patience to read the pages of this book serially are referred to the index. There a list of commands and 'book addresses' provide a ready means of random access to this particular (literary) storage device!

2 GETTING STARTED

2.1 DISK FORMATTING

"... in form, in moving, how express and admirable!"
William Shakespeare: Hamlet II, ii

Hamlet, in speaking in such fulsome terms about his disk system, had probably just succeeded in formatting his first ever disk - a necessary exercise to complete before any useful work can be done.

It will almost certainly be the case that the reader will have obtained, along with his new disk drive, what is usually termed a 'utilities disk'. This will be a disk which holds one or more programs which can be used to impose a particular format on a soft-sectored disk. The BBC utilities disk, the content of which has already been mentioned (see 1.3), contains two formatting programs called FORM40 and FORM80. These can be used to initialise disks to 40- or 80-track formats respectively. (Readers who have purchased disk units from a manufacturer other than Acorn should have been supplied with a disk which contains a formatting program. If this is the case, there may well be minor differences in the example which follows.) Exactly which format to employ is determined by the disk drive to be used, and the reader should be aware of whether he owns a 40- or 80-track drive; mixing drives with disks of an incompatible format is a singularly unsuccessful exercise.

Note that all examples in this book are based on a 40-track system although, apart from disk capacity, there would be no visible differences for 80-track systems.

For the owner of a single-drive system, the dialogue in using the BBC formatting program FORM80 would be as shown below. Note that in this, and in many other sample dialogues in the book, additional explanation is provided by comments within brackets ().

```
(Insert the utilities disk into the disk drive.)

>*FORM80 (or *FORM40 if appropriate)  <-------- Call Formatting Program
Disk formatter 1.04

(REMOVE UTILITIES DISK!!! Get into the habit of taking out the
utilities disk at this point. Reflex responses to the next two
prompts would start a formatting of the utilities disk itself!
Now load the disk which IS to be formatted.)

Format which drive ? 0
Do you really want to format drive 0 ?Y

Formatting drive 0

00  01  02  03  04  05  06  07  08  09
0A  0B  0C  0D  0E  0F  10  11  12  13
14  15  16  17  18  19  1A  1B  1C  1D
1E  1F  20  21  22  23  24  25  26  27

Disk formatted - repeat (Y/N) ? N
```

Dialogue with the program

Owners of dual drives could of course use drive 0 for the utilities disk and nominate drive 1 as the format drive. Whatever the case, great care must be exercised since this operation obliterates the original contents of the newly-formatted disk. By way of proof, have a look at the contents of the catalogue for a newly-formatted disk:

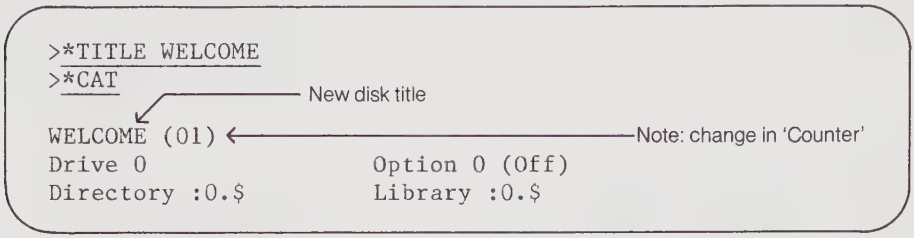
```
>*CAT

 (00)
Drive 0                 Option 0 (Off)
Directory :0.$          Library :0.$
                                    <-------- No files!
```

***TITLE (abbreviation *TI.)**

This listing is as unimpressive as it should be unsurprising. One thing that can be done at this point, and not just to make things look more exciting, is to give the disk a title. Identifying a disk as soon as it is formatted provides a simple check which may prevent the disk being used in error at a later date. The command to use is

*TITLE <title>

where <title> is any name not exceeding 12 characters in length. Use of the command causes two changes in the listing generated by *CAT. For instance,



When the *TITLE command is typed, the <title> part need not be enclosed in quotes "", but no harm befalls the user if it is. Thus,

*TITLE "WELCOME" and *TITLE WELCOME

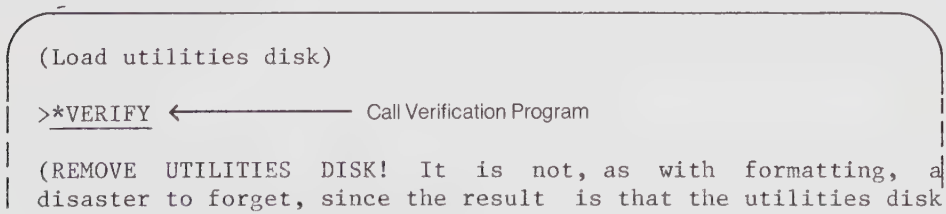
are equally acceptable. This point applies to all of the DFS commands.

With the *CAT output notice firstly that the disk title is the opening item to be output. Also that the number next to the title is altered (in this example from (00) to (01)). This value acts as a 'disk change counter'. It is increased by 1 whenever the contents of the disk change in any way. In this instance, the change has been that of the <title>.

2.2 DISK VERIFICATION

It is worth mentioning at this stage the need for a 'disk verification program'. To verify a disk means no more than to check that each sector of the disk can be read by the DFS. (The actual information on the disk may be absolute rubbish, but that is of no consequence to the DFS.) As such, it is a useful check to make both immediately after formatting and on regular occasions thereafter. Verification failure - which usually indicates a corrupt disk format - could cause a grievous loss of data unless it is detected at an early stage.

The BBC utilities disk contains a program called 'VERIFY' (the reader owning a disk drive from a manufacturer other than Acorn may have a version of this program which differs from that described). The dialogue when using VERIFY, again assuming a single drive system, looks as follows:



```
is itself verified - which may be no bad thing! However, the
next step should be to load the disk to be verified.)

Press 'Y' to verify drive 0 : Y
Verifying drive 0
00  01  02  03  04  05  06  07  08  09
0A  0B  0C  0D  0E  0F  10  11  12  13
14  15  16  17  18  19  1A? 1B  1C  1D
1E  1F  20  21  22  23  24  25  26  27

Disk verified - repeat (Y/N) ? N
```

Dialogue with the program

Although producing a similar output, this operation works at a much faster speed than *FORM40. If all is well, the output should be a series of track numbers 00 to 27. Any doubtful tracks will be indicated (e.g. track 1A in the example) by a '?' symbol. The action taken should be to try *VERIFY again. If the '?' output persists, then the disk is faulty and needs to be re-formatted.

2.3 SAVING BASIC PROGRAMS ON DISK

The formatted and verified disk is now ready for use. As with tape, it can hold both programs (either in BASIC or machine-code) and data. We will be looking at the many possibilities that a disk system affords. At this stage, however, we just consider what is likely to be the most immediate need - storing BASIC programs on disk.

2.3.1 From Keyboard to Disk

SAVE (abbreviation SA.)

With a BASIC program that has been typed in from the keyboard, there is no major difference between the techniques for transferring it to disk or cassette tape. The same command is used, namely

SAVE "<filename>"

Notice, however, that the name of a file on disk has to be limited to a maximum of seven characters (where a character can be anything printable except '*', '#', ':' or '.'). Those with vocabularies sufficiently powerful to fall foul of this limitation will be told by the DFS when they try to specify a filename that is too long,

Bad filename

Another point over which care must be taken is that any existing disk file with the quoted name will automatically be replaced by the SAVE command. If this is not the user's wish, then the existing file should

have its name changed before the SAVE is performed (see *RENAME in Chapter 3).

2.3.2 From Tape to Disk

***TAPE (abbreviation *T.)**
***DISK (abbreviation *D.)**
***DISC (abbreviation *D.)**

A user who is moving from a cassette tape to a disk system will probably have a treasured collection of programs stored on tape. Transferring these programs to disk requires just a little more effort, since it is necessary for the BBC Micro to be instructed to switch between its different filing systems. Three simple commands are involved:

*TAPE - causes the BBC Micro to switch to the Cassette Filing System for subsequent program file or data file input and output.
*DISK - causes the BBC Micro to switch to the Disk Filing System for future file input and output.
*DISC - is an alternative spelling for *DISK.

Thus, a sample dialogue to transfer program "INDEX" from the WELCOME tape would be

```
>*TAPE

(This causes a switch to the Cassette Filing System for file
input/output. A cassette tape recorder should be connected(!),
and the WELCOME tape loaded.)

>LOAD "INDEX"
Searching

(Normal tape loading responses now follow)
Loading
INDEX 00 .. etc

>*DISK

(Causes reversion back to the Disk Filing System)

>SAVE "INDEX"

(Transfers the program to disk)
```

```
>*CAT
WELCOME (02)
Drive 0                 Option 0 (Off)
Directory :0.$          Library :0.$

   INDEX <------------------- File now added to disk
```

2.4 LOADING BASIC PROGRAMS FROM DISK

"Will ye no come back again?
Better lo'ed ye canna be,
Will ye no come back again?"
Baroness Naime, Life and Songs

LOAD (abbreviation LO.)
CHAIN (abbreviation CH.)

The good Baroness would have had no trouble at all in retrieving her disk files if she had been able to spell a little better - 'lo'ed' would have just produced 'Mistake' every time! No, as with cassette tape, to load and run a BASIC program one uses the commands

```
LOAD "filename"
```

and

```
RUN
```

or, to effect an immediate execution of the program,

```
CHAIN "filename"
```

The only difference in using the two media is that the file name must be provided with a disk system, whereas with tape it can be omitted if the next program on the tape is the one required.

2.5 PROGRAM RELOCATION

Programs which occupy a considerable area of store, however, might have to be relocated in order to run successfully. A classic symptom with such a case is that LOAD appears to work, but RUN has no effect. Alternatively, the program starts to run, but fails during execution with the awful error message:

```
NO ROOM
```

indicating that the program had suffocated for lack of store.

The reason, quite simply, is that the Disk Filing System

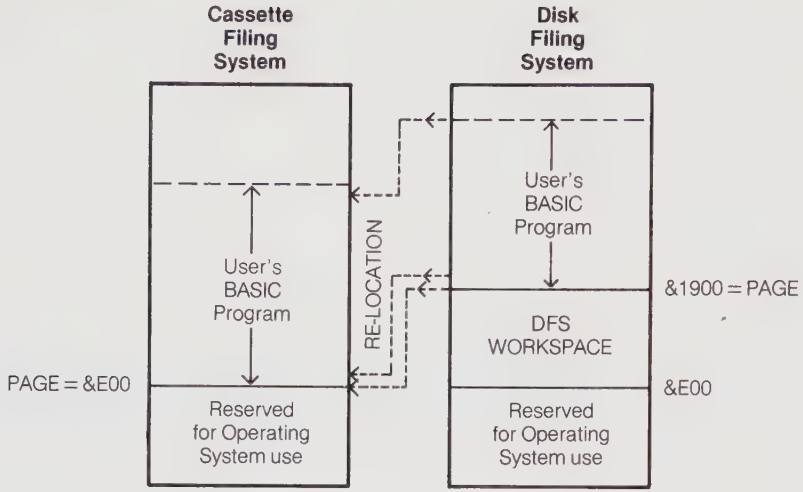


Fig. 2-1 Memory maps for program re-location

occupies a certain amount of store in excess of that used by the Cassette Filing System. A highly simplified memory map (Fig. 2-1) shows that 2816 fewer bytes are available for the BASIC program to use.

However, as long as the program itself will fit into the available space, there is no reason why it cannot, before it is 'RUN', be moved 'down' the store - that is, to the position that it would have occupied had it been loaded under the cassette system.

To relocate a program requires the execution of a rather crude little sequence of instructions which move every single byte of the program 2816 locations down in store. That is to say, byte 6400 is moved to 3584, byte 6401 to 3585, and so on. This effect can be achieved by adding a 'top' and 'tail' to the version of the program held on disk:

```
1 GOSUB 30000: CLEAR
          *
          *
          *
   (Original tape version
        of program)
          *
          *
          *
30000 *TAPE
30010 FOR I% = PAGE TO TOP
30020 ?(I%-2816)=?I%
```

← Loop to move program byte-by-byte

```
(The ? operator (see User Guide p.409) is used to retrieve the
contents of byte I% and to place those contents in location
(I%-2816))

30030 NEXT
30040 PAGE=&E00: RETURN

(Resets PAGE to hold the start address of the relocated
program.)
```

An alternative approach would be to assign the relocation instructions to one of the red function keys. This certainly avoids the need to modify programs in any way and, if many programs are involved, may well be preferable. Essentially the code given above would be employed, although some changes are necessary since line numbers cannot be used - function keys must be related to single-line statements.

```
*KEY 0 *TAPE|M FOR I%=PAGE TO TOP: ?(I%-2816)=?I%:
        NEXT: CLEAR: PAGE=&E00: RUN|M
```

Here the sequence of events, having programmed the function keys, would be to LOAD the program from disk, and then to press key f0. Relocation - possibly taking a few seconds if the program is long - would be carried out, and then the program run would start.

One final point has to be made on this topic. It should be self-evident that when the program has been moved it will occupy that area of store between &E00 and &1900 previously owned by the DFS (used in fact as temporary storage space for information being transferred to and from disk). Thus any subsequent use of a DFS Command will almost certainly result in the DFS reclaiming this area, to the severe detriment of the relocated program's lower parts.

3 GETTING ORGANISED

3.1 FILENAMES

"Oh, talk not to me of a name . . ."
Lord Byron, Stanzas

Byron died of a fever. Therefore charity forces one to attribute his utterance to delirium since a clear understanding of file naming conventions is crucial if the user is to gain fully from a Disk Filing System.

Disk filenames have been introduced already, and it has been stated that they can be up to seven characters in length. This definition was rather simplified. In fact the full form of a filename (or 'file specification' to use the correct title) comprises three elements: drive number, directory identifier and name. A full file specification is written

```
:<drive no>.<directory id>.<name>
e.g.    :0.$.INDEX
```

However, if the first and second components are not given explicitly, the DFS uses its own 'default' values instead. When the BBC Micro is switched on, or when BREAK is used, these default values are set as

```
          $     for     <directory id>
and       0     for     <drive no>
```

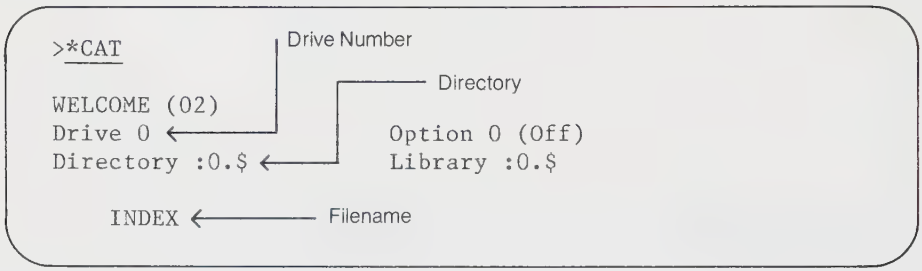
but they may be reset by the user (see commands *DIR and *DRIVE below). It is therefore quite valid to use the <name> part alone. The example

```
SAVE "INDEX"
```

used in Chapter 2, for instance, is completely equivalent to

```
SAVE ":0.$.INDEX"
```

The full DFS filename can be deduced from the catalogue listing, since all three elements are referenced within it:



The naming convention for disk files is thus rather more elaborate than that employed with cassette tape systems, although the Byron-like user is able to ignore this fact for the majority of the time. This is unfortunate if it happens, because the naming technique is designed to give the user greater flexibility when it comes to storing and retrieving programs and data. In presenting the advantages, it is convenient to consider the parts <directory id> and <drive no> separately.

3.2 FILE DIRECTORIES

A directory facility enables the files on a disk to be grouped together, effectively under a number of different headings: the 'directory identifiers'. The reasons for wanting to organise files in this way are many and various. The user might like to group together, say, games programs in one directory, business programs in another directory, data files in another, and so on. Alternatively, programs which are being repeatedly altered during their development might have different versions held in different directories. Mrs. X, for instance, would almost certainly employ such a directory structure with her rent-a-washing-line service (Fig. 3-1).

Look again at the catalogue listing for the BBC utilities disk. The files here are grouped together under two directory identifiers: 'W' for programs found in their original incarnation on the WELCOME cassette tape, and '$' (the DFS default) for the various utility routines. Thus programs can be loaded from this disk by using for instance,

```
                CHAIN "!BOOT"
or              CHAIN "W.INDEX"
```

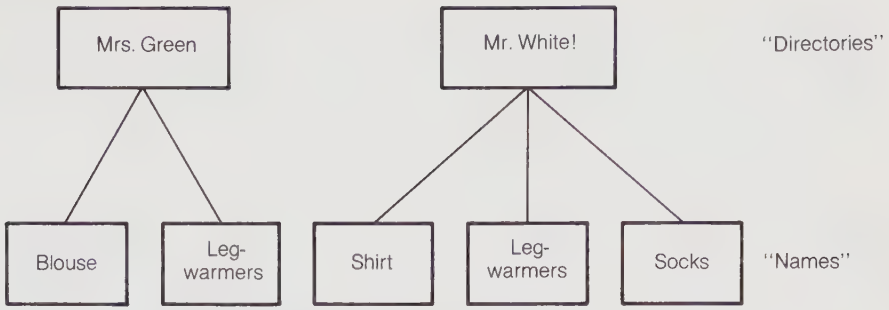


Fig. 3-1 "Washing-line" directory organisation

with the directory taking the default of $ in the first call, and being stated explicitly in the second.

```
>*CAT
WELCOME (77)
Drive 0                 Option 0 (Off)
Directory :0.$          Library :0.$

    !B                      !BOOT
    DCONV    L              FORM40   L   } Directory $
    FORM80   L              VERIFY   L   }
    content

 W.ALPHA                 W.BATBALL
 W.BIO                   W.BIORTHM
 W.BPART2                W.CALC
 W.CLOCK                 W.HELP
 W.INDEX                 W.KEYBD          } Directory W
 W.KINGDOM               W.MESSAGE
 W.MUSIC                 W.PATTERN
 W.PHONE                 W.PHOTO
 W.POEM                  W.SKETCH
 W.WELCOME
```

3.2.1 Setting the Default Directory

*DIR (abbreviation *DI.)

It is very convenient to be able to change the default directory. This can be achieved by using the command

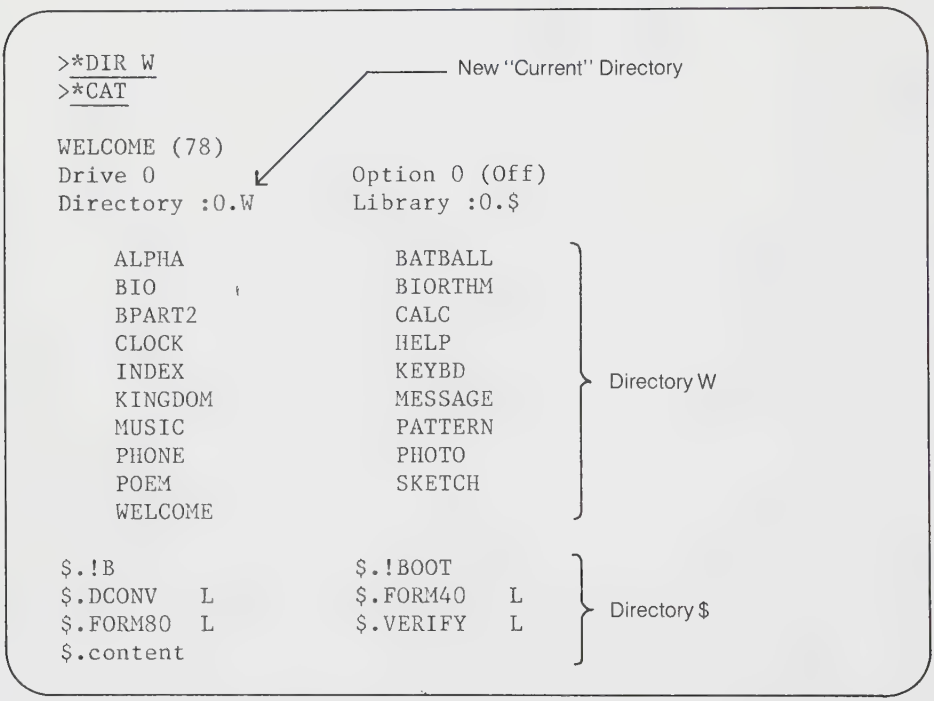
*DIR <directory id>

and, from this point on, the specified directory becomes that which the

DFS uses by default. All file references, be they LOAD, SAVE, *CAT or whatever, will therefore relate to this new directory. The <directory id> can be any single character although the letters A-Z, or digits 0-9, are most commonly used. For instance, the combination

```
*DIR W
SAVE "INDEX"
```

creates a file named W.INDEX. There is a nice change in the catalogue listing which makes the situation very clear:



The files given at the head of the listing are always those which are held within the current directory.

Note that the *DIR command does not alter anything on the disk. In particular, it does not switch a file from one directory to another. To achieve this, the file has to be renamed.

3.2.2 Renaming Files

*RENAME (abbreviation *RE.)

Any existing file can easily have its name changed. The command used is

*RENAME <old fsp> <new fsp>

where <old fsp> means the filename that is to be expunged, and <new fsp> the name that is to take its place. This operation does change details in the disk catalogue. For instance, to alter some of the utility disk filenames to give an 'E' (for 'Educational') directory would require

```
>*RENAME W.ALPHA E.ALPHA
>*RENAME W.CLOCK E.CLOCK
>*RENAME W.KEYBD E.KEYBD
>*RENAME W.MUSIC E.MUSIC
>*RENAME W.SKETCH E.SKETCH
>*DIR E
>*CAT
WELCOME (79)
Drive 0              Option 0 (Off)
Directory :0.E       Library :0.$

    ALPHA                CLOCK
    KEYBD                MUSIC
    SKETCH

 $.!B                 $.!BOOT
 $.DCONV   L          $.FORM40   L
 $.FORM80  L          $.VERIFY   L
 $.content            W.BATBALL
 W.BIO                W.BIORTHM
 W.BPART2             W.CALC
 W.HELP               W.INDEX
 W.KINGDOM            W.MESSAGE
 W.PATTERN            W.PHONE
 W.PHOTO              W.POEM
 W.WELCOME
```

Directory E

Directories $ and W

The *RENAME command is not of course limited to directory changes. To change the name of a file in the current directory for instance:

*RENAME MUSIC TUNES

3.3 DRIVE NUMBERS

A file specification can also contain a 'drive number', which serves to indicate which disk drive is to be used in searching for the required file. For readers having single-drive systems this feature is of little interest since the drive number will be set to 0, the value to which the DFS normally defaults anyway.

Those with dual-disk systems will certainly be able to reference

drives 0 and 1 and, with double-sided disks, drives 2 and 3 also. In the latter case it is confusing, but necessary, to appreciate that the BBC DFS regards each side of a double-sided disk as being a 'drive'. Thus an 800K unit will physically use two disks - the upper surfaces of the disks will be referenced as drives 0 and 1 as normal, and the lower surfaces as drives 2 and 3! To distinguish then between files having the same name and directory, on two different drives, would require the <drive no.> to be supplied with any command, e.g.

```
LOAD ":1.W.INDEX"
```

Note that whenever a <drive no.> is quoted it has to be preceded by the ':' character.

*DRIVE (abbreviation *DR.)

As with a directory identifier, the default <drive no.> can be changed. The command

```
*DRIVE <drive no.>
```

will do the job, and the nominated drive will be used by the DFS in all file references thereafter

```
*DRIVE 1
LOAD "W.INDEX"
```

would be equivalent to

```
LOAD ":1.W.INDEX"
```

in that the file named W.INDEX on the disk in drive 1 would be loaded. It would have the additional effect, however, of also setting the DFS default drive to drive 1.

3.4 FILE ARCHIVING

"For double the vision my eyes do see,
And a double vision is always with me"
William Blake, Letter to Thomas Butts

Witches: "Double, double, toil and trouble"
William Shakespeare, Macbeth IV, i

Taking regular copies of valuable files is an extremely wise precaution. Blake was obviously a model disk user in this respect. But, it must be said, the task can become something of a chore, especially as

one's volume of disk files increases. Macbeth's team of programmers apparently found this to be the case.

A good Disk Filing System will always provide facilities which help the user to maintain a well-ordered and secure set of disk files. It will provide various features for file copying, and this the BBC DFS does in good measure. What it cannot do, however, is to give the user the essential discipline that is necessary to make those copies. It has to become almost second nature to copy files that have just been created or have undergone some substantial change. Now it might seem that this advice applies more to the user of the less reliable cassette tape than to the disk user. Not so. The sheer ease and speed with which disk operations can be carried out means that greater diligence is needed on the part of the disk user to maintain accurate records and backup copies of important files.

In what follows then, not only technique but also procedure will be suggested. The latter may appear to the reader to be somewhat pedantic, and really not worth the effort. So be it. Perhaps only the salutary experience of totally re-constituting a disk for which no copy existed will convey the value of establishing, if not one of those suggested, at least some form of regular routine for file archiving.

*COPY (abbreviation *COP.)

The standard DFS command for making a copy of a file, or files, is:

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

Basically, this command will read a file (<afsp>) from a disk loaded in what is termed the 'source drive' (<src drv>) and write an exact replica of that file to the disk loaded into the 'destination drive' (<dest drv>). Note that when typing this command, all three spaces need to be supplied to separate the different parts. A large number of permutations are possible within the overall operation of this command, enabling the duplication of single or multiple files using either single- or dual-disk systems. We will be taking a broad view, considering the use of *COPY for

* individual files, with a single-disk drive;
* individual files, with dual-disk drives;
* multiple file copying.

In addition, a related function which uses a different command, *BACKUP, gives the capability for

* whole disk copying.

3.4.1 Individual Files, Single-Disk Drive

Copying an individual file with a single-disk drive needs a clear

head since, for obvious reasons, a fair amount of disk loading and unloading is bound to take place. A typical sequence is:

```
>*DIR W
>*COPY 0 0 POEM
Copying from drive 0 to drive 0
Insert source disk and hit a key S
W.POEM        001900 00801F 0025B1 039
Insert destination disk and hit a key D
Insert source disk and hit a key S
>
```

Note: the character does NOT appear on the screen

In carrying out this function, the DFS reads from the source disk into RAM, and then writes from RAM to the destination disk. For this reason any BASIC program in store before a *COPY is unlikely to survive the operation.

The *COPY still works quite happily even if a file is larger than the available area of RAM. What happens in such a case is that the DFS transfers the file in sections, with the user being asked to repeatedly load source and destination disks until the whole of the operation is complete.

If a file of the same name already exists on the destination disk, then it is replaced by the new copy. For this reason using the same disk as both source and destination can only be regarded as the computing equivalent of climbing up an escalator which is going down. To hold two copies of a program on the same disk the directory facility should be used, e.g.

```
LOAD "INDEX"
SAVE "1.INDEX"
```

3.4.2 Individual Files, Dual-Disk Drive

There is no doubt at all that, for the serious and/or affluent user, a dual-disk system is an excellent investment. The fact that file copying becomes so much easier, making the user more inclined to do it, is almost sufficient reason in itself. After loading source and destination disks into their appropriate drives, all that is needed is to type the *COPY command and elevate one's metatarsi:

```
>*DIR W
>*COPY 0 1 POEM
Copying from drive 0 to drive 1
W.POEM        001900 00801F 0025B1 039
>
```

i.e. put your feet up!

Moreover, by switching drive numbers,

e.g. *COPY 1 0 POEM

transfers can be made in any direction. Note that this is also a danger of course; care must be exercised to ensure that source and destination disks are in drives which match the command. For this reason a sensible procedure is always to use the same drives for the same purpose - source disk always in drive 0, and destination disk always in drive 1 is fairly standard.

3.4.3 Multiple File Copying

The need will often arise for a group of files, rather than a single file, to be copied - possibly every file on the source disk. The same *COPY command is used in such cases, but with a change in the way in which the file specification is given. Within a file specification two characters assume a special meaning:

- which is taken to represent any single character
* - which represents multiple #'s

'#' and '*' are said to be 'wildcard' characters when they appear within a file specification. By way of example, referring to files on the BBC utilities disk,

```
$.FORM#0   specifies FORM40 and FORM80
$.FORM*    specifies FORM40 and FORM80
$.###P     specifies $.DUMP
#.###P     specifies $.DUMP and W.HELP
$.*        specifies every file in directory $
#.*        specifies every file in every directory (= all files
                                                    on the disk)
```

So for a multiple copy of every file in directory 'E',

```
>*COPY 0 1 E.*
Copying from drive 0 to drive 1
E.KEYBD          001900 00801F 0024FE 14F
E.SKETCH         001900 00801F 00079D 03C
E.ALPHA          001900 00801F 001180 02A
E.CLOCK          001900 00801F 000808 021
E.MUSIC          001900 00801F 0004F3 01C
>
```

One line of display for each file copied

Similarly, to add the complete contents of the source disk to those of

the destination disk,

COPY <src drv> <dest drv> #.

This feature can be used with either single or dual-disk systems of course. It is rather more energy-sapping with the former, however, since the DFS will request the loading of source and destination disks every time it finds a file to transfer.

*COPY is the first of the DFS commands introduced which accepts file specifications containing wildcard characters. *RENAME, for instance, does not allow them since a command like

*RENAME E.MUSIC E.MUSIC#

would clearly be impossible to interpret. Those commands which do allow wildcards will have the element <afsp> (rather than <fsp>) included in their syntax definition. <afsp>, appropriately, stands for 'ambiguous file specification', and will be met with a vengeance in the next chapter.

3.4.4 Whole Disk Copying

*ENABLE (abbreviation *EN.)
*BACKUP (abbreviation *BAC.)

The *COPY command operates on files, taking them from one disk and adding them to another. If a duplicate copy of a complete disk - unused areas as well as files - is needed then the *BACKUP command is used. Its form is

*BACKUP <src drv> <dest drv>

and its effect is to produce a clone of the disk in <src drv> on the disk in the <dest drv>. Note that whatever happens to be on the destination disk at the start of the operation is overwritten and lost forever. Since this is an irretrievable step, the DFS requires what amounts to a 'dual-key' input before it will act. The command

*ENABLE

has to be used before *BACKUP can be invoked. This feature works in the same sort of way as *COPY, in that the BBC's RAM is used during the transfer with the consequence that any resident program is lost. Also, with a single-disk system, the prompting to load disks is the same. The sequence would be

```
>*ENABLE
>*BACKUP 0 0
Copying from drive 0 to drive 0
Insert source disk and hit a key S
Insert destination disk and hit a key D
Insert source disk and hit a key S
Insert destination disk and hit a key D
Insert source disk and hit a key S
Insert destination disk and hit a key D
                    |
                    |
Insert source disk and hit a key S
Insert destination disk and hit a key D
>
```

Note: *ENABLE then *BACKUP

Switch disk before hitting key!

Multiple 'swaps' of source and destination disks must take place because only a certain amount of data can be copied in one go. This amount, and therefore the number of swaps, will depend on the RAM which is available. With Mode 7, in which transfers of 20K chunks can take place, it will be necessary for six 'load source, load destination' steps to *BACKUP a 100K disk, and eleven for a 200K disk. In other modes, all of which decrease the amount of available RAM, a higher number of swaps would be needed. Using *BACKUP in Mode 3 doesn't bear thinking about!

Once more, life with a dual-disk drive imposes considerably less demand on one's stamina:

```
>*ENABLE
>*BACKUP 0 1
Copying from drive 0 to drive 1
>
```

Copying takes 30 seconds or so after message

3.5 STRATEGIES FOR FILE ARCHIVING

It is customary for air-hostesses to possess three editions of their uniform - 'one on, one at the cleaner, and one in the wardrobe'. The standard approach to file archiving is based on a loosely similar premise - that every file should have at least two backup copies. These copies would be taken at key points in the life history of the file and, for personal computing use, two copies should be sufficient. What will be required then is for two disks to be reserved as backup disks for every working (or Master) disk. Then either when a major change is to take place, or at regular intervals for creatures of habit, copying takes place as indicated in Fig. 3-2.

In practice, the backup disks only need to be used alternately, so long as the user is always aware of which has the next turn. To be absolutely sure, the number of backup disks could be increased to three,

Fig. 3-2 Taking backup copies

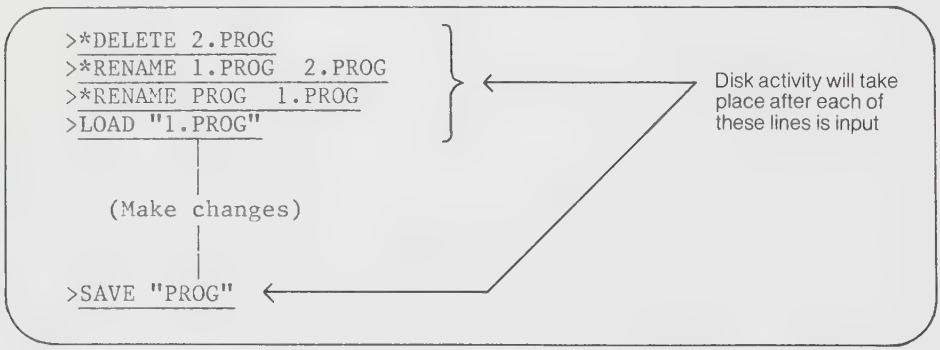
giving the 'Grandfather', 'Father', 'Son' approach beloved of large computing organisations.

The actual process of making these copies with the BBC DFS would be by means of the dynamic duo, *ENABLE and *BACKUP. This is no hardship at all for users of dual-disk systems, as we have seen, but for single-disk owners regular copying of complete disks would certainly involve both time and patience.

For such users, so long as the master disk has relatively few (up to 10) programs on it, the following technique might commend itself. It combines the principle given above with a simple variant of what is known as 'incremental dumping' - making regular copies of only those files that change, rather than performing full disk copies. The idea is that the disk uses three directories:

```
$ - working directory
1 - recent copy ('SON')
2 - old copy ('FATHER')
```

Programs in their development state are held in directory $. When a copy needs to be made of any file, the relevant entry in directory 1 is moved to directory 2, and that in directory $ to directory 1, thus:



This sequence has to employ a command, *DELETE, which is described more fully in the next chapter. Basically, it does what might be expected and removes the quoted file (2.PROG) from the disk. This is necessary because another file cannot be *RENAMEd to 2.PROG if a file of this name already exists. Two additional disks can now be used as general 'SON' and 'FATHER' backups for a number of working disks. At

rather longer intervals, current copies of all 'SON' files could be copied to the appropriate backup disk with the *COPY command, e.g.

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

Similarly, the 'FATHER' files are copied to the other backup disk:

```
*COPY 0 0  2.*
```

Fig. 3-3 illustrates the process.

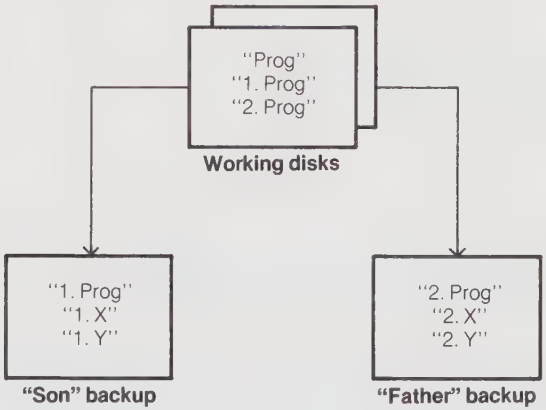


Fig. 3-3 Incremental dumping

Finally, and for the same reason that the President and Vice President of the U.S.A. never fly on the same aeroplane, at least one of the backup disks should be lodged in some safe location away from the other backup copy. Leaving one's disks on the train is then not quite so much of a disaster if at least one copy is safely tucked away at home.

Now admittedly all this requires some effort on the part of the user. The end result, however - security of files and peace of mind - should make it all worthwhile!

4 GETTING RID

"In every parting there is an image of death"
George Eliot, Scenes of Clerical Life

George (or Mary to those who knew her well) used to become quite attached to her disk files - hence the rather soulful reflection on the subject of file deletion. But, given the finite capacity of a disk (31 files with the BBC DFS), it is often necessary to harden the heart and just get on with the grisly business.

Accidental deletions are a different matter though, and most certainly a cause for mourning. For this reason, just as one should know how to stop a car before starting it, we begin with ways of protecting files against deletion.

4.1 FILE PROTECTION

Broadly speaking, decisions concerning the protection of a disk are rather like those involved when insuring a house against various destructive forces. Two types of policy can be adopted - either the whole structure can be covered, or individual items within it. As regards disks, the decisions concern whether one should protect the whole disk against any chance of its being altered, or merely protect individual 'high-risk' files. Depending on the circumstances, either or both methods can be adopted. What must be borne in mind is that a file can not only be explicitly removed by the deletion commands that the DFS provides, but also by more or less subtle means, such as file copying and disk formatting. A disk insurance policy should be formulated accordingly.

4.1.1 Write Protection

Protecting a disk against any form of modification is both simple and crude - but effective. Every disk is manufactured with a notch cut out of it. This is a 'write-protect' notch (see Fig. 4-1) and has to be detected by the disk drive before it will attempt to write to the disk.

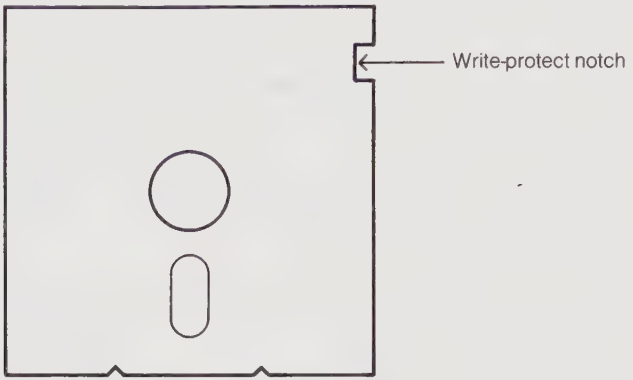


Fig. 4-1 Location of disk 'write-protect' notch

All that is required to ensure that the disk remains unsullied then is for this notch to be covered by something like a tab of adhesive tape. Most boxes of disks arrive with a strip of these tabs enclosed. A disk modified in this way will now be free from all danger. Any attempt to alter its contents in any way will produce the DFS message,

Disk Read Only

In particular, this is the only sure way of protecting the contents of a disk from accidental erasure through either re-formatting or the use of *BACKUP.

Now whilst it makes good sense for disks which contain permanent or long-term information (such as backup disks) to be protected in this way, it is rather cumbersome to be continually removing and replacing write-protect tabs on everyday working disks. For these it is more suitable to adopt the policy of preventing individual files from being changed unless the user makes a conscious effort to indicate that this is what he wants to do.

4.1.2 File Access Protection

*ACCESS (abbreviation *A.)

Individual files can be protected against being either deleted or overwritten by having their status in the disk catalogue defined as 'locked'. Such a file can then only be input to the computer. It cannot be changed in any way (which includes deletion!) until it has been

'unlocked' again. Both operations are performed by using the DFS command:

*ACCESS <afsp> (L)

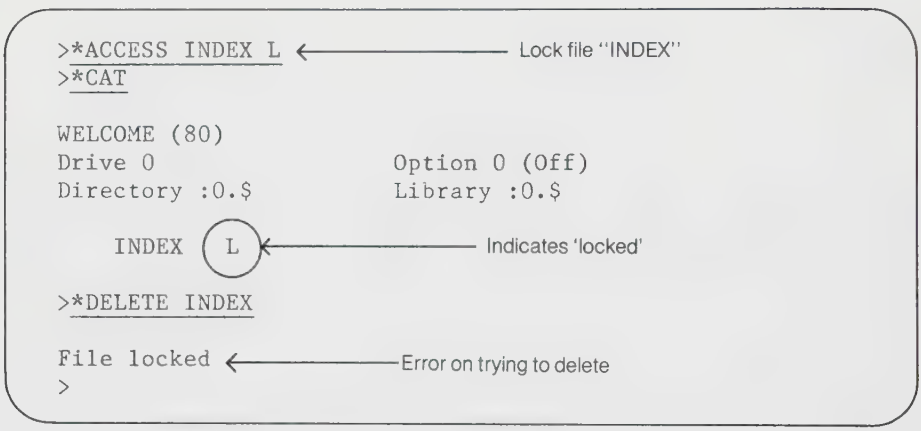
A file is given 'locked' status by a command such as

*ACCESS INDEX L

this new state being reflected in the catalogue listing with the character 'L' appearing after the file name. After this, any attempt to remove the file is met with a blunt refusal and the message,

File locked

as shown below:



Since *ACCESS can take an ambiguous file specification, use of the 'wildcard' feature enables a number of files to be locked at once. To lock all files in the default directory, for instance, type

*ACCESS * L

and to lock all files on a disk - that is, all files in all directories - type

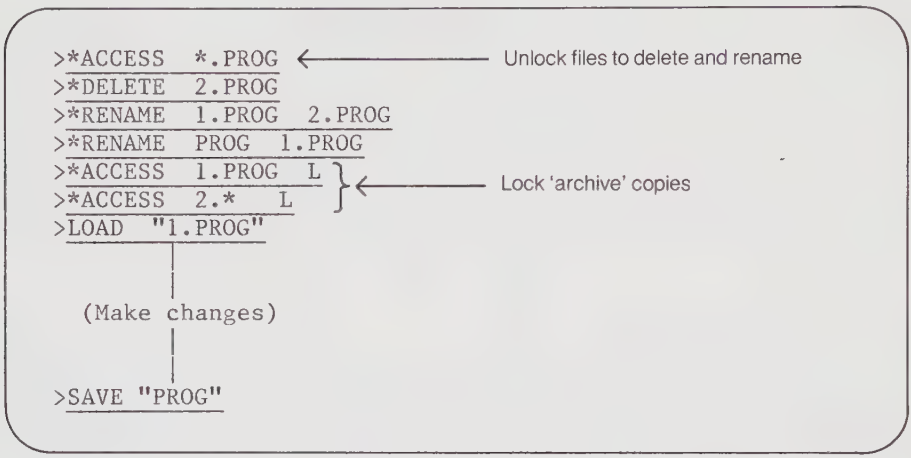
*ACCESS *.* L

A number of files on the BBC utilities disk are locked against overwriting. To unlock a file the same command is used, but without the 'L' indication at the end:

*ACCESS INDEX

The file can now be deleted.

It is a good idea, if one is following the incremental archiving procedure outlined in Chapter 3, to protect all files in the 'SON' and 'FATHER' directories in this way as an added safeguard. The given sequence would change to the following:



It is necessary to unlock the files first, since *RENAME counts as a deletion and is prevented from taking place on a locked file. Also the ambiguous filespec when re-locking does no harm in that locked files just stay locked, and might do some good. Every file in directories 1 and 2 is set as locked, which may correct any previous omissions.

4.2 FILE DELETION

The Disk Filing System provides three commands especially for the deletion of disk files. We look at each in turn to consider its method of use. Then an example sequence shows how, sacrilegious though it might be, the commands could be employed to erase large portions of the BBC utilities disk.

*DELETE (abbreviation *DE.)

Removal of a particular file, as shown in our archiving approach of the previous section, is achieved by the command:

```
                *DELETE <fsp>
e.g.            *DELETE INDEX
```

Nothing more is required. The nominated file, INDEX in this case, will be removed. The file specification, as we have seen with other commands,

will default to the current directory and drive if these are not given. Note in particular that *DELETE only accepts <fsp> and does not like ambiguous filespecs at all. This is consistent with its declared function of deleting an individual file.

*WIPE (abbreviation *W.)

However, there may well be occasions, in the spring perhaps, when a fair number of files are to be deleted. In such cases *DELETE can be used repeatedly, but this is somewhat tedious. A more efficient ploy is to use

*WIPE <afsp>

Because an <afsp> can be used here, groups of files may be nominated for deletion, e.g.

WIPE E.

specifies every file in directory 'E'. Given the finality of this exercise however, the actual execution of the file is not carried out until a second decree is made. This is forced by the *WIPE command in that every unlocked file which matches the filespec is displayed in turn, the user being required to type either 'Y' or 'N' to confirm deletion or grant a stay of execution respectively. This command is particularly useful for performing a considered purge of all files in a directory or even on a complete disk. One thing to be very careful about with this command is that the 'Y/N' response does not have to be followed by RETURN. Thus if the 'Y' or 'N' key is held down for too long the same character will be taken as the response to the prompt for the next file(s) in sequence. This may cause a deletion which was not wanted.

An example in which the *WIPE command is used extensively begins on the next page.

*DESTROY (abbreviation *DES.)

For those who prefer to think big, who customarily deal in multiples rather than singles, there exists the command,

*DESTROY <afsp>

which will delete en bloc every file covered by the <afsp>. Thus,

DESTROY E.

would delete every file in directory E. The difference between *WIPE and *DESTROY is that the former asks the user again before acting against any of the files under sentence whereas the latter presents a list of the condemned and wants no more than a positive response before it

deletes the lot. This is another instance of a DFS command that gives a user the opportunity to do irreparable damage to the contents of a disk. For this reason, as has been seen with *BACKUP, a 'dual-key' approach is taken for the purpose of self-protection. Before *DESTROY will be performed by the DFS, *ENABLE must have been used.

This trio of commands will, with a well-organised disk user, be called into action quite regularly. In many ways it would be sensible, if at times tedious, just to use *DELETE - at least then a file has to be explicitly named before it is deleted. However, with care *WIPE is exceedingly useful. It does after all ask the user about each specified file before removing it, and so accidents should not occur! Frankly, *DESTROY could so easily be a disaster that for the time saved by using it in preference to *WIPE, it might be felt best to ignore its existence. Be that as it may, there now follows by way of example a systematic attack on the BBC utilities disk which is designed to show how each of these commands is used for real.

```
>*CAT
WELCOME (79)
Drive 0            Option 0 (Off)
Directory :0.$     Library :0.$

   !B                  !BOOT
   DCONV    L          FORM40    L
   FORM80   L          VERIFY    L
   content

  E.ALPHA             E.CLOCK
  E.KEYBD             E.MUSIC
  E.SKETCH            W.BATBALL
  W.BIO               W.BIORTHM
  W.BPART2            W.CALC
  W.HELP              W.INDEX
  W.KINGDOM           W.MESSAGE
  W.PATTERN           W.PHONE
  W.PHOTO             W.POEM
  W.WELCOME

>*DELETE content
>*DELETE W.BIO
>*DELETE W.P*

File not found
>*WIPE W.P*
W.PHOTO      :Y
W.POEM       :Y
W.PHONE      :Y
W.PATTERN    :Y
```

Initial contents of disk

Delete individual files

'Wildcards' not allowed with *DELETE
—thus file 'W.P*' not found

```
>*CAT
WELCOME (85)  <-- This value increased by 1 every time the catalogue changes
Drive 0             Option 0 (Off)
Directory :0.$      Library :0.$

   !B                  !BOOT
   DCONV   L           FORM40   L
   FORM80  L           VERIFY   L

  E.ALPHA             E.CLOCK
  E.KEYBD             E.MUSIC
  E.SKETCH            W.BATBALL
  W.BIORTHM           W.BPART2
  W.CALC              W.HELP
  W.INDEX             W.KINGDOM
  W.MESSAGE           W.WELCOME

>*DIR W
>*WIPE *
W.BIORTHM     :Y
W.BATBALL     :Y
W.HELP        :N  <-- Do not delete "HELP" and "WELCOME"
W.KINGDOM     :Y
W.MESSAGE     :Y
W.BPART2      :Y
W.CALC        :Y
W.INDEX       :Y
W.WELCOME     :N
>*CAT

WELCOME (92)
Drive 0             Option 0 (Off)
Directory :0.W      Library :0.$

   HELP                WELCOME

 $.!B                $.!BOOT
 $.DCONV   L         $.FORM40   L
 $.FORM80  L         $.VERIFY   L
 E.ALPHA             E.CLOCK
 E.KEYBD             E.MUSIC
 E.SKETCH

>*WIPE !B*  <-- Because directory "W" still in use, and no file "W.!B" exists

File not found
>*WIPE $.!B*
$.!B          :Y
$.!BOOT       :Y
```

```
>*DESTROY E.*

Not enabled <-------------- Must use *ENABLE before *DESTROY
>*ENABLE
>*DESTROY E.*
E.KEYBD   ]
E.SKETCH  }
E.ALPHA   } List of condemned files
E.CLOCK   }
E.MUSIC   ]

Delete (Y/N) ? Y
Deleted
>*CAT

WELCOME (95)
Drive 0            Option 0 (Off)
Directory :0.W     Library :0.$

   HELP                WELCOME

 $.DCONV   L        $.FORM40   L
 $.FORM80  L        $.VERIFY   L

>*DELETE $.DCONV

File locked
>*DIR $
>*ACCESS DCONV <-------------- Unlocks file "DCONV"
>*DELETE DCONV
>*CAT

WELCOME (97)
Drive 0            Option 0 (Off)
Directory :0.$     Library :0.$

   FORM40   L          FORM80   L   ]
   VERIFY   L                       } Final contents of disk
                                    ]
 W.HELP             W.WELCOME
```

Post Mortem

To say that a file is 'deleted' by the commands *DELETE, *WIPE and *DESTROY is to convey the impression that the contents of the file concerned are actually removed in some way from the recording surface of the disk. This in fact is not the case at all. What the DFS does, in order to delete a file, is to simply remove its name from the disk catalogue which is held in sectors 0 and 1; the area of the disk which

held the content of the file is left untouched. (An interesting consequence of this fact is that the possibility must therefore exist for a file that has been accidentally deleted to be recovered if only its name can be restored in the catalogue. This can indeed be achieved if action is taken swiftly. It needs, however, a routine which uses calls at the Operating System level, and is consequently beyond the scope of this book.)

What is important, though, is that the DFS will record the fact that the area on disk occupied by a deleted file has become 'free' - that is, this area can be used by the DFS whenever it has to find some space in which to store a new file. This process is carried out automatically and in this respect is of little concern to the user. However, the end product - the allocation of disk space to files - does need to be appreciated if disks are to be used efficiently.

4.3 DISK SPACE ALLOCATION

A simple example will serve to illustrate the resultant state of a disk which has been subjected to a number of file deletions.

*INFO (abbreviation *I.)

To discover how the recording surface of a disk is currently occupied, the command

*INFO <afsp>

can be used. This produces for every file nominated (note that it can take an ambiguous filespec) the following items of information:

Directory
Name
Access
Load Address
Execution Address
Length of file
Start Sector

The only items which have not been mentioned already are Load Address (the location in RAM to which the file would be loaded by default), and Execution Address (for files containing programs, the location of the first executable instruction).

The *INFO command can be used to produce information for a single file or, more commonly perhaps, for every file on a disk. This is achieved by

*INFO *.*

For the remnant of the BBC utilities disk after the previous demonstration of how to delete files in different ways, we see

```
>*INFO *.*
$.VERIFY   L  002800 002800 000200 14B
$.FORM80   L  002800 002804 000300 148
$.FORM40   L  002800 002800 000300 145
W.HELP        001900 00801F 00012B 0BC
W.WELCOME     001900 00801F 0007D4 005
```

One line of output per file

In a very real sense one now has to 'read between the lines' to discover which areas on the disk are regarded as 'free space' by the DFS. Summarising the above output from *INFO then, with Start Sector converted to Track/Sector, and remembering that one sector equals 256 bytes, we have

	START Track/Sector		LENGTH (Bytes)	FINISH Track/Sector	
FREE	0	2		0	4
W.WELCOME	0	5	2004	1	2
FREE	1	3		18	7
W.HELP	18	8	299	18	9
FREE	19	0		32	4
$.FORM40	32	5	768	32	7
$.FORM80	32	8	768	33	0
$.VERIFY	33	1	512	33	2
FREE	33	3		39	9

Now if the user attempts to create a new file on this disk, the DFS will utilise a suitable area from those which it knows are FREE:

```
>NEW
>10 PRINT "THIS IS A DUMMY PROGRAM"
>20 GOTO 10
>30 END
>SAVE "W.DUMMY"
>*INFO *.*
$.VERIFY   L  002800 002800 000200 14B
$.FORM80   L  002800 002804 000300 148
$.FORM40   L  002800 002800 000300 145
W.HELP        001900 00801F 00012B 0BC
W.WELCOME     001900 00801F 0007D4 005
W.DUMMY       FF1900 FF8023 000035 002
```

Saved program

occupies one sector

This trivial program only needs a single sector of disk space, and so

the DFS slots it into the first area that is big enough - in this case, Track 0/Sector 2.

Although a simple example, it is hopefully enough to enable the reader to see how the DFS tackles the problem of finding disk space for new files. It should also be apparent that over a period of time, with a number of deletions and creations, a disk will become littered with little pockets of free space in between areas that are holding files.

Sooner or later then it will be the case that the user will want to save another file, and the DFS will be unable to find a large enough area of continuous free space into which that file can be put. If this is the case then the message

Disk full

is produced even though, considering all of the free areas added together, there might well be enough space available. What is required is for the disk space to be 'compacted'.

4.4 DISK SPACE COMPACTION

*COMPACT (abbreviation *COM.)

The command,

*COMPACT (<drv>)

comes to the rescue here. Using either the current or the nominated drive, it moves every file on the loaded disk so that, commencing from Track 0/Sector 2, they follow each other elephant-fashion in one continuous sequence. The result is that the disk space beyond the end of the last file is that deemed 'free', and it is also in one large section. If a new file cannot now be saved on the disk, then there simply is not enough room available without deleting some other file.

It is sound practice to perform a *COMPACT operation on a fairly regular basis, and almost certainly after any major sessions of file extermination. The command produces a summary of the new locations of each file and a statement of how large the revised area of free store is:

```
>*COMPACT
Compacting drive 0
W.DUMMY        FF1900 FF8023 000035 002
W.WELCOME      001900 001900 0007D4 003
W.HELP         001900 00801F 00012B 00B
$.FORM40    L  002800 002800 000300 00D
$.FORM80    L  002800 002804 000300 010
$.VERIFY    L  002800 002800 000200 013
Disk compacted 17B free sectors
```

Each line of output produced after file has been moved

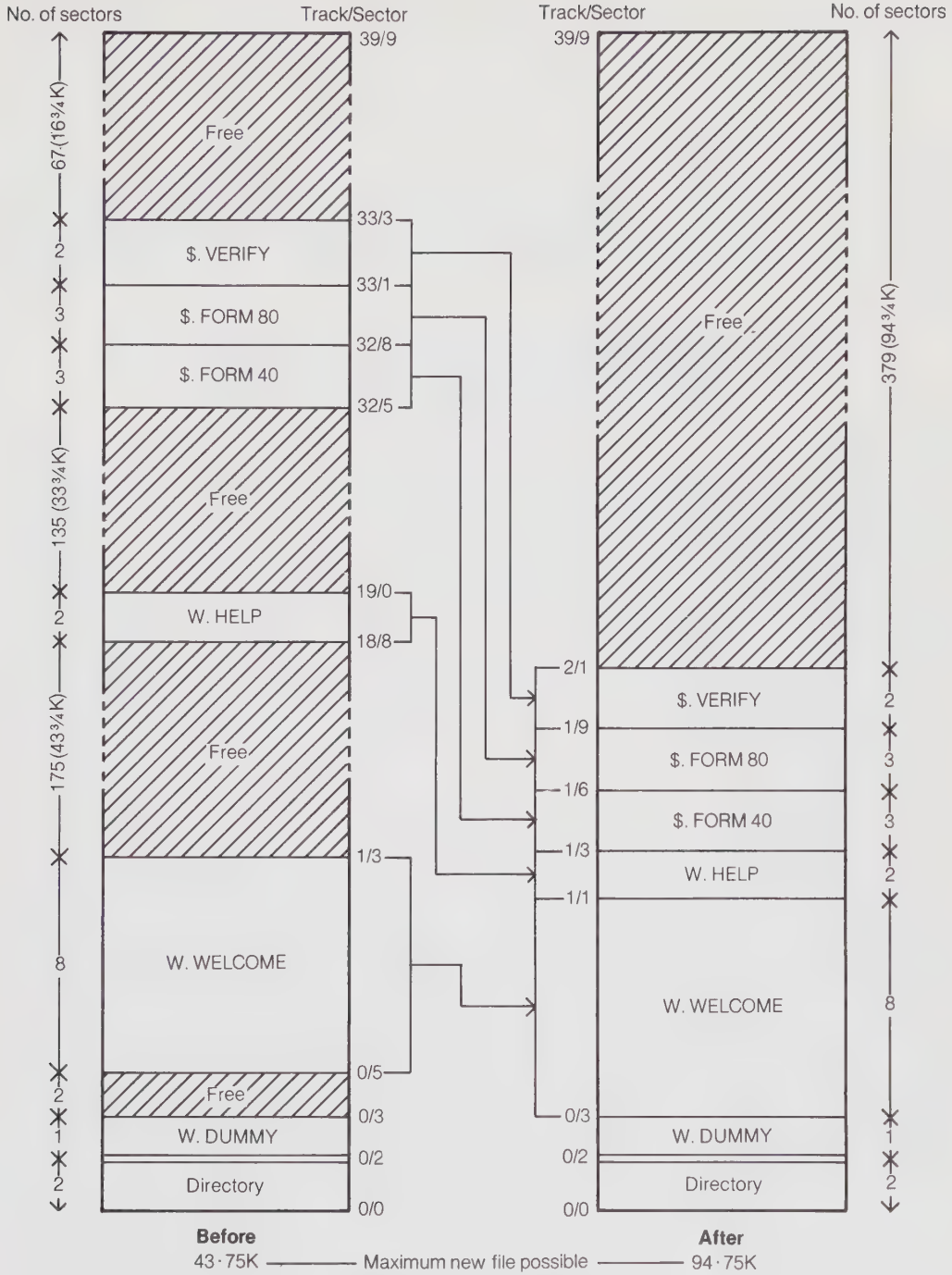


Fig. 4-2 Use of *COMPACT

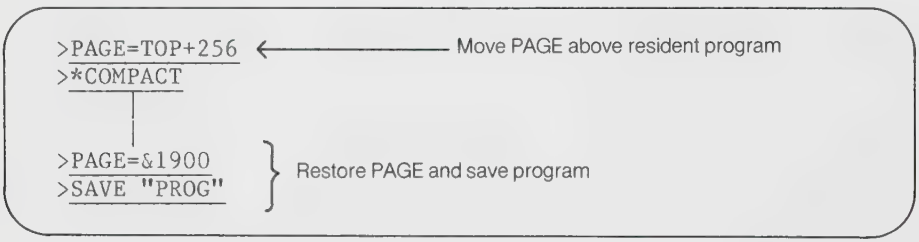
Fig. 4-2 shows this example of using *COMPACT in diagrammatic form. Note that the largest file that could be created will increase dramatically from about 44K to 95K once the disk has been compacted.

It is important to remember that *COMPACT is another of those DFS commands which use the BBC Micro RAM as a temporary store. (It is not difficult to identify such commands - any that involve copying of files in some way, e.g. *COPY, *BACKUP, *COMPACT, will seize RAM without so much as a by-your-leave.) This means that any program currently in store must be saved before the *COMPACT operation begins. Now the alert reader will have spotted a potential 'Catch-22' situation here:

```
*No room to save program on disk
*Cannot *COMPACT until program is saved!
```

The simple recourse in such a situation is to use another disk temporarily and transfer the resident program to that disk. After the compaction has been carried out on the original disk, the saved program has to be copied to it from the temporary. Those whose allowance from the family budget is insufficient to enable them to purchase more than one disk are duly warned.

Alternatively, since *COMPACT actually uses RAM from location PAGE onwards, a resident program could be protected by the command sequence,



For large programs though, the area of RAM between TOP and HIMEM could be very small. Compacting a 100K disk by moving 10 bytes at a time will not be a swift operation.

Once again the reader is advised to make this operation something of a habit. Using *COMPACT regularly, perhaps as the final step of the archiving sequence, will prevent the occurrence of this sort of inconvenience.

5 SERIAL DATA FILES

"It is a capital mistake to theorise before one has data"
Sir Arthur Conan Doyle, Scandal in Bohemia
(A Sherlock Holmes story)

Holmes, doubtless with Watson looking on open-mouthed, shows that his perception was not limited to matters involving the criminal classes. His observations with regard to programming were also acutely accurate - before any computer program is developed due regard must be given to the nature of the data upon which that program will be expected to operate. This means not only the type of data - numbers, strings, etc. - but also the way in which the program will be accessing that data: is serial or random access necessary? In this chapter we will be concerned with programming techniques for data files which are to be accessed serially. Random access files are covered in Chapter 6.

5.1 SERIAL DATA FILES

As its title suggests, a serial data file is one whose elements are to be accessed by a program in sequence, starting from the first and working one by one through to the last. Since this is precisely what happens with a data file on cassette tape, one might expect a serial disk file to be handled by a program in much the same sort of way. As we shall see, this is indeed the case.

In like manner the application areas for serial disk files are no different to those for which cassette tape can be employed. Typically, the application will need to read the data items, either processing them individually as they are input or storing them in arrays for later processing. The end product of the program might well be another serial data file. A word processing program is an example here. The data file which contains, say, the text of a letter is input serially by the program and displayed on the screen. After manipulation by the user the revised text is written into another serial file.

Some applications can be programmed using either serial or random access files. Case Study 1, which appears in Section II of this book, is an example. There a serial data file is established which

holds names and telephone numbers, the case study program reading the whole of the file into store in order to operate upon it. In Case Study 2, however, the same application is shown to be more effectively dealt with by using a random access file to enable just a selected entry to be retrieved.

With serial files on disk then the user should be aware that nothing is being achieved that could not also be managed with cassette tape. The advantages of the former of course are primarily in terms of speed, but there are other bonuses. A disk does not have to be rewound in order to read the file again; neither does it have to be positioned before a new data file can be created - the standard features of the DFS take care of these aspects.

5.2 PROGRAMMING FOR SERIAL DATA FILES ON DISK

In terms of programming for serial files on disk then it should come as little surprise to discover that there are few differences from the approach taken with cassette tape. The example programs in this chapter, for instance, would run without alteration under a tape system. All of the BBC BASIC statements to be used apply to both disk and tape systems in a similar sort of way, and each is documented in the User Guide. Since serial data files are either being created by a program or having their contents input by a program, it is convenient to introduce the relevant BBC BASIC statements with regard to these two operations.

5.2.1 Creating a Serial File

In simplistic terms a file is created in three steps - it is 'opened', information is put into it, and finally it is 'closed'. Each step is essential to ensure satisfaction both of programmer and of Disk Filing System.

OPENOUT (abbreviation OPENO.)

The action of 'opening' a file serves to communicate certain items of information to the DFS. In particular it names the file to be created (with an initial allocation of 64 sectors) and enables the DFS to conduct various internal operations such as putting the file name into the disk catalogue. The function is also a means of stating how the file is to be used by the program. Having opened a file for output, a program cannot subsequently try to read data from that file, and the DFS is able to check for this. In BBC BASIC the statement

```
OPENOUT
```

is used to open a disk file ready to receive data output by the program.

Its general form is:

<channel>=OPENOUT(<filename>)

The <channel> is a variable name which is assigned a 'channel number' by the DFS when OPENOUT is obeyed. The number itself is of no consequence; what is important is that this variable has to be quoted whenever the file is used within a BASIC program. It is common to use an integer variable here since the <channel> value will always be an integer. In the sample programs which follow and in the Case Studies the variable 'fo%' is always used - this is to indicate that the variable is representing a (f)ile which is opened for (o)utput. The <filename>may be either a string or a string variable, giving the name of the file to be opened. It is the case, and will be assumed from here on, that <filename> can be in any of the forms that we have considered to date - that is, it may include both directory and drive if required. Some example calls of OPENOUT are

```
fo%=OPENOUT("DATA1")
fo%=OPENOUT(":1.$.NAMES")
fo%=OPENOUT(fnam$)
```

There are two important differences between disk and tape systems as far as OPENOUT is concerned. With disk any existing file with the specified <filename> is deleted. This is one of the rather more subtle ways in which a user can foul things up on his disk unless the original file has been protected by use of *ACCESS. The second point is that more than one file can be opened at a time (to a maximum of five open simultaneously). Different <channel> variables are used with each OPENOUT and, by using the right one, data could easily be sent to different files by the same program. This is really not a feasible proposition with tape systems!

PRINT# (abbreviation P.#)

Having opened the file, data can now be written to it. The BBC BASIC statement employed is

PRINT#<channel>,<item(s) to be output>

The <channel> quoted is that used with the corresponding OPENOUT and in this way the DFS is able to determine into which file the data is to be written. For example,

```
PRINT#fo%,N
PRINT#fo%,name$,tel$
```

Note that, although it is used in much the same way, there are certain differences between PRINT# and PRINT, its counterpart for screen/ printer output. For instance, only the character ',' can be used to

separate items in the list after PRINT#. The character ';' commonly used in PRINT statements, is not to be used with PRINT#. Functions such as TAB and SPC cannot be used either, and the '@' feature has no effect on the way that numbers are held in a disk file.

In summary, therefore, any data file will comprise only two types of information - numbers and strings (a subject that is examined more closely in Chapter 7).

CLOSE# (abbreviation CLO.#)

The final operation needed to create a disk file once all of the data has been sent, is to 'close' it. The BBC BASIC command is

CLOSE#<channel>

As with OPENOUT, this is a command that provides the DFS with news. In this case the DFS is able to fully update the disk catalogue with details of how long this new file is, and to alter its record of free disk space accordingly. There are two variants to <channel> with this command (but no other!):

CLOSE#fo%

closes one file, namely that to which the channel variable 'fo%' refers. On the other hand,

CLOSE#0

will cause the DFS to close every file that happens to be open. The correct practice is to use the first variant to close just the single file - after all, it is a poor program that is unaware of which files it has opened! CLOSE#0 should only be used in extremis, normally as one part of the ON ERROR section of the program to ensure that the program is not terminated with any files left open.

As an example of what has been described so far, the following little program would read ten names and telephone numbers from the keyboard, and output them to a file called PHONE:

```
10 fo%=OPENOUT("PHONE")  <-- "Open" the file
20 FOR i%=1 TO 10
30 INPUT"Name",name$
40 PRINT#fo%,name$       <-- Write data to the file
50 INPUT"Telephone number",tel$
60 PRINT#fo%,tel$        <--
70 NEXT
80 CLOSE#fo%             <-- "Close" the file
90 END
```

Line 40 could be removed from this program by changing line 60 to

```
60 PRINT#fo%,name$,tel$
```

On obeying line 80, the CLOSE#fo%, file PHONE becomes a fully paid-up member of the disk catalogue. It can thus be the subject of any DFS command. One useful ploy, for instance, might be to add a line:

```
85 *ACCESS PHONE L
```

to lock the file as soon as it has been created.

5.2.2 File Input

Input of data from a serial file is also conducted in three stages - 'open' the file, read data from the file, 'close' the file.

OPENIN (abbreviation OP.)

To open a file for input, use is made of the BBC BASIC command,

<channel>=OPENIN(<filename>)

where <channel> and <filename> are used in precisely the same way as with OPENOUT. Similarly, all future commands relating to the data file must refer to <channel> for the purposes of identification since it is perfectly legal for more than one input file to be open at once. Some example calls of OPENIN are

```
fi%=OPENIN("INDATA")
fi%=OPENIN(":1.$.NAMES")
fi%=OPENIN(fnam$)
```

Throughout this book the <channel> variable 'fi%' is used to refer to a (f)ile that has been opened for (i)nput.

INPUT# (abbreviation I.#)

Data items may now be read in sequence from the file. The command used is

INPUT#<channel>,<list of variables>

Once again there are important differences between INPUT# and its keyboard equivalent INPUT. With the latter a string can be supplied to act as a prompt message to the user. The DFS needs no such message of course.

CLOSE# (abbreviation CLO.#)

As with output files, a file that has been opened for input has to be closed (even though the end of that file should have been reached). The same command forms apply as for file creation - either CLOSE#<channel> or CLOSE#0 can be used, with the arguments for using the former just as previously described. Note that CLOSE#0 is totally devoid of prejudice - it will close every file that is open whether it is a file that is being created or one that is being input.

The following program is the converse of the earlier example which illustrated file creation. It reads data from the file named PHONE which contains ten names and telephone numbers and outputs them to the screen:

```
10 fi%=OPENIN("PHONE") <-------------- 'open' the file
20 FOR i%=1 TO 10
30 INPUT#fi%,name$,tel$ <-------------- Read data from the file
40 PRINT"Name is: ";name$
50 PRINT"Telephone number is: ";tel$'
60 NEXT
70 CLOSE#fi% <-------------- 'close' the file
80 END
```

EOF# (no abbreviation)

With a program which takes repeated inputs from the keyboard, some check has to be made on whether or not the last data item has been met. For instance, a program can look for a particular value as a terminator value, or use some sort of counting process to decide when to stop asking for data. Both of these techniques can be used with disk files but both depend on the data in the file being as expected. What happens to a program which tries to read ten names from a file that contains only nine? The EOF# facility gives a foolproof way of checking for 'end of data'. Its form is

EOF#<channel>

where EOF# is actually a logical function which, while the file indicated by <channel> is in use, normally has the value FALSE. The function will only return the value TRUE when the end of the file has been reached. Thus a guaranteed way of reading a whole file - whether the expected number of data items exist or not - is to use a construct as follows:

Our program for reading names and telephone numbers from file PHONE could be rewritten to cope with any amount of data, rather than its fixed number of ten, in this way:

```
10 fi%=OPENIN("PHONE")
20 REPEAT
30 INPUT#fi%,name$,tel$
40 PRINT"Name is: ";name$
50 PRINT"Telephone number is: ";tel$'
60 UNTIL EOF#fi%
70 CLOSE#fi%
80 END
```

Repeated until end of file reached

For a fuller example of how a serial data file can be created and used, the reader is referred to Case Study 1 which describes an enhanced version of the example used in this chapter of a program for creating and maintaining a file of names and telephone numbers.

5.3 EXTENDING DATA FILES

5.3.1 The Problem

It has been mentioned already that one of the functions of a DFS is to manage the positioning of files on the disk. The user is concerned with storing information and quite rightly expects the DFS to worry about where to put it. This is fine when the disk is being used to hold programs. So long as the occasional *COMPACT operation is carried out the user is not likely to run into any problems at all. Unfortunately life is not quite so simple when it comes to holding data files on disk, for here difficulties could arise. The major source of trouble is the need to extend a data file.

A program may create a file on a single occasion, but on subsequent runs of the program may wish to add more data to this file, requiring in turn more disk space than was originally allocated. (Case Study 1 is a typical example in this respect.) Whether or not this will be possible is dependent on where the file is positioned on the disk.

In order to illuminate the problem let us consider the various steps taken during a couple of runs of the Case Study 1 program. It is not essential for the reader to have looked at the program to understand what follows, although of course it would help. The salient points are that the program initially creates a telephone directory file, and

thereafter further entries may be added to the file by using the program again. Concentrate now.

(a) The telephone directory file is created during the first run of the program. In this case OPENOUT causes the DFS to find a space on the disk which is 64 sectors in length. The first such area found determines the position of the file on the disk. The filename etc. is entered into the disk catalogue (see Fig. 5-1a). Note that a consequence of this approach is that the DFS will not create a new file if less than 64 sectors of disk space are available, even though only a couple of measly sectors might be all that is wanted!

(b) Data (in this case, names and telephone numbers) are input. At the end of this process all is sent to the disk file, which is then CLOSEd. It is at this point that the DFS, discovering that not all of the allocated 64 sectors of disk have been used, takes the surplus back again. The file entry in the disk catalogue shows that the file occupies the smallest whole number of sectors into which its data fits (Fig. 5-1b).

(c) Subsequently, having gained so many friends through owning a BBC Micro, the program is run again so as to extend the directory file. The file is input using arrays, and the new data added to these arrays in store. When OPENOUT is invoked this time, however, a file of the given name already exists. The action of the DFS in this case is not to look for 64 sectors, but to use the area previously occupied by the file in question, whatever its size. Since the new file needs more disk space than the old file, trouble may, but only may, occur; it depends on the positions of the other files on the disk.

Basically, the DFS is able to extend a file's disk space as far as the start of the next disk file. There are three possibilities:

1. There isn't a next disk file (Fig. 5-1c). This is a cause for much merriment and rejoicing, since it means that the file can be extended as far as the limits of the disk itself.

2. There is a gap of some sectors before the start of the next file is reached (Fig. 5-1d). All of this gap can be used to accommodate the extended file. Some merriment etc. here, depending upon whether or not the gap is big enough.

3. The next file starts in the sector immediately following the last sector of the file that hoped to be extended (Fig. 5-1e). Gloom and despondency prevail.

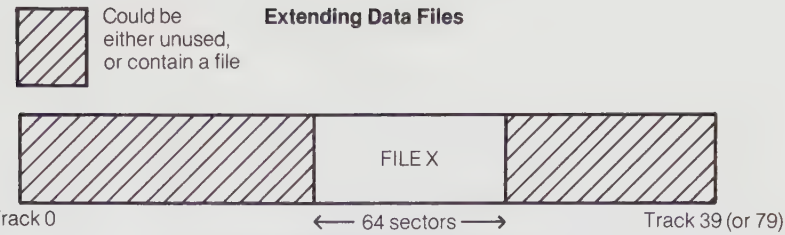


Fig. 5-1a After using fo% = OPENOUT ("X")

Fig. 5-1b After CLOSE # fo%—X only occupies sectors it needs

Fig. 5-1c File X can be extended to the end of the disk

Fig. 5-1d File X can be extended up to the start of File Y

Fig. 5-1e File X cannot be extended

If, when writing to a file that has been opened for output, the DFS finds that it has run out of room, it issues the error message (more of an error wail, really),

Can't extend

Unfortunately, by the time this message appears the damage has been done - the original file has been lost and part of a new file put in its place. Almost certainly the last item entered into the file has been truncated through lack of room, and this may well be sufficient to guarantee that the file cannot be used without some 'doctoring' (Chapter 7 will help here). Needless to say, this is a situation to be avoided.

5.3.2 Some Solutions

Basically, any solution must seek to guarantee that some free space will exist immediately following the end of the disk file that is to be extended. Three possibilities are outlined below; doubtless there are other approaches that would serve as well.

1. Only have one data file per disk! In this way there is bound to be room available for expansion unless the whole disk has been taken up. Obviously this is a solution which is only feasible if the data file is extremely large. For small files the cost and number of disks would be prohibitive.

2. Create the file so that it is of maximum size from the beginning, even though it may not be full of meaningful data. Basically, this involves the creation of a file containing what are usually termed 'dummy' records. The following little program, for instance, would implement this 'grab it now' policy by creating a file called PHONE containing 200 'dummy' names and telephone numbers:

```
10 name$=STRING$(15,"X"): tel$=STRING$(11,"0")
20 fo%=OPENOUT("PHONE")
30 FOR i%=1 TO 200
40 PRINT#fo%,name$,tel$
50 NEXT
60 CLOSE#fo%
70 END
```

Define record lengths

Write "dummy" record

This is a crude but quite effective approach. There is no guarantee of success of course, since the amount of disk space set aside for the future may still prove to be insufficient. And looking at things from the other direction, the space that has been reserved can no longer be used for anything else. A *COMPACT cannot be performed on the disk because the technique is really only creating a big hole into which a smaller file is being placed, and

*COMPACT's aim in life is to do away with such holes! It has to be accepted that a disk with a number of files with 'hoarded' space is being under-utilised.

3. The third solution aims to overcome the disadvantages of the previous two methods by ensuring that the file being extended is always the end file on the disk. This is achieved as follows:

   (a) A *COMPACT operation is performed on the disk before the program runs. This maximises the amount of free space available at the end of the disk. Now this operation can be carried out by the user explicitly before the program is run - but it might be forgotten. One way of making sure that the disk is compacted every time is to put the necessary instructions into the program itself:

```
1 page%=PAGE: PAGE=TOP+256
2 *COMPACT
3 PAGE=page%
```

   The setting of PAGE has to be stored and restored because, as the reader may remember from Chapter 4, the *COMPACT function uses the BBC Micro RAM as a temporary storage area. Setting PAGE to TOP+256 makes sure that the program (and the variable page%) are not destroyed.

   (b) The program is altered so that it will generate a file with a different name to that of the existing file. The DFS will then automatically locate that file at the start of the free area on the disk. The simplest way of producing a name which is different but easily remembered, is to use a special directory identifier. For instance, changing the OPENOUT statement of our "PHONE" program to

```
20 fo%=OPENOUT("Z.PHONE")
```

   will open the output file PHONE in directory Z, and any extended version of the file will be sent to it.

   (c) At the end of the program run the user then has to do no more than type

```
*DELETE PHONE
*RENAME Z.PHONE PHONE
```

   to resume the original naming convention.

5.4 TRANSFERRING DATA FILES FROM TAPE TO DISK

In Chapter 2 a description of how to transfer programs from tape to disk was given. Now, having become a firm believer in disk systems, the erstwhile tape user may well want a number of existing taped data files to make the same journey. There are a couple of methods available. The first, and perhaps least attractive, is to write a special purpose tape-to-disk transfer program for the data file in question. Something along the lines of the following would serve for a file (called PHONE) of names and telephone numbers previously created on tape:

```
 10 *TAPE
 20 tape%=OPENIN("PHONE")      <-- 'Open' tape and disk files
 30 *DISK
 40 disk%=OPENOUT("PHONE")     <--
 50 REPEAT
 60   *TAPE
 70   INPUT#tape%,name$,tel$   <-- Read tape record
 80   *DISK
 90   PRINT#disk%,name$,tel$   <-- Write this record to disk
100 UNTIL EOF#tape%
110 CLOSE#disk%               <--
120 *TAPE
130 CLOSE#tape%               <-- 'Close' tape and disk files
140 END
```

The general technique should be apparent - the program switches between tape and disk filing systems as it conducts its input and output respectively. By writing little programs in the same rather inelegant vein, data files can be copied from one medium to the other.

*LOAD (abbreviation *L.)
*SAVE (abbreviation *S.)

A rather neater method can be used, however. The only condition as to its use is that the whole of the file to be transferred must be small enough to fit into the available RAM. Given that in Mode 7 this means files in excess of 22000 bytes in length, there will be few occasions when the following approach cannot be employed.

Two commands exist in BBC BASIC, primarily for the loading and saving of machine code programs:

*LOAD <filename> <load address>

will load the file called <filename> into store from position <load address> onwards. If <load address> is not specified, the file is loaded at whatever store position is given in the <load address> element of the file's catalogue entry. The counterpart to *LOAD is

*SAVE <filename> <start address> <finish address>
(<execution address>)

and its effect will be for the contents of RAM between addresses <start> and <finish> to be written to the backing store device as a file called <filename>. An alternative to <finish address> is +<no. of bytes>. The <execution address> if given, states the position at which the computer must begin obeying the program when it is run. The value is often omitted, being then assumed to be the same as <start address>.

Now because these commands are concerned with the plain transfer of bytes of information between RAM and backing store, and do not inspect the data in any way (unlike LOAD/SAVE, which expect file and RAM contents to be in the form of a BASIC program) they can equally well be used with data files as with machine code programs. The general method is as follows:

*TAPE
*LOAD <filename> <start>

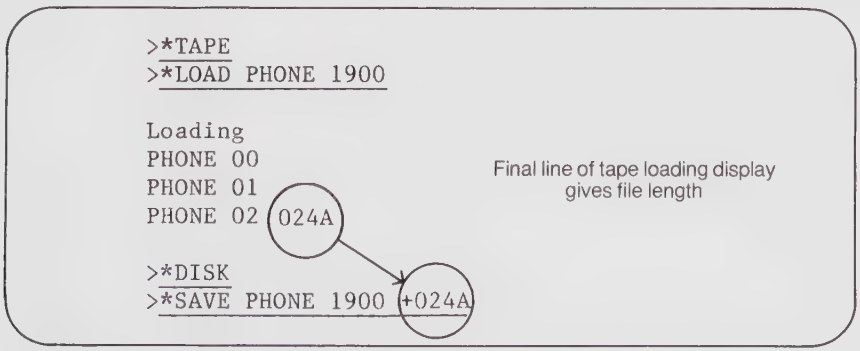
This will cause the file <filename> to be brought into store at location <start> onwards. The usual tape loading messages will appear, finishing with one of the form,

<filename> <no. of blocks> <no. of bytes>

This <no. of bytes> value must be noted since it is needed in the next step, which is

*DISK
*SAVE <filename> <start> +<no. of bytes>

causing the transfer to file <filename> of <no. of bytes> bytes from RAM position <start> onwards. That's all there is to it! By way of an example, the dialogue to transfer a file called PHONE as before might be



Here <start> is set as location 1900 (hexadecimal), which is the first

free location within a BBC Micro which has a disk interface fitted. Naturally any value above this could be used but this would reduce the size of the file that could be transferred.

This technique, although described for tape-to-disk transfers, is generally applicable. It could be used for copying any file from disk-to-tape, tape-to-tape, or disk-to-disk. The important thing in every case is to know how long the file is. The technique for tape files has been shown; for a disk file *INFO is used to discover the file's extent. Bearing in mind the value of taking backup copies, why not use tapes as inexpensive disk copies? With luck they will not be needed - but you never know!

5.5 SCREEN DUMPS

As we have seen *SAVE actually transfers the contents of an area of RAM to disk, with *LOAD performing the reverse operation. Now one particular area of RAM which could be saved in its own right is that which holds the current screen display. By saving this graphics area on disk, displays which have been produced by a program can quickly be retrieved without the need to run the program again - a facility of much value in the classroom.

The technique is simple. At an appropriate point in the program producing the graphics display, or else keyed in when the program has finished, use a statement like

*SAVE <filename> <start address> <finish address>

The value for <finish address> will be 8000 for a BBC Model B, 4000 for a Model A. The value for <start address> depends on the screen mode in use:

MODE	Model B	Model A
0, 1, 2	3000	----
3	4000	----
4, 5	5800	1800
6	6000	2000
7	7C00	3C00

For example, to save a Mode 3 display in file SCREEN:

*SAVE SCREEN 4000 8000

To retrieve the display, ensure that the correct screen mode is set, and then use the command

*LOAD <filename>

This will automatically load the file into the correct graphics area of RAM, resulting in an immediate display of the file contents. Amazing!

6 RANDOM ACCESS FILES – Part 1

"I remember you saying you had notions of a good genius presiding over you. I have of late had the same thought - for things which I do half at random are afterwards confirmed by my judgement in a dozen different features of propriety."
John Keats, Letters (to B. R. Haydon)

6.1 RANDOM ACCESS FILES

To refer to the Disk Filing System of the BBC Micro as a 'good genius' is rather going overboard but, being a poetic sort of chap, Keats was rather inclined to write that way. At least he had started to discover something of the potential of his disk system by investigating, albeit half-heartedly, the use of random access files.

The characteristic of a random access file is that it is a file whose elements are to be used not sequentially, as with a serial file, but in any order at all. With such a file individual items can be input and output without the necessity to read or write the whole file. Stated as baldly as this the reader may be tempted to venture an opinion along the lines of 'so what?'. This would be a premature and narrow judgement. The facility for random access files, used well, is a powerful tool indeed. To own a disk system and deal solely with serial files is rather like buying a Rolls-Royce and then pushing it everywhere.

6.2 PROGRAMMING FOR RANDOM ACCESS FILES

6.2.1 The File Pointer

It has been mentioned earlier that the disk itself can be randomly accessed by the Disk Filing System; indeed this is a fundamental aspect of the whole business. Now when introducing this feature, reference was also made to the need for a disk addressing capability - that is, for a way in which a particular sector of the disk could be nominated as that to be accessed next. A similar principle is involved with random access files, except that the level of control has to be finer - a program must be able to specify which byte of a file is to be used next. This level of control is provided by what is known as the 'file pointer'.

When any file is opened, the pointer for that file is set by the DFS to indicate the first byte of that file. Thereafter, when any file operation takes place, this pointer is updated by the DFS so that it points to the next data item in the file. It is by readjusting this pointer that a BBC BASIC program can select a different data item to that which would normally be next in sequence, and thereby effect a random access to that file.

Now this presupposes that the program can discover exactly where in the file its required data item resides - for how else can that program set the pointer? Either an exact position must be known, or else some other information which enables an exact position to be calculated. This in turn means that the would-be user of a random access file must have some idea of how data items themselves are represented on disk. Briefly, since the next chapter looks at this topic in more detail, the relevant facts are as follows:

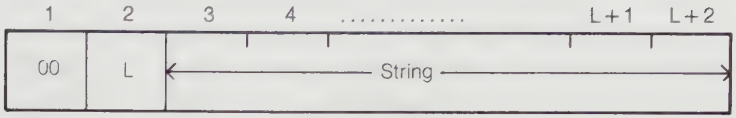
STRINGS (Fig. 6-1a)
How long is a piece of string? A string will occupy one byte for every character it contains, the characters being written to the disk in reverse order. In addition the character bytes are preceded by two other bytes. The first is always 00; the second is a value in the range 0-255 which specifies the number of characters in the string.

INTEGERS (Fig. 6-1b)
An integer value will always occupy five bytes, the first of which is the byte 40(hex).

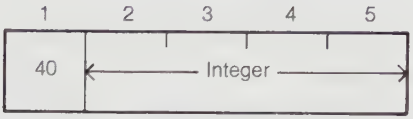
REAL NUMBERS (Fig. 6-1c)
A real number will always occupy six bytes, the first of which is the byte FF (hex).

Thus if the first two data items in our file PHONE were the strings ANNE and 424-5647 then the actual content of the opening bytes of the file would be as shown in Fig. 6-2a. When the file is opened the file pointer is initialised to zero, i.e. it is pointing to the

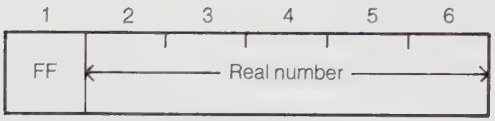
Fig. 6 Data representations on disk



6-1a String



6-1b Integer



6-1c Real

start of the first data item. As a program reads data items from the file, the pointer is continually updated (see Fig. 6-2b) so as to point to the next item in sequence.

PTR# (abbreviation PT.#)

The fundamental aspect of programming with random access files is that this file pointer may be referenced and changed by the program. In BBC BASIC the file pointer can be used by means of the variable

PTR#<channel>

where <channel> is that defined when the file is opened. It is by altering the value of this variable between input or output statements that enables a file to be randomly accessed. For instance, if, before any input takes place, the pointer were to be set to 16 by using the statement

PTR#<channel>=16

then the first input statement would read the data item which starts at byte 16 of the file. In our example (see Fig. 6-2c) this would serve to reposition the file pointer so as to cause the first INPUT# statement to read the second name in the file rather than the first. Further items could be input in order, if required, without another change to PTR#. Similarly, decreasing PTR# would position the pointer so that earlier items in the file could be input (or overwritten).

At this point it is worth emphasising three things, each of which

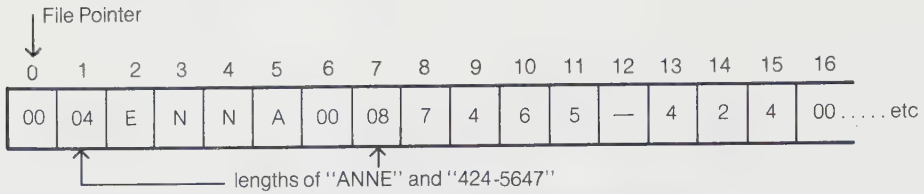


Fig. 6-2a Opening bytes of name/telephone number file

Fig. 6-2b Updating of file pointer after reading "ANNE"

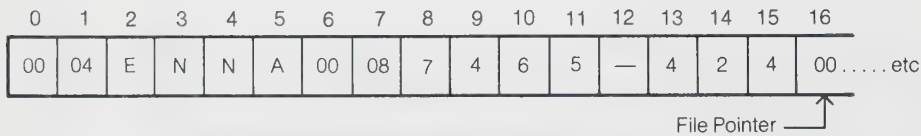


Fig. 6-2c Setting of file pointer after PTR # fo% = 16

follows on from what has just been said:

1. The input instruction and the data item which it is to input from disk have to correspond. An input instruction which is expecting a string will not get very far if the file pointer has been moved to the start of a real variable.

2. Similarly, but just as bad, any input instruction will fail if the pointer is not positioned at the first byte of a data item. That byte - 00, 40 or FF - is defining the type of item that follows. A random placement of the file pointer somewhere in the middle of the item will either fail miserably or, if the byte pointed to happens to be one of the three valid start bytes, will produce spurious results. In the end both of these instances, and that of 1. above, will evoke the error message

   Type mismatch

   indicating that the input statement has not encountered the type of data item it had set its heart on.

3. When overwriting the data item at position PTR#, the program must be aware of exactly how much space is available. For instance, a real value cannot overwrite an integer nor a string be replaced by one of a greater length. Remember, we are not dealing with a serial file which is to be completely rewritten to disk. With a random access file individual items are being altered and, if the new item takes more space than the old, data will be corrupted.

These points should be borne in mind as we cover the steps involved in using a random access file: creation and update.

6.2.2 Creating a Random Access File

A random access file is created by using the same commands in the same sequence as introduced in the discussion on creating a serial file:

OPENOUT - to open the file for output;
PRINT# - to send data to the file;
CLOSE# - to close the file when all data sent.

However, there are significant differences between the two processes, even though the commands employed may be the same. Firstly, since it is essential with a random access file that it be possible to locate the start of any particular data item, those items will commonly be output with a fixed length. The term 'fixed-length record' is often used in this context. With this type of organisation the file pointer can thus be adjusted by a known amount so as to reference data items in the file. For instance, with our telephone directory it might be decided that a limitation be imposed on the data such that

* a name can be a maximum of 15 characters, and
* a telephone number a maximum of 11 characters.

Given that a string needs two additional bytes when stored on disk, these two data items together will therefore require 30 bytes in all. Thus names in the file start at byte positions 0, 30, 60, 90 and so on. With this regular pattern to work with it is simple for a program to calculate the value for PTR# so as to access any particular entry.

As an example, let us consider again our simple program to create a file of names and telephone numbers. This time the program will create a file for random access. Each name will occupy a fixed-length record of 15 characters (17 disk bytes) and each telephone number a fixed-length record of 11 characters (13 disk bytes). As an added refinement, any number of data items may be input.

```
 10 fo%=OPENOUT("PHONE")
 20 INPUT"How many data pairs",N%
 30 FOR i%=1 TO N%
 40 INPUT"Name",name$
 50 PRINT#fo%,name$: PTR#fo%=i%*17 <-------
 60 INPUT"Telephone number",tel$
 70 PRINT#fo%,tel$: PTR#fo%=i%*30 <-------
 80 NEXT
 90 CLOSE#fo%
100 END
```

Reset file pointer to start of next field

Note in particular the way in which PTR#fo% is calculated (lines 50 and 70) so as to point to the start position for the next data item to be output to the file.

Now - a problem. A corollary to the idea of fixed-length records is that the length fixing must be rigidly imposed by the program which is creating the file. If data items are being input by the user, as will most often be the case, they have to be constrained to the appropriate length before being written to the file - the user certainly cannot be relied upon to input data that is within the limits. In our example above, lines 50 and 70 should be altered to

```
50 PRINT#fo%,LEFT$(name$,15): PTR#fo%=i%*17
```
and
```
70 PRINT#fo%,LEFT$(tel$,11): PTR#fo%=i%*30
```

to ensure that the string name$ does not exceed its allowed 15 characters on the disk, and tel$ its allotted 11 characters, no matter how many had been input by the user.

Once created, a random access file is not expected to need an extension of its allocated area on disk (but see Section 6.3 below). The reason for this is that such a file is subsequently going to have its data items accessed and updated individually - it is not usual for the whole file to be written to disk again. Thus the process of creation will aim to reserve the whole disk area needed for the file. This is achieved in one of two ways:

1. By writing a series of dummy records to the file, the intention being that these data items will be replaced when the file is updated later. Filling our telephone directory file with additional dummy data items so as to reserve enough space for 200 names and numbers could be carried out by extending our program as follows:

```
 10 fo%=OPENOUT("PHONE")
 20 INPUT"How many data pairs",N%
 30 FOR i%=1 TO N%
```

```
 40 INPUT"Name",name$
 50 PRINT#fo%,LEFT$(name$,15): PTR#fo%=i%*17 <-------- Write defined
 60 INPUT"Telephone number",tel$                        records
 70 PRINT#fo%,LEFT$(tel$,11): PTR#fo%=i%*30 <----------
 80 NEXT
 90 FOR i%=N%+1 TO 200
100 PRINT#fo%,STRING$(15,"*") <-------- Write 'dummy' records
110 PRINT#fo%,STRING$(11,"-") <--------
120 NEXT
130 CLOSE#fo%
140 END
```

(Note: if you have just tried the previous program, remember to delete file PHONE before running this one, or else a 'Can't Extend' error might occur - see 5.3 for the reason why.)

2. Alternatively, the *SAVE command could be used. A file for 200 name and number records would require 200*30=6000 bytes. In hexadecimal which *SAVE likes to speak, this means 1770 bytes. (Conversion is simple: type PRINT ~6000 in immediate mode and the answer pops up.)

```
*SAVE PHONE 0 1770
```

would create a file of the required length. It would contain junk (whatever the contents of RAM locations 0 to &1770 happen to be) but since the aim is to overwrite this rubbish with real data at some later stage, this is of no consequence.

6.2.3 Reading from a Random Access File

The command sequence used to read from a random access file again bears a marked resemblance to that used with serial files:

OPENIN - to open the file for input;
INPUT# - to read data from the file;
CLOSE# - to close the file at the end of input.

Remember, though, the way in which the data in a random access file is organised. Data items do not follow end-on, but are located at positions to which the file pointer will be directed. The input part of the sequence, therefore, will necessarily need to set PTR# before trying to read a data item. For instance, the following program would read name/telephone number pairs from the file set up in the example of the previous section, and output each to the screen:

```
 10 fi%=OPENIN("PHONE")
 20 i%=1
 30 REPEAT
```

```
 40 INPUT#fi%,name$: PTR#fi%=i%*17
 50 INPUT#fi%,tel$: PTR#fi%=i%*30
 60 PRINT"Name is: ",name$
 70 PRINT"Telephone number is: ";tel$'
 80 i%=i%+1
 90 UNTIL name$=STRING$(15,"*")
100 CLOSE#fi%
110 END
```

Reset file pointer to start of next field

Input records until dummy record found

Here the file pointer is first set on the name to be input. Since this string may not occupy its complete quota of 15 characters, the pointer must be adjusted so as to reference the next string in sequence, the telephone number, before another INPUT# is performed. It then has to be adjusted again to point to the start of the next name, and so on, until the first dummy name is input.

6.2.4 Updating a Random Access File

'Updating' is what random access files are really all about. The term means writing information to a random access file so as to replace what is already there, a creation program of some form having set the file up in the first place. Once a random access file has been created, the usual operation performed upon its contents is that of updating, which involves both input and output.

OPENUP (abbreviation OPENU.)

Now to open a file for updating is rather different to opening it for output. The OPENOUT command cannot be used since its first step is always to delete any existing file with the given filename on the assumption that a whole new file is to be produced. With updating this is manifestly not the case. Here another command is needed.

<channel>=OPENUP(<filename>)

will open the file with name <filename> for both reading and writing, i.e. updating. The variable <channel> is used with both INPUT# and PRINT# statements which subsequently reference this file.

A minor digression is called for at this point. The OPENUP command is only available on those BBC Microcomputers which have the version of BBC BASIC known as BASIC II. Machines which have the original BASIC will refuse to recognise OPENUP as a valid command. In this latter case the command OPENIN actually opens a file for updating rather than for just plain input. Now this places your author in something of a quandary - do the examples use OPENUP or OPENIN? Either way a source of confusion will exist for a number of readers. In the hope that it is not too off-putting for those BBC Micro users who have BASIC I, this

book will use the OPENUP command in its examples and Case Studies. However, the statements will always be of the form,

fu%=OPENUP(<filename>): REM Use OPENIN with BASIC I

Use of <channel> fu% rather than fi% will serve to indicate that the file being used is open for updating rather than input, and the REM aims to act as a reminder that the actual statement needs to be changed if the program is to run on a BBC Micro which has BASIC I.

In answer to your question, dear reader: to conduct the exciting investigation into which version of BBC BASIC your computer possesses, press BREAK and then type the command

REPORT

The effect will be to produce a message of the form,

(C)<date> Acorn

If <date> is 1982 then you have BASIC II and OPENUP is available for use; 1981, and your machine contains the original BASIC and you must use OPENIN to open a random access file for updating. A <date> for any year later than 1982 means that this book is getting rather long in the tooth. Write to Messrs. Prentice-Hall immediately, suggesting that they commission the author to produce a second Edition! End of digression.

To return to the business at hand. Having opened a random access file for updating, either INPUT# or PRINT# can be used to either read from or write to the file at the position given by the current value of PTR#. A typical requirement with "PHONE" file would be to change the recorded number if its particular name happens to move house. This can be achieved with the following sort of program:

```
 10 fu%=OPENUP("PHONE"): REM Use OPENIN with BASIC I
 20 INPUT"Whom do you seek",seek$
 30 i%=-1
 40 REPEAT
 50 i%=i%+1: PTR#fu%=i%*30
 60 INPUT#fu%,name$
 70 UNTIL name$=seek$
 80 INPUT"What is the new telephone number",tel$
 90 PTR#fu%=i%*30+17: PRINT#fu%,tel$ <------ Overwrite current
                                             record on disk
100 CLOSE#fu%
110 END
```

One more point about updating. A program should avoid moving PTR# beyond the end of the file by incorporating suitable checks. Our example program above, for instance, should really ensure that it does not try to read beyond the end of the file if it is asked to search for a name that does not actually exist.

EXT# (no abbreviation)

Now the logical function EOF# has been introduced, but is of little use in this instance since the value of EOF# only becomes TRUE when PTR# has reached the end of the file. What is required is a means of checking for this before PTR# is altered. The variable,

EXT#<channel>

contains the address of the last byte in the file to which it refers. Thus if our example program has its line 50 changed to the rather longer

```
50 i%=i%+1: IF i%*30<EXT#fu% THEN PTR#fu%=i%*30
    ELSE PRINT"Name not found": CLOSE#fu%: END
```

then the trauma of falling off the end of a random access file can be avoided.

6.3 EXTENDING RANDOM ACCESS FILES

When all is said and done, programmers are only human. With the best will in the world a random access file can be created, seemingly with acres of room to use, only to find that more is needed. If this does prove to be the case, the safest way out of the situation is to create a larger file and copy the existing file across to it. If the file will fit into the normally available RAM, the *LOAD/*SAVE combination can be used again. To extend the telephone directory file, for instance, so that it occupies 3000 (hex) bytes rather than its original amount,

(i) Use *INFO to find out how big the existing file is, as a guide to how much bigger to make it!

```
*INFO PHONE
```

might produce for instance,

```
$.PHONE xxxxxx xxxxxx 001775 xxx
```

showing that the file currently occupies 1775 bytes.

(ii) Then *LOAD the existing file into a stated position in store:

```
*LOAD PHONE 1900
```

(iii) Use *SAVE to output from the store position to a new file, but stipulating an increased number of locations for transfer:

```
*SAVE PHONE 1900 +3000
```

This will give an extended file, but the data beyond the last item of the original file will be totally random

For a fuller example of how a random access file can be created and used, the reader is referred to Case Study 2, which builds on the examples of this chapter to develop a program to handle a name/telephone number random access file.

7 DIAGNOSTIC AIDS

"We are always doing", says he, "something for Posterity, but I would fain see Posterity do something for us."
Joseph Addison, The Spectator

It may be, as Addison seemed to feel, that a good disk file should live forever; that it could perhaps become an heirloom, handed down from generation to generation along with the family silver. A bequest of one's complete collection of floppy disks will only be truly appreciated, though, if the files that they contain are error-free. Addison's complaint appears to be that, with his disk system, there was no easy way of finding out just what his newly-created data files contained. The user of the BBC Micro is in a much more fortunate position.

7.1 FILE DUMPS

*DUMP (abbreviation *DU.)

Any file can be examined in detail by using the utility command,

*DUMP <filename>

The effect of this command is to list in a rather low-level form the contents of the specified disk file. It is perhaps easier to describe the form of the output by looking, as an example, at a particular random access file XDIRECT. What follows is based on a file which contains names and telephone numbers in fixed-length fields of 17 and 13 disk bytes respectively. Unused byte positions in these fields are filled with the '?' character for the name field, and the '!' character for the telephone number field. The first item of the file is an integer value, which is set to the number of data pairs stored in the file. (For those that have looked ahead, the file is actually that created in the sample dialogue of Case Study 2.)

```
>*DUMP RDIRECT
```

produces the following listing:

```
0000 40 00 00 00 05 00 04 45 @......E
0008 4E 4E 41 3F 3F 3F 3F 3F NNA?????
0010 3F 3F 3F 3F 3F 3F 00 0B ??????..
0018 37 34 36 35 2D 34 32 34 7465-424
0020 2D 31 30 00 08 54 45 52 -10..TER
0028 41 47 52 41 4D 3F 3F 3F AGRAM???
0030 3F 3F 3F 3F 00 0B 35 34 ????..54
0038 36 35 38 39 2D 31 33 32 6589-132
0040 30 00 09 48 54 45 42 41 0..HTEBA
0048 5A 49 4C 45 3F 3F 3F 3F ZILE????
0050 3F 3F 00 0B 37 36 38 39 ??..7689
0058 2D 35 34 35 2D 31 30 00 -545-10.
0060 0F 53 45 4C 52 41 48 43 .SELRAHC
0068 20 26 20 41 4E 41 49 44  & ANAID
0070 00 0B 72 6F 74 63 65 72 ..rotcer
0078 69 44 2D 78 45 00 0F 43 iD-xE..C
0080 20 59 52 41 4D 20 52 45  YRAM RE
0088 46 49 4E 4E 45 4A 00 0B FINNEJ..
0090 32 36 37 38 39 38 2D 30 267898-0
0098 39 32 33 ** ** ** ** ** 923.....
```

This output, termed a 'file dump', comprises a number of rows, each of which conforms to the format,

<byte address> <8 bytes, hexadecimal> <8 bytes, ASCII>

Referring to the ninth line of the dump, for instance,

```
0040 30 00 09 48 54 45 42 41 0..HTEBA
```

the <byte address> is 0040. All such addresses are given in hexadecimal and apply specifically to the file in question. That is, they are not absolute disk sector addresses, but addresses which are relative to the start of the file. In particular, they are the values to which PTR# would need to be set if the file were to be used as a random access file. The address is followed by <8 bytes, hexadecimal>:

```
30 00 09 48 54 45 42 41
```

where the first of the bytes (30) is that held at address 0040, the next (00) that held at address 0041, and so on up to the last byte in the row (41) held at address 0047. These bytes are also given using hexadecimal

notation, and can thus range from 00 to FF.

The third part of the row, <8 bytes, ASCII>, presents the contents of the same 8 bytes of the disk file, but in this case showing them as ASCII characters:

```
0..HTEBA
```

That is, the contents of any particular byte is taken to be an ASCII character code, and the appropriate character displayed. (The full set of ASCII codes is given in Appendix A.) So with this example:

```
0040 30 00 09 48 54 45 42 41 0..HTEBA
```

it can be seen that byte 0040 contains the hexadecimal value 30 which, if interpreted as an ASCII code, represents the character '0'. Byte 0047 contains the value 41, or ASCII character 'A'. Notice, however, the character outputs for bytes 0041 and 0042. These bytes contain values (00 and 09) which do not translate into printable ASCII characters. In these, and all such instances, the byte value is shown as a '.' in the ASCII part of the output line.

7.2 INTERPRETING FILE DUMPS

In Chapter 6 the subject of data representation on disk was introduced and the characteristics of integer, real and string values differentiated. By referring to the example dump given above, that discussion can now be clearly illustrated.

An integer value for instance, can be identified in the first line of the dump:

```
0000 40 00 00 00 05 00 04 45 @......E
```

The integer, value 5, occupies bytes 0000 to 0004 inclusive. It is discerned by the value 40 as the opening byte.

String representations occur throughout the dump. For example,

```
0040 30 00 09 48 54 45 42 41 0..HTEBA
0048 5A 49 4C 45 3F 3F 3F 3F ZILE????
0050 3F 3F 00 0B 37 36 38 39 ??..7689
```

shows the string ELIZABETH starting at byte 0041, with the introductory byte 00. The next byte, at 0042, is the value which indicates the length of the string - nine characters in this case. Bytes 0043 to 004B contain the ASCII codes for the characters of this string, held in reverse order.

The next string, 01-545-9867, starts at byte 0052. What then of the bytes between these two strings - 004C to 0051?

Well this is a clear example of what will happen when a 'dummy' record is subsequently overwritten by a 'real' record. Here a dummy

record of 15 '?' characters was initially written to addresses 0043 to 0051. When, subsequently, this slot was updated to contain the name ELIZABETH, only the new characters replace the original '?' characters. This means that six of them, in addresses 004C to 0051, remain unchanged. There is no problem about this since the length of the new entry only reflects the valid characters, and omits those at the end of the fixed-length field. However, there is no doubt that, until one becomes accustomed to this technique, file dumps can appear to be rather messy and at first quite confusing to interpret.

The end of the file can also be seen quite clearly. In the example that has been used in this chapter the final line of the dump is

```
0098 32 33 30 ** ** ** ** ** 230.....
```

The ** entries in the dump indicate that those bytes are beyond the recorded end of the file. In this case, therefore, the last byte of the file is at address 009A. That this is in fact the case can be seen by using *INFO to obtain the catalogue details for this file:

```
>*INFO RDIRECT
```

would be expected to produce the output,

```
$.RDIRECT   xxxxxx  xxxxxx  00009B  xxx
```

indicating that file 'RDIRECT' occupies 9B bytes - namely bytes 0000 to 009A inclusive.

7.3 TRACING FILE ACCESSES

*OPT1 (abbreviation *O.1)

Another feature which may be of some use in helping the user to remove the nasty habits of a wayward program is the *OPT1 command. Used in the simple form,

```
*OPT1,1
```

it causes the Disk Filing System to report file accesses - the output generated is exactly as produced by the *INFO command. Thus by judicious use of *OPT1 the user should be able to determine that his program is communicating with a file at the right times and in the right places. For instance, if *OPT1,1 was used before calling the Case Study 2 program RTELLY, the opening lines of the dialogue would be as follows:

```
>*OPT1,1
>CHAIN "RTELLY"
$.RTELLY  xxxxxx  xxxxxx  xxxxxx  xxx     (*OPT1 output)

ooooooooooooooooooooooooooooooooooooooooooScreen Cleared

CREATE a New File (Y or N)?Y

Name of File to be CREATED?TEMP
$.TEMP    xxxxxx  xxxxxx  004000  xxx     (*OPT1 output)

Input data items (RETURN to finish)  ............ and so on.
```

To turn this tracing facility off requires nothing more than

*OPT1,0

either input at the keyboard or as a line in the user's BASIC program.

7.4 PATCHING DISK DATA FILES

HEALTH WARNING - 'Patching' can seriously damage file health.

One of the facts of life with data files, be they on disk or tape, is that they need to be altered if created or updated incorrectly. For a tape file this always means that the erroneous file has to be totally reconstructed. This need not be the case with a disk file. Here, using the detailed information provided by a file dump together with the capability for random access to any byte position, it is possible for a disk file to be 'patched'.

Now a file patch means just what it says - it is a way of covering over a particular byte or group of bytes. That is to say, a file patch is the replacement of some existing bytes by new values. An expansion or contraction of the file by either inserting or deleting bytes is not usually undertaken by this technique.

BPUT# (abbreviation BP.#)
BGET# (abbreviation B.#)

Two BBC BASIC commands are particularly important in this context. Both operate on a single byte, either a constant or a variable; in both cases the numerical value is restricted to the range 0 - 255.

BPUT#<channel>,<byte>

will transmit the value <byte> to the disk file to which <channel> refers. The byte is written to the disk address indicated by the file

pointer PTR#, which is then incremented by 1. For example,

```
BPUT#fu%,&40
BPUT#fu%,byte%
```

The converse command in BBC BASIC is

```
<variable>=BGET#<channel>
```

which causes input of the byte at the address given by the file pointer. This byte has to be input to a variable, which should really be an integer variable. After the input, PTR# is incremented by 1.

Thus a very crude way of patching a disk file would be to use these commands in immediate mode. For instance, to change the byte at address &45 in file FRED so that it contained &20 (i.e. ASCII for a space) could be achieved in the following way:

```
>fu%=OPENUP("FRED"): REM Use OPENIN with BASIC I
>PTR#=&45
>byte%=BGET#fu%: PRINT~byte%: REM as a check!!
>PTR#=45: BPUT#fu%,&20
>CLOSE#fu%
```

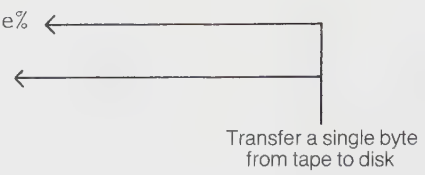
Notice that it is perfectly possible to write integers, real values and strings to a disk using the same sort of method. If the BPUT# command in the above example were replaced by, say, PRINT#fu%,"HELLO MUM" then the whole string would be written to the file from address &45 onwards. However, this technique is not to be recommended to the unwary since it is obviously fraught with danger.

If file patching is to be undertaken, then a much better approach is to use a purpose-built program for the job. By an amazing coincidence Case Study 3, which can be found in Section II of this book, is just such a program.

7.5 COPYING FROM TAPE TO DISK - AGAIN

Another use for BGET# and BPUT# is with a tape to disk copy program which uses virtually no additional RAM above that needed for the program itself. Although a relatively slow method for copying, this might be preferable to writing a special-purpose transfer program for a very large file. The program simply reads a single byte from tape and transfers it to disk, this operation being repeated until the end of the tape file is reached:

```
 10 INPUT"Filename",fnam$
 20 *TAPE
 30 tape%=OPENIN(fnam$)
 40 *DISK
 50 disk%=OPENOUT(fnam$)
 60 REPEAT
 70   *TAPE
 80   byte%=BGET#tape%
 90   *DISK
100   BPUT#fo%,disk%
110 UNTIL EOF#tape%
120 *TAPE
130 CLOSE#tape%
140 *DISK
150 CLOSE#disk%
160 END
```



One additional point needs to be made about this program, and about the use of BGET# and BPUT# in general. Neither tape nor disk can ever be used to physically transfer an individual byte of information. They work in terms of blocks of data, typically 256 bytes comprising a single block. Thus when a file is being used for input, a whole block of the file must be read into RAM; if the file is being used for output, a whole block must be sent from RAM. At any time then a file which has been opened will require a certain amount of RAM to be allocated to hold a block of data for use. (The extra space that a Disk Filing System takes up is mainly for this reason.) This area of RAM is termed a 'file buffer', and it is one of the functions of the appropriate filing system to deal with the mechanics of managing these buffers. Any input or output statements, for instance, will cause the filing systems to access the relevant buffer. When the end of the buffer is reached, the filing system must automatically cause the next block of data to be input or, for output files, the current block of data to be written.

So it is that functions like BGET# and BPUT# can be provided, even though the actual device - tape recorder or disk drive - will not operate in terms of individual bytes. When BGET# is invoked, a byte is taken from the tape file buffer, a new block being read when the buffer becomes empty. (If too much time elapses before the next block is read from tape it will be essential for the tape recorder to have a motor control.) A call of BPUT# causes the byte to put into the disk buffer, the whole buffer being sent to disk when it is full. This is another reason for remembering to CLOSE# the file, since it forces the final, half full, buffer to be output.

8 EXECUTIVE FILES AND LIBRARIES

"Words are also actions, and actions are a kind of words."
Ralph Waldo Emerson, The Poet

Highly competent programmer though he undoubtedly became, Emerson still had a habit of reverting to his previous trade as a philosopher when talking about computing topics. The above is a classic example. What the chap is really getting at, in a profound sort of way, is this: 'Not only is it possible to create files which contain data, but also to create files which contain commands.' These 'Executive' files, or EXEC files for short, when invoked, pass their contents to the BBC Micro to obey just as if those contents had been typed in at the keyboard.

In this chapter we shall be looking at some of the different ways in which EXEC files might be used and looking at the ideas of file and program libraries. Following the sort of precedent that good old Ralph Waldo might have appreciated, however, we begin with Creation.

8.1 CREATING AN EXEC FILE

***BUILD (abbreviation *BU.)**

An EXEC file is most simply created by using a utility command designed specifically for this purpose:

*BUILD <filename>

will allow the user to input a number of lines to the specified file, prompting him each time with a line number. For instance, creating an EXEC file which contains commands to set up some of the red function keys on the BBC Micro might proceed thus:

```
>*BUILD SKEY
  1 *KEY0  MODE 7 |M
  2 *KEY1  RUN |M
  3 *KEY2  LIST |M
  4   <---------(Press ESCAPE key to finish)
>
```

ESCAPE also closes the file

The file SKEY in this example is created (effectively an OPENOUT performed by the DFS) when *BUILD is used. The file is closed when ESCAPE is pressed. In between, the action of the *BUILD command is to issue prompts (1, 2, 3, etc.) and to faithfully transcribe to the disk file whatever is typed in return. It is useful to have a look at just what is sent to the file:

```
>*DUMP SKEY

0000  2A 4B 45 59 30 20 4D 4F  *KEY0 MO
0008  44 45 20 37 7C 4D 0D 2A  DE 7|M.*
0010  4B 45 59 31 20 52 55 4E  KEY1 RUN
0018  20 7C 4D 0D 2A 4B 45 59   |M.*KEY
0020  32 20 4C 49 53 54 20 7C  2 LIST |
0028  4D 0D ** ** ** ** ** **  M.......
```

ASCII characters

Notice in particular:

1. The file does NOT contain character strings in the form that was discussed in the previous chapter. It holds the ASCII characters, certainly, but as a straight copy of precisely what was input from the keyboard. This should be no great surprise, since, after all, the file is subsequently going to be used as a replacement for keyboard input.

2. The line numbers which *BUILD outputs as prompts do not get sent to the file. These line numbers should not be confused in any way with those produced by using the AUTO facility when typing in a BBC BASIC program. The AUTO line numbers form part of the program that is being input; the *BUILD line numbers are nothing more than a guide to how many lines have been input to the file so far.

It must be pointed out that *BUILD may be used to create any file of ASCII characters, not just an EXEC file. If, for a particular application, a data file of such a form is needed then *BUILD could be used to create that file. However, since the most common use of *BUILD is to produce EXEC files, this type of file will be used for examples in what follows. It should be assumed, though, that what is described could be equally applicable to any data file of ASCII characters.

8.2 LISTING AN EXEC FILE

*LIST (no abbreviation)
*TYPE (abbreviation *TY.)

Two other utility commands exist for listing the contents of a file which has been created using *BUILD. The command,

*LIST <filename>

causes the contents of the specified file to be output to the display screen, each line of output being preceded by a line number. For example,

```
>*LIST SKEY
1 *KEY0 MODE 7 |M
2 *KEY1 RUN |M
3 *KEY2 LIST |M
```

*LIST output with line numbers

The line numbers are generated as part of the *LIST output; they are not of course part of the file. A similar command,

*TYPE <filename>

also produces a listing of the file contents but, in this case, without line numbers:

```
>*TYPE SKEY
*KEY0 MODE 7 |M
*KEY1 RUN |M
*KEY2 LIST |M
```

*TYPE output without line numbers

Remember, however, that *LIST and *TYPE expect to be used with files containing nothing but ASCII characters. Using either of these commands with, say, a BASIC program file, or a data file created with PRINT# statements, will generate a listing of psychedelic unreadability.

8.3 USING AN EXEC FILE

*EXEC (abbreviation *E.)

Having created an EXEC file, calling it into action is simplicity itself:

*EXEC <filename>

will cause the file to be input, and its lines of data to be treated

exactly as though they were lines being typed in from the keyboard. Lines which look like BASIC, with a line number at the front, are added to any existing program that happens to be in memory. Other lines will be taken as commands, and an attempt made to obey them. Whichever is the case the lines read from the EXEC file are displayed on the screen. For instance, the dialogue in calling the function key EXEC file described above would be

```
>*EXEC SKEY <---------- Only this line is input
>*KEY0 MODE 7 |M
>*KEY1 RUN |M      } These lines output as file SKEY is read
>*KEY2 LIST |M
>
```

that is, just as though the three function key commands had been typed in.

8.4 EXAMPLES OF EXEC FILES

Basically, any set of commands that the user recognises as needing to be keyed in regularly can be made into an EXEC file. Such files will normally be quite short, comprising only a handful of lines. A file to set up all ten of the red function keys on the BBC Micro in some way would be a typical example. Another might be an EXEC file to archive a particular file (e.g. DATA, of size 3400 bytes) using the technique suggested in Chapter 3:

See page 30

```
>*BUILD ARCHIVE
   1 *DELETE 2.DATA
   2 *RENAME 1.DATA 2.DATA
   3 *RENAME $.DATA 1.DATA
   4 *LOAD 1.DATA 1900
   5 *SAVE $.DATA 1900 +3400
   6 *COMPACT <---------- Include *COMPACT to keep things tidy
   7 *CAT
   8   <--------(Press ESCAPE)
>
```

A single call,

```
*EXEC ARCHIVE
```

would be all that was needed to have the whole archiving process carried out.

Another example especially suited to an educational environment is that in which a BBC Micro is being used by a number of people. Here a 'good manners' EXEC file tries to ensure that the machine is in a fit state for the next person to use.

```
>*BUILD BYEBYE
  1 VDU 3 |M          (Printer off)
  2 VDU 20 |M         (Restore logical colours)
  3 VDU 26 |M         (Restore default windows)
  4 VDU 15 |M         (Page mode off)
  5 MODE 7 |M         (Start-up Mode)
  6    <--------(Press ESCAPE)

or, more briefly:

>*BUILD BYEBYE
  1 VDU 3,20,26,15 |M <---------- Individual VDU values can be combined
  2 MODE 7 |M
  3    <--------(Press ESCAPE)
```

8.5 THE AUTO-START FACILITY

The normal manner by which an EXEC file is called into action is by use of the *EXEC command. However, there is one other method which is used in the special circumstance of the BBC Micro being reset by the depression of the BREAK key. It can be arranged that, if BREAK and SHIFT are pressed simultaneously, the operating system, in addition to the usual actions associated with pressing BREAK, also attempts to load and obey a special EXEC file. This 'auto-start' file has to be called

:0.$.!BOOT

that is, file name !BOOT in the default directory $, on drive 0. Since even the humblest disk configuration will own a drive 0, the auto-start feature is thus available to all. (The name BOOT is an ancient item of computing terminology. It derives from 'bootstrapping', in that a BOOT program/file enables a system to pull itself up by its own bootstraps!)

To use the auto start feature, two steps have to be taken. Firstly, the !BOOT file has to be created - *BUILD is used exactly as we have seen so far:

```
>*BUILD :0.$.!BOOT <---------- Note: the '!' is an essential part of the filename
  1 *KEY0 MODE 7 |M
  2 *KEY1 RUN |M
  3 *KEY2 LIST |M
  4    <--------(Press ESCAPE)
```

Secondly, the Disk Filing System has to be informed that the start-up file is to be used.

*OPT4 (abbreviation *0.4)

This is done by using the command,

*OPT4,<value>

where <value> can be 0, 1, 2, or 3. Using

*OPT4,3

informs the DFS that the file !BOOT on the currently loaded disk is an EXEC file. This information is stored on the disk and reflected by means of a reference in the catalogue listing:

```
>*CAT                                       Disk in 'EXEC' mode

xxxxxx (xx)            Option 3 (EXEC)
Drive 0
Directory :x.x          Library :*.x
```

Now by holding down the SHIFT key and pressing BREAK, the system can be made to (automatically) reset, and then look for file !BOOT and act on it as an EXEC file. The screen output for our example would be:

```
  (SHIFT + BREAK pressed)
  ooooooooooooooooooooooooooooooooooScreen Cleared
    BBC Computer     }
                     }  Output as though 'BREAK'
    Acorn DFS        }      had been pressed
                     }
    BASIC            }

    >*KEY0 MODE 7    }
    >*KEY1 RUN       }  but SHIFT + BREAK reads
    >*KEY2 LIST      }    ! Boot file as well
    >
```

It is regularly the case that an auto-start EXEC file will comprise just one line! A quick and easy way of entering a program is by having a !BOOT file with a single line of the form,

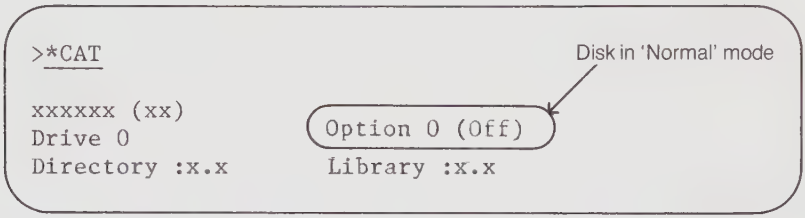
CHAIN "<program name>"

In this way pressing SHIFT+BREAK causes the !BOOT file to be executed, which in turn loads and runs the required program. This is a very attractive way of enabling novice users, such as young children or their parents, to use the BBC Micro without their needing to know anything about what the program is called, and to run it.

EXEC mode can be turned off by using

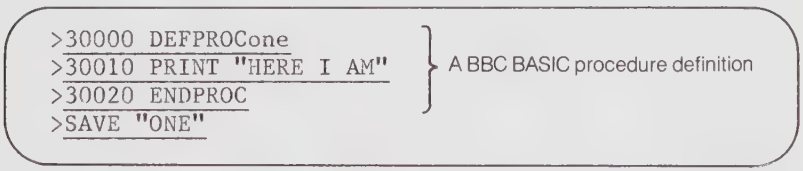
```
*OPT4,0
```

in which case the catalogue listing would show:



8.6 COMBINING PROGRAM FILES

Since an EXEC file is treated just as though its contents were being typed in from the keyboard, it seems reasonable to suggest that such a file could contain lines of a BASIC program. Then, by invoking that file with an *EXEC command, the lines of BASIC would effectively be added to whatever program currently happened to be in store. This little scenario is not quite right, however. To illustrate the problem assume that a file has been created with a few lines of code:



Now imagine, difficult as it might be, that this is a procedure that you find you are using regularly in your programs. There may be a number of other commonly-used procedures that can be identified, and these are all put either individually, or in groups, into a series of files to form a 'procedure library'. By using *EXEC the contents of one or more of these files can be quickly appended to the stored program. However, look at the contents of the file ONE:

```
>*DUMP ONE

0000  0D 75 30 0A 20 DD F2 6F  .u0. ..o
0008  6E 65 0D 75 3A 12 20 F1  ne.u:. .
0010  20 22 48 45 52 45 20 49   "HERE I
0018  20 41 4D 22 0D 75 44 06   AM".uD.
0020  20 E1 0D FF ** ** ** **  ........
```

To discuss exactly what the file contains, and why, is outside the scope of this book. (As it happens, the file contains an image of the 'internal representation' of the BASIC code - that is, the form in which it is held within the BBC Micro.) This is not a problem since our concern is merely to point out that the file contents are NOT as required for use with the *EXEC command; they are not ASCII characters.

*SPOOL (abbreviation *SP.)

In order to put a section of BASIC onto disk as a stream of ASCII characters, one of two steps have to be taken. Either the program lines can be input directly to disk using the *BUILD command in the way that has been described, or else another command, *SPOOL, can be employed.

*SPOOL (<filename>)

is a ubiquitous command which can be used to make a file copy of character streams that are sent to the screen. The command has to be used twice, firstly to open the 'spool file' <filename>, and secondly to close the file. In between these two *SPOOL calls, any data output to the screen by a BASIC command (NOT a DFS command) will also be sent to disk. Using LIST therefore would create an ASCII file of a BASIC program, since the listing would both appear on the screen and be copied to the file. For example, the following sequence would create an ASCII file of our little BASIC procedure ONE:

```
>30000 DEFPROCone
>30010 PRINT "HERE I AM"
>30020 ENDPROC
>*SPOOL ONE            (To open the file "ONE")
>LIST                  (Lists the program to the
30000 DEFPROCone           screen and, hence, also
30010 PRINT "HERE I AM"    to the spool file.)
30020 ENDPROC
>*SPOOL                          (Closes the spool file)
```

This file could now be combined with a program in store as follows:

```
>   10 PROCone          (Input lines of program
>   20 END                     from the keyboard)
>*EXEC ONE              (Call spooled file)
```

Retrieving file ONE from the disk would result in a screen display like this:

```
>>LIST
Syntax error                    } Can be ignored

>30000 DEFPROCone
>30010 PRINT "HERE I AM"        } Relevant contents of *SPOOL file
>30020 ENDPROC
>>*SPOOL
Syntax error                    } Can be ignored
```

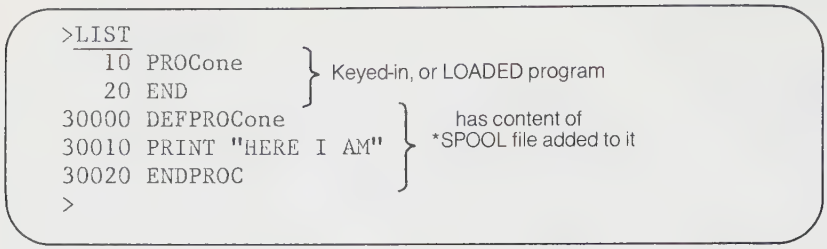
The two 'Syntax error' messages are nothing whatever to worry about. They are unavoidable. Look at the contents of file "ONE" as created by using *SPOOL and LIST:

```
>*DUMP ONE
0000  3E 4C 49 53 54 0A 0D 33  >LIST..3
0008  30 30 30 30 20 44 45 46  0000 DEF
0010  50 52 4F 43 6F 6E 65 0A  PROCone.
0018  0D 33 30 30 31 30 20 50  .30010 P
0020  52 49 4E 54 20 22 48 45  RINT "HE
0028  52 45 20 49 20 41 4D 22  RE I AM"
0030  0A 0D 33 30 30 32 30 20  ..30020
0038  45 4E 44 50 52 4F 43 0A  ENDPROC.
0040  0D 3E 2A 53 50 4F 4F 4C  .>*SPOOL
0048  0A 0D ** ** ** ** ** **  ........
```

Note: >LIST and >SPOOL are in the file

*SPOOL does exactly what has been claimed for it - having opened a spool file, characters that are sent to the screen are also sent to the file until it is closed. Thus the lines >LIST and the closing >*SPOOL also end up in the file! This in turn means that they are passed across to be obeyed when the file is input with *EXEC. Since commands >LIST and >*SPOOL are not recognised, the error message for each is a quite reasonable reaction on the part of the operating system.

The end result of this process is that the program lines from the file will have been added to the lines originally input from the keyboard:



There is no reason why this technique could not be repeated with a number of files. Note, though, that lines in the spool file will replace any lines in the stored program which have the same line number. This implies that either each file should contain sections of program with line number ranges which are distinct, or else that the stored program should be renumbered (with RENUMBER) in between using the different EXEC statements.

The alert reader will have spotted that here we have the germ of a technique for building up a library of EXEC files containing well-loved and, more importantly, tried and tested sections of BASIC programs. With BBC BASIC in particular there exists the attractive idea of constructing a library of procedures. What is really needed is a good way of automating the process of building a complete program from a number of these files. Surprising as it may be, Case Study 4 would seem to be just the sort of program that is wanted!

8.7 UTILITY PROGRAMS

At the beginning of this book some of the programs on the Acorn BBC utilities disk were used to illustrate some early ideas. One of these, the formatting program FORM40, was covered in detail. The way in which it was called differed from the usual LOAD/RUN of a BASIC program. Instead the program was initiated in much the same way as a DFS command, by typing its name preceded by an asterisk:

```
*FORM40
```

FORM40 is an example of what is termed a 'utility program'. Such programs will usually be written and held on the disk in machine code form. Here a brief look is taken at the features of the Disk Filing System which are concerned with loading and executing utility programs.

*RUN (abbreviation *R.)

In exactly the same way that the command CHAIN will load and enter a BASIC program, the command *RUN loads and enters a machine code utility program. Its general form is

```
*RUN <filename>
```

In fact this particular command is not very useful. The reason for this is that, for instance,

```
                    *RUN FORM40
          and       *FORM40
```

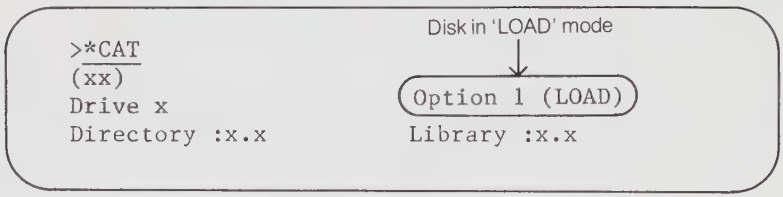
amount to exactly the same thing.

*OPT4 (re-visited)

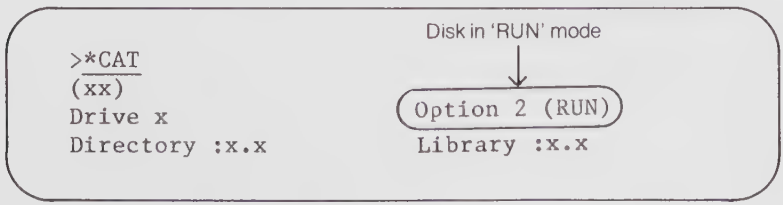
If the reader has not been on holiday since *OPT4 was introduced earlier in this chapter, it will be remembered that it could be arranged for an EXEC file, !BOOT, to be automatically entered whenever SHIFT+BREAK was used. The *OPT4 command was used with option value 3: *OPT4,3.

In this particular case !BOOT was required to contain an EXEC file. But *OPT4 can also be used in cases where !BOOT contains a utility program. There are two variations:

(i) *OPT4,1 - when SHIFT+BREAK is used, file !BOOT is read from disk into RAM by the equivalent of *LOAD. In this instance the contents of !BOOT are placed in the store at the load address given in the file's catalogue entry. The selection of this option is reflected in the catalogue listing:



(ii) *OPT4,2 - when SHIFT+BREAK is used, file !BOOT is loaded from disk and entered, i.e. the equivalent of *RUN is performed. Again the position of !BOOT in store is taken as load address from the file catalogue. A simple example of how this feature might be used would be that of making the FORM40 into the !BOOT file. Then, whenever SHIFT+BREAK was hit, the formatting program would be loaded and entered automatically. Again this setting from *OPT would appear in the disk catalogue heading:

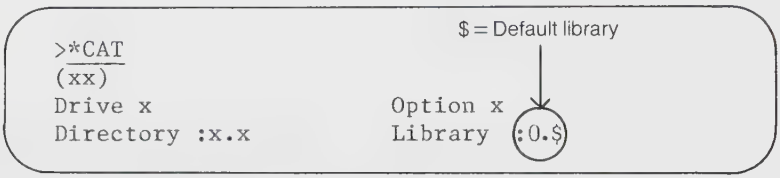


*LIB (no abbreviation)

When <filename> is mentioned with a *RUN command, it can include both drive and directory references. For example,

*RUN :0.$.FORM40

If the drive/directory parts are omitted the DFS has to assume default settings for these elements. Just as with other files, when the BBC Micro is switched on the default settings are for drive 0 and directory $. For utility programs these two parts together specify what is known as the 'utilities library' - that is, the drive/directory used by default whenever a utility program is called. Its current setting is given in the heading obtained with a *CAT listing:



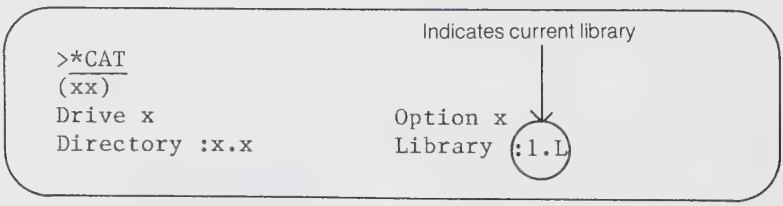
The library drive and/or the library directory can be altered by using the command,

*LIB (<drive>.) <directory>

where <drive> and <directory> have their usual meanings. Thus it is possible to hold all utility programs in one directory and have that directory referenced by default. Moreover, owners of dual-disk systems could arrange for all utility programs to be held on a disk which is loaded into a drive other than drive 0. Then any call to a utility program would cause the DFS to be rerouted from the drive 0 disk, which it examines first, to the library drive specified. For example,

*LIB :1.L

sets the utilities library as directory L on drive 1. Once more this change is reflected in the catalogue list heading:



9 RANDOM ACCESS FILES – Part 2

"Variety's the source of joy below.
From whence still fresh revolving pleasures flow."
John Gay, On a Miscellany of Poems

9.1 VARIABLE-LENGTH RECORDS

The first part of our treatment of random access files in Chapter 6 considered files which contained 'fixed-length' records; that is, files of records whose length, in terms of the number of disk bytes that they occupy, never changes. Examples there, for instance, looked at a telephone directory program which used fixed-length records, its data comprising a name of 15 characters maximum and a telephone number of up to 11 characters. The advantage of this approach when developing random access files is that it is very easy to calculate a value for the file pointer PTR# if every data item is the same size. In general terms a formula of the form,

```
PTR#fu% = record_size*(record_number - 1)
```

can be used to calculate the position of any record in the file.

The big disadvantage of a fixed-length record, if universally applied, is that it will sometimes impose a constraint which is undesirable. For instance, limiting a telephone number to a maximum of 12 digits is fair enough, since this maximum (at least for national calls) cannot be exceeded. But people tend to have names which vary wildly in the number of characters that they require, making it very difficult to impose a satisfactory fixed length for such a data item. Allocating a large enough space for every name, so that the likes of 'Major-General F.G. Barrington-Smythe' can be accommodated, means that large amounts of disk space are wasted when 'Mr. A. Jones' is the name being stored. Conversely, reducing this potential waste by defining a field length which is relatively small means that long names will be truncated, leading to a rather unpleasant output display.

Variable-length records, as the name suggests, are records which occupy as much or as little disk space as they require. Within a file containing such records there will exist a number of data items, each of which may be of a different length. Thus a file containing records comprising a name and an address would, by using variable-length records, be able to hold each name and address in full without incurring an unacceptable waste of disk space.

There will be some waste however, if 'waste' is defined as disk space which is not actually holding the user's data. For with a collection of variable-length records, our simple technique of calculating the start position for any record by use of a little formula is no longer possible. That formula was based on the fact that all of the records were of the same size. Now we shall see that, in order for random access to take place, additional elements have to be held on disk along with the actual data items. In this chapter we look at three different ways of implementing variable-length records:

1. **Identification bytes** - separating records by a single item of data which is used to signify the start of a new record.

2. **Record length** - including as part of the record a data value which indicates how many bytes that record actually occupies.

3. **Linked records** - methods of incorporating within a record a value which acts as a 'pointer' to another record.

It will be seen that each of the methods can be used in isolation, or in combination with one or more other methods if the application warrants it.

Case Study 5, which relates to the content of this chapter, covers the implementation of a database for general use. The study aims to draw together many of the points made about programming random access files both in this chapter and in Chapter 6.

9.2 IDENTIFICATION BYTES

The principle involved here is that of separating the variable-length records in a file by an additional byte which is used to 'identify' the start of a new record. (For the author this organisation always conjures up a mental picture of the 'school crocodile' - large numbers of scruffy schoolchildren in variable-length groups, each group being separated by an 'identification' teacher.)

Diagrammatically, the organisation is shown in Fig. 9-1.

Fairly obviously, the code which is used as the identification byte should be selected as one which cannot appear anywhere within a record. Using the byte &FE as an example, the following would create a file of 20 names and addresses with each name/address record separated by an identification byte:

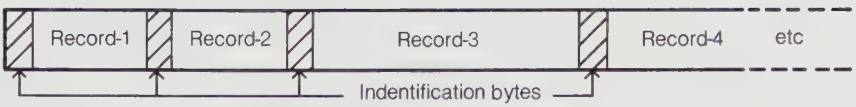


Fig. 9-1 Use of identification bytes

```
10 fo%=OPENOUT("DATA")
20 FOR i%=1 TO 20
30   BPUT#fo%,&FE <------------------ Write identification byte
40   INPUT"Name",name$: PRINT#fo%,name$   } then record fields
50   INPUT"Address",add$: PRINT#fo%,add$  }
60 NEXT: CLOSE#fo%
70 END
```

(It is suggested that the reader, after trying these little programs, use *DUMP to have a look at the contents of the files that have been created.) A simple program which would access this file, looking for a particular name, could be

```
10 fi%=OPENIN("DATA")                          Find identification
20 INPUT"Looking for which name",name$         byte as start
30 REPEAT                                      of record
40   REPEAT: id_byte%=BGET#fi% <-----------------
50   UNTIL id_byte%=&FE OR EOF#fi% <-------------
60   IF EOF#fi% PRINT"Name not found": CLOSE#fi%: END
70   INPUT#fi%,name_in_file$
80   IF name_in_file$=name$ THEN INPUT#fi%,address$:
       PRINT"Address is: ",address$: CLOSE#fi%: END
90 UNTIL 1=2
```

What happens is that the program, in looking for the required name, searches for the identification byte which marks the start of a new record (lines 40 and 50). The name part is input (line 70). If it is the required name (line 80) then the address part is input and displayed. However, if the name does not match then the address part is not input, the program passing straight on to a search for the identification byte which starts the next record.

It is worth noting here that the identification byte can play a dual role. Not only does it mark the start of a record but can also indicate the type of record that follows. This description should ring some sort of bell - it is precisely the method employed by the DFS to store different types of data item on disk, where 00, 40 and FF are the

identification bytes for string, integer and real data values respectively.

Now the astute reader will have spotted the limitation inherent in this technique for implementing variable-length records. It is that, since it is necessary to read every byte of the file (in order to detect an identification byte), the file must be accessed serially rather than randomly. To allow the random access of variable-length records, this method has to be extended somewhat.

9.3 RECORD LENGTHS

Any random access technique will involve the manipulation of the file pointer PTR#. Using our previous example, for instance, to skip over the address part of a record and directly access the name part of the next record would need PTR# to be incremented by some value. The simplest approach, given that PTR# is pointing to the start of one record, is to increase PTR# by a value equal to the length of that record. PTR# is thereby moved so that it points to the start of the next record. Now with fixed-length records the record length, by definition, is known. For variable-length records this is obviously not the case. If the record length is to be used to update PTR# in this way then that value must form part of the record itself. The organisation for our name/address example would change as shown in Fig. 9-2.

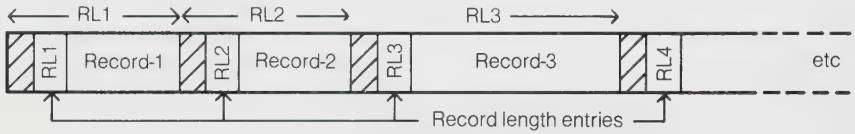


Fig. 9-2 Use of Record length entries

Note that the stored record length must account for all bytes in the record - including the record length entry itself. (Although it differs in this respect, the way in which the DFS holds strings on disk - identification byte 00, followed by a 'record length' and then the characters of the string - is essentially based on this same principle.) The short program to create a file of 20 names and addresses would now become as follows:

```
10 fo%=OPENOUT("DATA")
20 FOR i%=1 TO 20
30   BPUT#fo%,&FE: rec_len%=2: REM including itself!
40   INPUT"Name",name$: rec_len%=rec_len%+LENname$+2
50   INPUT"Address",add$: rec_len%=rec_len%+LENadd$+2
60   BPUT#fo%,rec_len%: PRINT#fo%,name$,add$
70 NEXT: CLOSE#fo%
80 END
```

Write record length then record fields

This example stores the record length, rec_len%, as a single byte. This means that the maximum length of the record is to be 255 characters (as with a BASIC string); if this is a limitation then rec_len% must be stored as an integer value, using a full 5 bytes. The revised program to search for a given address now becomes

```
10 fi%=OPENIN("DATA")
20 INPUT"Looking for which name",name$
30 REPEAT
40   ptr%=PTR#fi%
50   id_byte%=BGET#fi%: rec_len%=BGET#fi%
60   INPUT#fi%,name_in_file$
70   IF name_in_file$=name$ THEN INPUT#fi%,address$:
       PRINT'"Address is: ",address$: CLOSE#fi%: END
80   PTR#fi%=ptr%+rec_len%
90 UNTIL EOF#fi%: PRINT"Name not found": CLOSE#fi%: END
```

Use record length to calculate start of next record

Now if the input name is not that required, the file pointer PTR# is increased by the record length (line 80) so as to access the next record directly. Note that although in this example the identification byte serves no useful purpose, it would be unwise to discard it altogether. As pointed out above, this byte customarily fulfils two roles and, since different types of records usually have to be identified, its presence will still be of value (see below on 'Record Deletion').

9.4 LINKED RECORDS

As we have just shown, storing the record length along with the data parts of a record enables the value of PTR# to be speedily updated so as to point to the start of the next record in sequence. With a linked record organisation this idea is taken one stage further so that the PTR# value for its 'successor' - the next record in sequence - is incorporated into a record. This 'pointer' entry could well replace the 'record length' element to give an arrangement which would look like that of Fig. 9-3.

The principle involved here is that, when the record is input, the pointer field, since it actually contains the address of the start of the next record, can be used to reset PTR# directly. A special case does exist as far as the last record is concerned. Its pointer value cannot refer only to the disk equivalent of a black hole, but must reflect the fact that this is the final record in the chain. A standard ploy here is to set the pointer to an otherwise impossible value - any negative number will do, for instance, since all pointers to other records must be greater than or equal to zero.

Turning to our name/address example once again, the creation of a file in which the records were linked together would proceed thus:

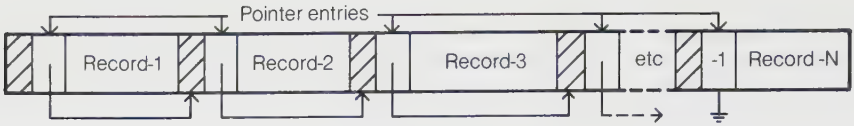


Fig. 9-3 Linked records

```
10 fo%=OPENOUT("DATA")
20 FOR i%=1 TO 20
30   BPUT#fo%,&FE
40   INPUT"Name",name$: INPUT"Address",add$
50   IF i%=20 THEN succ%=-1
        ELSE succ%=PTR#fo%+5+(LENname$+2)+(LENadd$+2)
60   PRINT#fo%,succ%,name$,add$ <-------- Write 'pointer' entry
70 NEXT: CLOSE#fo%                        then record fields
80 END
```

The pointer value calculated in line 50 is equal to the sum of the bytes occupied by the two strings, name$ and add$, plus the 5 bytes occupied by the (integer) pointer itself, plus the current value of the file pointer PTR#. For the 20th record, the last in the chain, the pointer value is set to -1.

Reading items from a file that is organised in this way is similar to our previous example which used record length entries as a way of recalculating the file pointer, PTR#. Here, in order to skip over all or part of the record, PTR# is set to the stored 'successor' pointer:

```
10 fi%=OPENIN("DATA"): succ%=0
20 INPUT"Looking for which name",name$
30 REPEAT
40   PTR#fi%=succ% <-------------------- Use 'pointer' to move to start
50   id_byte%=BGET#fi%: INPUT#fi%,succ%  of next record
60   INPUT#fi%,name_in_file$
70   IF name_in_file$=name$ THEN INPUT#fi%,address$:
       PRINT"Address is: ",address$: CLOSE#fi%: END
80 UNTIL succ%=-1: PRINT"Name not found": CLOSE#fi%: END
```

The repeated input of successive records ends when the pointer, succ%, is found to contain the value -1. This indicates that the end of the chain has been reached.

9.5 WHY USE LINKED RECORDS?

The idea of one record containing a pointer to another is the starting point for the vast majority of commercial disk file organisations. Case Study 5 aims, by implementing a simple database, to give the reader a feel for the programming techniques associated with data structures which comprise a collection of linked records. Before the reader scans eagerly through to Section II of the book, however, it is worth conducting a short enquiry into the major reasons for using a linked record organisation.

9.5.1 Ordered Records

Many applications require data to be presented in some sort of order. For instance, an obvious extension of our name/address example would be to add the facility of producing a listing of all names and addresses stored on disk, this listing being output in alphabetic order of name. Now if this 'sorted' output is to be achieved, either the data has to be in the correct order on disk or some sorting operation has to be carried out on the whole of the file's data before printing can take place: the former is obviously most desirable. The difficulty is that the data - names/addresses in this instance - will not necessarily arrive in order. The file may initially be set up in this way but subsequent entries added to the end of the file, unless one is exceptionally careful in choosing new friends, will destroy this alphabetic ordering.

A linked organisation, however, enables a required order to be maintained by rearranging the pointer parts of the appropriate records to accommodate any newcomer. For instance, assume that Fig. 9-4 is a section of the name/ address file:

Fig. 9-4 Section of name/address file

It is required to add a new record,

to this file, with alphabetic ordering being maintained.

This would be achieved as follows. The new record would be physically placed at the end of the file. By searching through the existing records, though, the correct position of this new record in the chain of records could be determined. The alteration of the relevant

pointers (Fig. 9-5 below) is all that is necessary to maintain the required ordering.

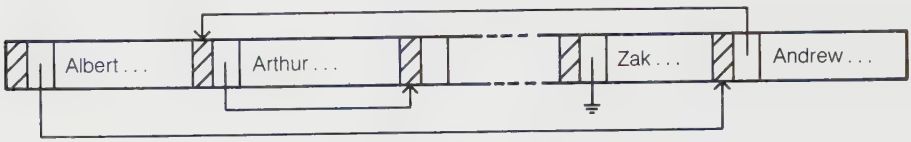


Fig. 9-5 "ANDREW" added with ordering maintained

The important point to realise here is that, with a linked organisation, records no longer have to be physically placed in the correct sequence on disk. The ordering is indicated solely by the way in which the pointers link the records to each other.

9.5.2 Deleting Records

The time will come when a record has to be deleted from a data file. With the name/address file, for instance, Uncle Clive, having passed on to that computer room in the sky where syntax errors never happen, should really have his entry removed.

With a linked organisation the removal of an unwanted record is just like removing links from a chain. The pointer to the unwanted record is adjusted so that it now skips this record and points instead to the next record in sequence. This is illustrated by Fig. 9-6.

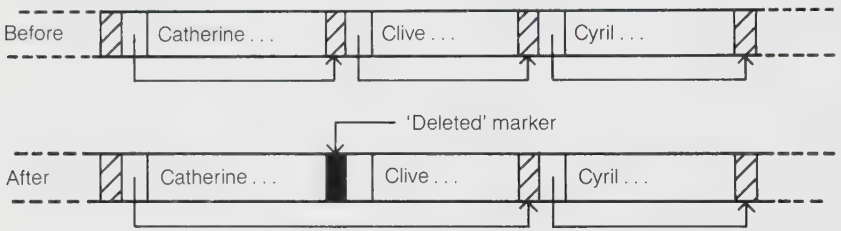


Fig. 9-6 Deletion of record "CLIVE"

When deleting a record it is sensible to indicate in some way within the record itself that it has been deleted. This is an occasion when the identification byte can play a part, a special setting for this byte being used to indicate 'deleted record'. (Note that this use of an identification byte is about the only sensible way of deleting a record in a file with a non-linked organisation, whether it be a file of fixed or variable-length records. The alternative, of physically moving data items so as to overwrite the unwanted record, is usually not worth thinking about.)

9.5.3 Multiple Pointers

If it serves a useful purpose, records can be designed so as to include more than one pointer item. A common enough organisation is that in which records contain two pointers. The first, indicating the next record in the chain (successor), is as we have seen already. The second pointer indicates the previous record in the chain (predecessor). Such a data structure is illustrated in Fig. 9-7.

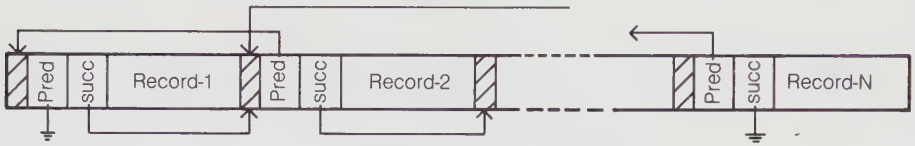


Fig. 9-7 Multiple pointers

Note here that the first of the predecessor pointers is set to indicate 'end of chain', since the first record can have no predecessor.

The use of two pointers like this enables the chain to be searched from either end, which might save time. Also, having these two pointers as part of every record, facilitates quick deletions since, rather than having to work through the whole chain to find the positions of the predecessor and successor records (which have to be linked together to effect the deletion) they can be located immediately.

9.5.4 Indexed Data Structures

Pointers enable data 'structures' to be built, which reflect more clearly the way in which the different data items relate to each other. This in turn leads to more efficient ways of manipulating the data, which is very important once a program starts to handle large data files. Although a treatment of the many different data structures regularly used in commercial data processing applications is beyond the scope of this book, it is possible to give the reader a feel for what is involved by considering an aspect that tends to be common to many - the use of an 'index'.

Turning once more to the name/address data file - as this becomes larger it will take longer and longer for the program to work through the chain of records to locate the one that is needed. Now this process could be shortened if the records were subdivided in some way, with any search limited to just one of the groups of records. An obvious grouping in this instance would be one based on the first letter of the name part of the record - all those starting with 'A' could form one chain, all those starting with 'B' another chain, and so on. Then, when any search is needed, only those records in the appropriate alphabetic group are examined and in this way the search time can be reduced dramatically.

To implement such a data structure requires that, in addition to the data records themselves, an index also be formed as part of the file. The sort of organisation that results is shown in Fig. 9-8.

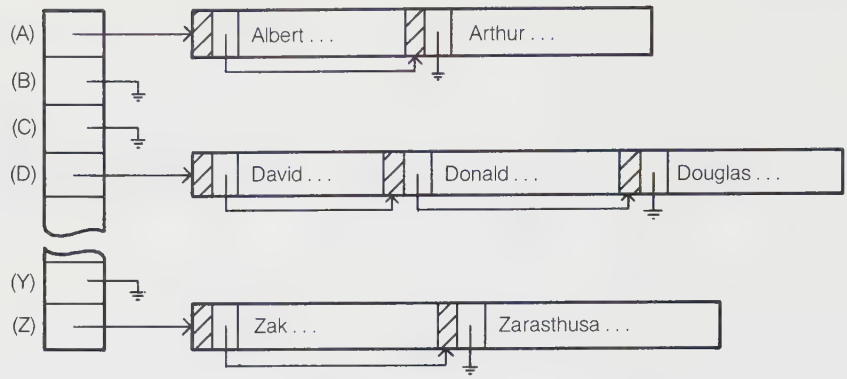


Fig. 9-8 Organisation of an indexed file

There the index part of the file comprises one entry for each separate chain into which the file is subdivided, each entry holding the file address of the first record of its particular chain. Now whenever a search is made, this index becomes the first port of call with the appropriate entry being used to initialise PTR#. An added refinement is for the index entries to comprise two items - the start address of both the first and the last records in their respective chains. This makes insertion of new records possible without the need to search through any particular chain to find its end. If no records have been added which in our example have name elements starting with a particular letter of the alphabet, then that item of the index will contain the value -1 to mark 'end of chain'.

By way of an example, the following program creates a name and address file with a 26-item index of start/end addresses at the front of it, as described:

```
10 DIM start%(26), end%(26)
20 fu%=OPENOUT("DATA")
30 FOR i%=1 TO 26: start%(i%)=-1: end%(i%)=-1: NEXT
40 nxfree%=265
50 FOR i%=1 TO 20
60   INPUT"Name",name$: INPUT"Address",add$
70   ch%=ASCname$-&40
80   IF end%(ch%)=-1 THEN start%(ch%)=nxfree%
         ELSE PTR#fu%=end%(ch%)+1: PRINT#fu%,nxfree%
90   end%(ch%)=nxfree%
100  PTR#fu%=nxfree%: BPUT#fu%,&FE: PRINT#fu%,-1
110  PRINT#fu%,name$,add$
120  nxfree%=PTR#fu%
130 NEXT
140 PTR#fu%=0: PRINT#fu%,nxfree%
150 FOR i%=1 TO 26: PRINT#fu%,start%(i%),end%(i%): NEXT
160 CLOSE#fu%: END
```

Here the index is initially set up with its start/end entries all at -1(line 30). A pointer, nxfree%, has to be set to point at the first available byte in the file for storing data records(line 40), i.e. the first byte beyond the index area. Now whenever a name is input, the appropriate end pointer is examined(80) and either a start made on a new chain or a successor pointer (value nxfree%) put into the record currently at the end of the chain (line 80). Both start/end elements of this entry are updated if necessary, and the new record written to position nxfree% in the file. Its successor is set at -1, since this new record is at the end of the chain. When all records have been added to the file, the current setting of nxfree% and the whole of the index are written to the first section of the file (lines 140-160). The search program for a file of this type would now become

```
 10 DIM start%(26),end%(26)
 20 fi%=OPENIN("DATA"): PTR#fi%=PTR#fi%+5
 30 FOR i%=1 TO 26: INPUT#fi%,start%(i%),end%(i%): NEXT
 40 INPUT"Looking for which name",name$
 50 succ%=start%(ASCname$-&40): IF succ%=-1 GOTO 120
 60 REPEAT
 70   PTR#fi%=succ%
 80   id_byte%=BGET#fi%: INPUT#fi%,succ%
 90   INPUT#fi%,name_in_file$
100   IF name_in_file$=name$ THEN INPUT#fi%,address$:
        PRINT'"Address is: ",address$: CLOSE#fi%: END
110 UNTIL succ%=-1
120 PRINT"Name not found": CLOSE#fi%: END
```

Line 30 reads the index entries into the arrays. Line 50 accesses the start element of this index, calculated by using the ASCII value of the first letter of the required name to produce 1 for 'A', 2 for 'B' and so on. If the index entry is -1 then the name is not in the file.

9.6 GARBAGE COLLECTION

We have mentioned in Section 9.5.2 above the occasional need to delete records from a file. Setting the identification byte to a particular value to denote 'deleted' is a simple way of performing this operation, with all unwanted records being ignored in any file processing that takes place. The big disadvantage with this approach is that the records, since they are not actually removed from the file, still occupy disk space. This is not a problem if, over the lifetime of the file, deletions are going to be few and far between. With a file which is continually changing, however, a time must come when the file will contain a very high proportion of 'deleted' records - as with many

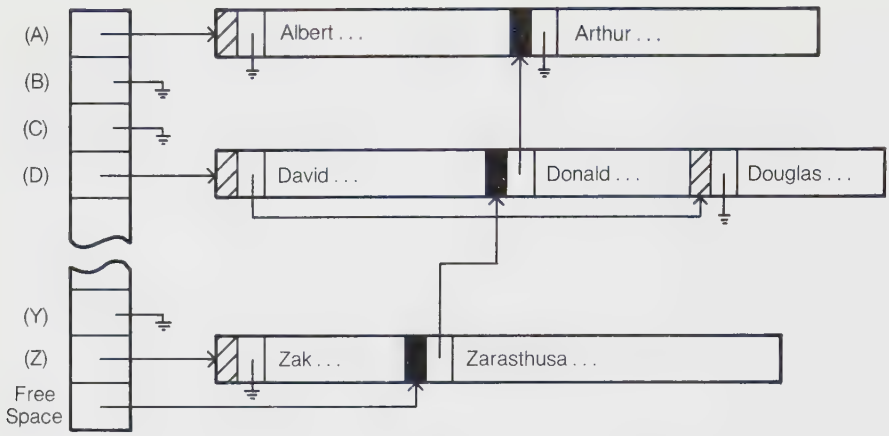


Fig. 9-9

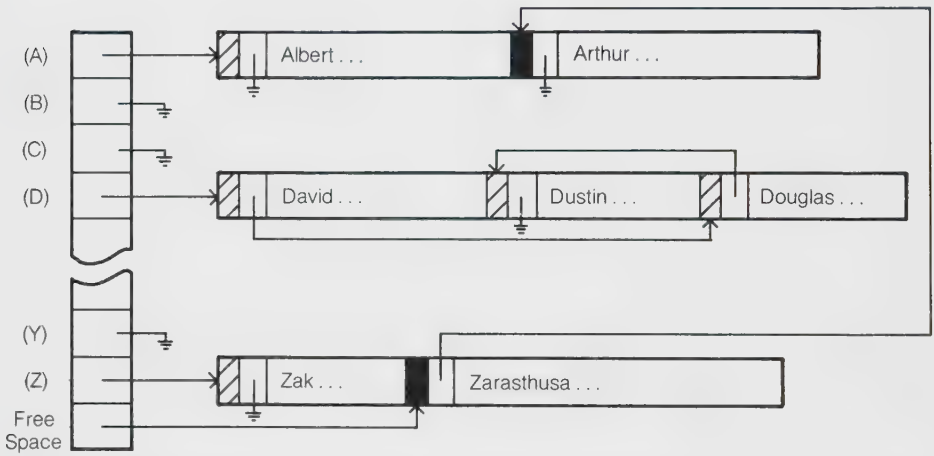


Fig. 9-10 Space previously occupied by "Donald . . ." now used by "Dustin . . ."

of the author's students, it will be more dead than alive. The logical conclusion of the process will be a ridiculous situation in which there will be no further room on the disk to add new records because so much space is being taken up by records that have been 'deleted'.

This is unacceptable of course. For a file which will be subjected to many deletions then there must be some way of re-cycling the disk space that was occupied by a deleted record. The process by which this space is gathered together and then made available for re-use when further records are added to the file, is known as 'garbage collection'.

One way in which garbage collection can be carried out is by extending the use of an indexed structure. What happens in rather basic terms is that an additional entry is added to the index to cater for deleted records. Then, when a record is deleted, it is unlinked from its predecessor and successor exactly as previously described (Section 9.5.3). But now, in addition, this record is linked onto the end of a chain of 'deleted records' - or 'free space' as it is often termed. In this way a tally of all of the 'deleted' record space is established (Fig. 9-9).

A modification of the procedure for adding new records to the file completes the picture. Instead of automatically adding a new record on to the end of the file, a scan through the 'free space' chain is made. If an entry in this chain is of sufficient size to accommodate the new record then this 'free space' is used; if not, the record is added to the end of the file as usual. If a deleted item has been overwritten then, as a final step the re-possessed space has to be removed from the 'free space' chain. This is carried out as a 'deletion' of a record from the 'free space' chain(!) and is conducted by means of the standard deletion process. Fig. 9-10 illustrates the end result of this sequence of events.

The technique of garbage collection, as we have seen, requires no small programming effort. In fact the processes of constructing and using files of linked records can become quite complicated.

As far as this book is concerned, the introduction to linked record organisations is the culmination of our study of disk programming techniques. For the ambitious programmer, though, this will just be the beginning, since any number of different data file organisations can be based upon these principles. Happy programming, and may your garbage always be collected!

10 GETTING THE (ERROR) MESSAGE

"There is no mistake; there has been no mistake; and there shall be no mistake."

The Duke of Wellington

Wellington was not only renowned for his !BOOTs. He was clearly also a member of that class of programmer who, when faced with an error message, proclaims, 'I can't see anything wrong - it must be the computer!' He lived to the age of 83, though, and the fact must have dawned on him by then that 'it' so rarely is the computer's fault.

When presented with a disk error message then, resist the temptation to castigate the machine, and look through the list below. For each possible error message there is a more detailed explanation of what the DFS is complaining about, together with an indication of the most likely reasons for that message having been produced. Where appropriate, some hints as to how to recover the situation are also provided.

Against each message is given an error number. This is the value taken by the system variable ERR if the error is caused by a statement appearing in a BASIC program. Disk errors may thus be intercepted and handled by use of the ON ERROR facility in exactly the same way as other program execution errors.

BAD ADDRESS (252)

This is a command which expects a store address to be provided as one of its arguments fails to find it. *LOAD and *SAVE are the likeliest commands to be in use if this error occurs. The former can be given, optionally, a start address for a loading position; *SAVE expects both a start and a finish address.

BAD ATTRIBUTE (207)

This error can only occur in connection with the *ACCESS command. This command has an optional second argument which, if supplied, can only be the character 'L'. The message is generated if a second argument is found but does not happen to be 'L'.

BAD COMMAND (254)

This is an indication that the command that has been input is neither a recognised DFS command/utility, nor the name of a utility program on the currently loaded disk. Common mistakes which will give rise to this error message are mistyped commands (or abbreviations), or the omission of the * which precedes DFS commands/utilities. In the case of utility programs, the current library is searched; if it does not contain the utility program then 'Bad command' will be produced here also, even though the program name might be quite correct.

BAD DIRECTORY (206)

Applicable only to the *DIR command, this message is generated if the quoted directory is not a single character. Any single character will be accepted apart from ':' (which produces a 'Bad drive' message!). Note, however, that the characters '#', '*', and '.' would not be wise choices for directory identifiers since they each have special meanings in the context of a filename.

BAD DRIVE (205)

The nominated disk drive is either incomprehensible or illegal. Commands *BACKUP, *CAT, *COMPACT, *COPY and *DRIVE can all take a drive number as an argument. If the argument supplied is not an integer in the range 0-3 then the 'Bad drive' message will be generated. Commands other than the above will expect a filename as one of their arguments, and a drive number can of course form part of that filename. In this case the drive number is preceded by a ":" and again must be in the range 0-3. Note that the system will not object to any command which refers to a legal drive number - even if that drive is not actually present! In such a case the DFS will try to obey the command and this will quite probably result in the system 'hanging' as it waits forlornly for a response from the absent drive.

BAD FILENAME (204)

The quoted filename is illegal for some reason. The most likely cause is that the restriction of seven characters on the length of the filename has been exceeded. Other possibilities might involve incorrect use of the wildcard character, '*'. The filename *FRED, for instance, would also be rejected as a 'Bad filename'.

BAD OPTION (203)

This is produced if the *OPT command appears to be neither *OPT1 nor *OPT4, since only the values 1 and 4 are acceptable. One by-product of this restriction is that there must be a space or comma separating the *OPT command from any argument that is provided, e.g. *OPT4 3. If this does not happen, e.g. *OPT43 is input, then this error will again be produced (since 43 is a 'Bad option').

CAN'T EXTEND (191)

An attempt has been made to extend a random access file - that is, to write data beyond the existing end of the file - and there was insufficient disk space available for this operation to be successfully completed. The reasons for this, and some solutions to the problem, are given in Section 6.3. Remember that in this situation the file WILL have been extended as far as possible before the free disk space was exhausted. This might well mean that the last data item in the file is incomplete and will need to be amended if the file is to be successfully input (see Section 7.4 on 'patching' data files).

CATALOGUE FULL (190)

Each file held on a disk requires an entry to be made for it in the disk catalogue. For the Acorn DFS, this catalogue has room for 31 entries. Once this number of files have been created, therefore, any subsequent command which attempts to create another file will produce the 'Catalogue full' message. This is not to say that the DISK is full! A collection of 31 small files would fill the available space in the disk catalogue whilst leaving a substantial amount of unused disk space.

CHANNEL (222)

This message is related to the use of disk files from within a BBC BASIC program. It is pointing out, with admirable brevity, that an attempt has made to use a file which is not 'open'. The message, 'Channel', is a reference to the fact that the 'channel' variable - which is initialised for a particular file by the commands OPENOUT, OPENIN and OPENUP and referred to in all subsequent operations on that file - has an undefined value. Apart from the obvious cause (that the appropriate 'open' command has not been called), another possibility

is that CLOSE#0 has been used to close all files when only one particular file should have been closed.

This same message can also be generated in a circumstance which, although technically correct, can be misleading. A program can happily open a random access file for updating even though that file DOES NOT EXIST! No error indication is given until an attempt is made to use the file, when 'Channel' appears. The little sequence,

```
>10 fu%=OPENUP("AWOL"): REM Use OPENIN with BASIC I
>20 PRINT#fu%,1
>30 END
>RUN
```

would produce

```
Channel at line 20
```

if file "AWOL" did not actually exist.

DISK CHANGED (200)

It requires a fair measure of concentrated idiocy to generate this message. The sequence of events which lead up to its production are:- open a file; remove the disk and replace it by another one; try to read from/write to the file that was opened. If the error does occur, its root cause can almost certainly be traced to the failure of a program to close a data file that it was using.

DISK FAULT (199)

This is bad news. The full form of the message is

```
Disk Fault <number> at <track> <sector>
```

which is indicating that the DFS has been unable to read from or write to the disk at the given track/sector position.

If the disk is actually faulty or has been damaged in some way, then it might well be unreadable. By far the most likely reason, however, is that there is a problem with the disk's format. The disk drive can only successfully read from a disk when the disk format is correct, since it is only by using the format information that separate data blocks can be identified. If, for instance, a disk that has been formatted for 80-tracks is used with a 40-track disk drive, or vice-versa, then the format data will not be detected correctly and 'Disk fault' will be produced. A disk that has been formatted on another brand of computer, or found in the gutter, is likely to produce this error message for the same sort of reason.

It is when a fault like this develops with a hitherto usable

disk that those with a nervous disposition should avert their eyes. For one thing it means that the disk has to be re-formatted with a consequent loss of some data. And for another it casts doubt upon the state of health of the user's disk drive, since it must be emphasised that IT IS IMPOSSIBLE FOR THIS SITUATION TO BE PRODUCED BY AN INCORRECT BASIC PROGRAM. The fault has nothing to do with the correctness, or otherwise, of the data on the disk. A drive that is occasionally a fraction out in its positioning of the disk before a write operation, for instance, is bound to overwrite some of the format data. Unhappily, the fault will not be detected until an attempt is made to read/write the affected sector of the disk. The drive must of course be returned to the dealer or manufacturer for repair.

Two of the Case Study programs have been designed to help out here. Case Study 6 gives a very simple 'disk soak' program which can be used to test a disk drive which is suspected of having such a fault. Case Study 7 can be used to partially resuscitate a disk file which has an unreadable sector within it.

DISK FULL (198)

As opposed to 'Catalogue full', this means that the DFS has been unable to find sufficient room on the disk for a file. This could occur with SAVE, *SAVE or *COPY which ask for files of a particular size to be written to the disk. It might also occur with *BUILD, or by use of OPENOUT from within a BASIC program - both require 40 sectors to be available in order that they may create their files.

'Disk full' is a bit dramatic in that it indicates first and foremost that the DFS cannot find an unused area of the right size on the disk. It may well be that, by using *COMPACT, a large enough space can be generated.

DISK READ ONLY (201)

Quite simply, a disk write operation has been attempted with a disk that has its 'write-protect' notch covered. Note that such operations include not only the more obvious SAVE, *COPY, *DELETE etc., but also commands such as *ACCESS, *DIR and *OPT4 which also put information onto the disk by way of writing to the disk catalogue.

DRIVE FAULT (197)

The full form of the message is:

Drive Fault <number> at <track> <sector>

It is stating that communication has failed between the BBC Micro and disk drive <number>. Unless the problem is simple, such as the computer-to-disk cable being incorrectly connected, this message does rather suggest that either the cable or the disk drive is in need of repair.

EOF (223)

A program has tried to read, by means of INPUT# or BGET#, beyond the end of a data file. The following little program would always produce the error, for instance:

```
10 fi%=OPENIN("DATA")
20 REPEAT
30   X=BGET#fi%
40 UNTIL 1=2
```

FILE EXISTS (196)

This message only occurs with the *RENAME command. It indicates that the suggested new name for a file is the name of a file already in existence. The solution is simply to *DELETE the existing file and then perform the *RENAME.

FILE LOCKED (195)

An attempt has been made to write to or delete a file that has been locked by use of the *ACCESS command. The file must be unlocked by *ACCESS, after which the required operation can be carried out.

FILE NOT FOUND (214)

It is not too difficult to appreciate that this message is produced when the DFS fails to locate the filename which has been quoted as an argument to a DFS command. The cause is not always so obvious, however, since it is possible to commit a multitude of sins each of which result in the same message. Some of the most popular are as follows:

(a) Typing error: the file really doesn't exist.

(b) The disk drive which the DFS references does not hold the disk which has the file on it. Perhaps the wrong disk is loaded, or the <drive number> part of the filename is wrong, or else the currently set default drive has caused the DFS to access the wrong drive.

(c) Similarly, the DFS could be using a directory or library which is incorrect for the required file. Either the <directory> part of the filename is wrong, or the currently set default directory needs to be changed. Remember that the system defaults for drive/directory are 0 and $ respectively, and that these are set whenever BREAK is used.

(d) Use of the wildcard characters # and * has produced a filename

which does not match any of those on disk.

(e) The auto-start facility (SHIFT+BREAK) has been attempted, but file :0.$.!BOOT does not exist.

The list is exhausting, but not exhaustive.

FILE OPEN (194)

An attempt has been made to open a file that is already open. This can be caused from within a program through omission of the CLOSE# command between, say, an OPENOUT (to create a file) and an OPENIN/OPENUP (to update it). Alternatively, the program fails to CLOSE# a file that it has been working on, with 'File open' being generated when any subsequent command (e.g. *DUMP, *LIST, *DELETE etc.) tries to access the file. The way out of this situation is to type, using 'immediate mode',

```
>CLOSE#0
```

which closes any file which happens to be open.

Leaving opened files is a common failing of programs which do not utilise the ON ERROR facility for intercepting run-time errors. When the program fails, an immediate jump out of the program is made, and the CLOSE# statements - designed to be obeyed if the program had run successfully - are bypassed. The situation is avoided by always having a simple ON ERROR routine which does nothing more than close all files and produce the standard system error message (see any Case Study program).

FILE READ ONLY (193)

For those programming in BBC BASIC, this is a rather superior error message which really only applies to the proud owners of a BASIC II system (see Section 6.2.3). It indicates that a file which was opened for input transfers only, i.e. by using OPENIN, has had an output statement, e.g. PRINT#, directed at it.

The reason for the distinction between the two versions of BASIC is that BASIC II has an OPENUP statement for opening a file which is to be updated, enabling OPENIN to be used with files that are to be classed as 'read only'. BASIC I on the other hand does not have the OPENUP statement and OPENIN is used in a dual role both to open a file for input, and to open a random access file for updating - which makes this particular 'File read only' situation impossible to detect.

NOT ENABLED (189)

Two commands, *BACKUP and *DESTROY, have to be preceded by the command *ENABLE before they will work. This is a simple protection against an unthinking user eradicating large groups of disk files by accident. The message 'Not enabled' says that *ENABLE has not been used.

SYNTAX (220)

Any DFS command which is recognised, but which has its argument list incorrectly supplied, will cause a 'Syntax' error message to be generated. This message will present as a guide the correct syntactical form for the command being attempted. The output for any particular command is in fact identical to that which is produced by the *HELP feature. For example, the incorrect command,

*RENAME DATA

would result in the message

Syntax: *RENAME <old fsp> <new fsp>

to indicate that some parts of the full command had been omitted.

TOO MANY OPEN FILES (192)

It is only possible for five files of whatever type to be open at once. (This limitation is primarily due to the amount of space available for disk buffers.) This error message is produced when an attempt is made to open file number six. If it is essential for a program to operate with this number of files, then the only recourse is to re-program so that files are only opened just before they are used, and closed immediately afterwards. This will inevitably take its toll on the execution speed of the program.

11 ALTERNATIVE DISK FILING SYSTEMS

"You pays your money, and you takes your choice."
Anon (from a peepshow rhyme)

This maxim, accredited to Anon (a prolific author if ever there was one), applies just as well to Disk Filing Systems as to anything else. Following the production of Acorn Computer's own DFS for the BBC Micro, it was natural and pleasing that alternatives should soon appear on the scene. At the time of writing, the major competitors to the Acorn DFS are those produced by Pace and Watford Electronics. (The advent of the 3" Micro drive as an alternative to the standard 5.25" disk drive will also mean that a crop of new DFS ROMs will appear specifically for this type of drive.)

It was not practicable to write this book without basing it on the features of one particular DFS, however, and the Acorn version was the obvious choice. But since all of the alternatives to this DFS claim to be Acorn-compatible, what has been written should be equally applicable no matter which brand of DFS is beating within the reader's BBC Micro.

Each alternative DFS, though, has added its own personal facilities to those offered by the Acorn DFS. This short chapter attempts to complete the picture by looking briefly at the most important of these additional features. To enable the reader to place any particular feature in the correct context, each is accompanied by a cross-reference to its relevant position within the body of the text.

11.1 AMCOM DFS (Pace Software Supplies)

(1.3) The disk catalogue may contain, optionally, up to 63 files per disk. This mode of operation, known by the term 'Extended mode', may be selected in preference to 'Acorn mode' which is the default. This choice can be arranged to be made on auto-start, or two special DFS commands can be used:

*SYS0 - selects Acorn mode
*SYS1 - selects Extended mode

(1.5) Command *CAT produces a disk heading which indicates the mode in operation.

(2.1) Formatting a disk is conducted by a DFS command rather than by a utility program. The command is *FORMAT, which has to be preceded by *ENABLE. The decision as to whether the disk is to be formatted to 40 or 80 tracks can be influenced by the command,

*OPT3,<tracks> (<tracks> = 40 or 80)

before invoking *FORMAT. If the disk to be formatted is not empty, the question

Erase? (Y/N)

has to be answered before formatting continues. The resultant catalogue size depends on whether Acorn or Extended mode is in force. A disk must subsequently be used in the mode applicable at the time it was formatted.

(2.3.1) Filenames can be up to 15 characters in length when in Extended mode.

(2.5) In both modes, &PAGE is set at &1500

(3.4.3) Wildcard characters may also be used with LOAD/CHAIN commands. In such instances the catalogue is searched for the first file which matches the ambiguous file name.

(4.1.2) Files are locked by placing a '~' character before the filename, e.g.

SAVE "~FRED"

A locked file is unlocked by using *RENAME, e.g.

*RENAME "~FRED" "FRED"

The '~' is effectively part of the filename; it must be used, therefore, whenever the file is referenced, e.g.

```
LOAD "~FRED"
```

In Extended mode a locked file will not appear in a *CAT listing.

(4.2) The contents of a whole disk may be deleted by the combination,

```
*ENABLE
*CLEAR <drv>
```

(4.3) Command *INFO produces output for each file in the order,

Directory
Name
Drive
Length of file
Execution Address
Load Address
Start Sector

(4.4) The DFS buffer may be located anywhere in RAM by use of two commands:

```
      *OPT5,<address>  - sets the start address
and   *OPT7,<length>   - sets the buffer length
```

By a judicious location of the DFS buffer, therefore, a *COMPACT operation may be performed without the program in store being overwritten.

(10) An additional error message,

```
BAD SYSTEM (208)
```

indicates that the DFS is in the wrong mode for the currently loaded disk.

11.2 WATFORD DFS (Watford Electronics)

(1.3) The disk catalogue may contain, optionally, either 31 or 62 files per disk.

(2.1) Formatting a disk is conducted by a DFS command rather than by a utility program. The command, which has to be preceded by *ENABLE, is either

```
*FORM40 <drive>
```

or

```
*FORM80 <drive>
```

depending on whether the disk is to be formatted to 40 or 80 tracks. As part of the formatting process the user is requested to indicate whether a 31 or 62 file catalogue is required. A large catalogue occupies a further 2 sectors in addition to sectors 0 and 1 used by the standard Acorn DFS. It is therefore possible for disks created with an Acorn DFS to be used with the Watford DFS. The reverse is best avoided, since an Acorn DFS will not recognise the extra two catalogue sectors and could well overwrite them.

(2.2) Verification of a disk can be carried out by a DFS command,

```
*VERIFY <drive>
```

rather than by a utility program.

(3.1) A command,

```
*WORK <filename>
```

may be used to set up a default filename. This filename will be used automatically with any command for which a <filename> argument has been omitted, e.g.

```
*WORK FRED
LOAD ""
```

is equivalent to

```
LOAD "FRED"
```

Any currently set *WORK filename appears as part of the heading of a *CAT listing.

(3.2.2) The *RENAME command allows the use of ambiguous filenames.

(3.3) The *WORK filename can be extended to include both <drive> and <directory>. If the former is specified, it becomes the default drive, as though *DRIVE had been used.

(3.4.3) The wildcard character '*' is defined to represent 'any string of characters'. It can therefore be used more than once in a file name, e.g. '*X*' would match with EXAMPLE and AXES. Wildcard characters may also be used with LOAD/CHAIN commands. In such instances the catalogue is searched for the first file which matches the ambiguous file name.

A command,

*MOVE <src drv> <dest drv> <afsp>

acts in the same way as *COPY, except that it prompts the user to confirm that a file is to be copied.

(4.4) It is possible to discover the areas of disk left behind through file deletions by using

*HELP SPACE

This command also gives the total free space available - which would all be moved into one area if *COMPACT were to be carried out.

(5.2.1) Files that have been left open after a program run can be discovered by using

*HELP FILES

which displays name, channel and PTR# value for each file that is open.

(5.3) OPENOUT selects the largest area on disk in which to create a file.

(5.4) A command,

*MLOAD <filename>

will load the quoted file to address &1100, switch off the disk system, and then move the file contents to its correct load address.

(8.7) A corresponding command for machine code programs is

*MRUN <filename>

which does the same things as *MLOAD, and then causes the program to be entered at its execution address.

(10) An additional error message,

AMBIGUOUS FILENAME (208)

appears if an ambiguous file name is used illegally.

THE CASE STUDIES
Introduction

"Studies serve for delight, for ornament, and for ability."
Francis Bacon, Essays

This section of the book comprises seven Case Studies. It would appear that Bacon added a bit more meat to his works by using a similar approach; his attempts to convince readers of their value do strike one as being something of a hard sell though. The reasons for including the Case Studies in this book are a little more down-to-earth: firstly, to give some non-trivial examples of programming the BBC Micro with disks; and secondly, to provide the reader with some programs which could actually be put to work in a serious way. As one might expect, the text will concentrate on the Case Studies as programming examples, since this is in accord with the main thrust of the book. Wherever possible, however, suggestions are made for areas in which the Case Study programs might be employed, and ways in which they could be modified by the enthusiastic and/or insomniac BBC Micro user.

In presenting each of the Case Studies, the same descriptive format is used. It comprises seven elements:

(a) FUNCTION - What the program is designed to achieve, and in what circumstances. Its capabilities, and its limitations.

(b) SAMPLE DIALOGUE - Examples of the program in use, showing what is output by the program and the responses made by the user. In these dialogues the style used throughout the book, of underlining the user inputs, is followed. In addition the dialogues have a textual commentary inserted where it is thought that it would help - these sections are enclosed in brackets (), and would not of course appear were the program to be run.

(c) METHOD USED - A description of the programming techniques used in the Case Study. This is not intended to be a section which deals with programming style. Rather it is concerned with such things as how the data is organised, both on disk and within the program, and the approaches taken in handling that data.

(d) STRUCTURE CHART - A diagrammatic view of the way in which the parts of the program fit together. The programs have been written using a 'structured' approach, the essence of which is the avoidance of GOTO and GOSUB statements unless they are absolutely necessary. (The author is not a member of the 'GOTO-less programming' school; rather he views the GOTO statement in much the same way as alcohol - beneficial in small doses, but the more you have the harder it is to see what's going on!)

In each case then, the Case Study program comprises a 'main' program composed very largely on calls of procedures and functions. Readers who are either green or rusty concerning this aspect of programming in BBC BASIC are advised to refer to the appropriate sections of the User Guide. In essence a procedure call - which is of the form PROC<name> or FN<name> - causes a section of code - identified by its first line of DEFPROC<name> or DEFFN<name> - to be obeyed. This technique of 'structured' programming has been covered in many books, and readers unfamiliar with the concept could only profit by looking more deeply into it. A study of the subject is rather beyond the scope of this book, however, and will not be mentioned explicitly again; hopefully the Case Study programs will themselves speak adequately enough of its virtues.

(e) FULL PROGRAM LISTING - The code of the Case Study program. Each of the Case Studies is implemented entirely in BBC BASIC. This is certainly not to say that Assembly Language has no part to play in programming for disks, because it can be employed with considerable profit. By eliminating it from the Case Studies, however, it is hoped that the programs will have greater meaning to the widest range of readers.

(f) ANNOTATED PROGRAM LISTING - The listings are also devoid of REM statements. It has to be stated at once that this approach is not being advocated as good programming practice - quite the opposite in fact. However, in defence of this heinous sin it is argued that filling a program with REM statements is not the best method of presenting it to the reader of a book. Consequently, as with the sample dialogues, the annotated listings punctuate the program code with explanatory text making, in these circumstances, REM statements superfluous.

One particular comment which appears at the head of every Case Study program listing is a summary of the variables used within that program, together with a short statement as to their role. Only 'global' variables are mentioned - that is, variables which have a meaning throughout the whole program. Variables which are declared within a procedure as LOCAL are dealt with in the text describing that procedure.

As a visual aid, both the full and annotated listings have been produced using the LISTO7 facility of BBC BASIC. This feature inserts spaces into a program listing so as to highlight the structure of REPEAT and FOR loops. Readers typing the Case Study programs into their BBC Micro should omit these redundant spaces since they occupy space in RAM, albeit not very much. This trouble, and quite a bit of time, can be avoided by purchasing the Case Studies disk. This is available separately and contains all of the Case Study programs, together with an example database of articles published in the 'Micro User' journal.

(g) ROOMS FOR IMPROVEMENT - The Case Studies do not claim to be perfect in every way. Any red-blooded computer user will find ways in which the programs could be modified so as to be more suited to their own tastes. This final section of the Case Study documentation, then, suggests areas in which the reader may care to experiment.

CASE STUDY 1
TELLY – A program for storing names and telephone numbers in a serial file

TELLY - FUNCTION

The program enables a user to create a serial file containing up to 200 entries. Each entry comprises two items of data:

(i) A name
(ii) A telephone number

Once created, a number of different operations may be carried out upon this file of data:

* EXTEND the number of entries
* MODIFY an entry
* SEARCH for a particular entry
* LIST all of the entries

The limitation of 200 entries is purely arbitrary and could be changed to something much larger. Also to keep the non-disk programming to a reasonable level in this initial Case Study, the MODIFY and SEARCH functions will only operate by being given a full 'name' to identify an entry.

TELLY - SAMPLE DIALOGUE

(The Case Study program uses the CLS function on a number of occasions to clear the display screen prior to the output of information. In particular the screen is always cleared before the menu is output. In what follows, the appearance of a 'Screen Cleared' message indicates that at that point in a run of the program the screen would be cleared.)

```
>CHAIN "TELLY"
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
CREATE a New File (Y or N)?Y

Name of File to be CREATED?DIRECT

Input data items (RETURN to finish)

Name?ANNE
Tel. No.?01-424-5647

Name?MARGARET
Tel. No.?0231-985645

Name?ELIZABETH
Tel. No.?01-545-9867

Name?   <---- (Hit RETURN key to finish list)
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
1. EXTEND the Directory
2. MODIFY a Directory Entry
3. SEARCH the Directory
4. LIST the Directory
5. QUIT and SAVE Directory

Which Function (1-5)?1

Name?DIANA
Tel. No.?0705-827861

Name?  <----(RETURN)
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
1. EXTEND the Directory
2. MODIFY a Directory Entry
3. SEARCH the Directory
4. LIST the Directory
5. QUIT and SAVE Directory

Which Function (1-5)?4
```

```
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
(The listing shown below appears to be in one long stream. In practice,
'Page mode' (VDU 14) is used so that when the bottom of the screen is
reached the listing pauses until SHIFT is pressed.)

LISTING of Telephone Directory

Name :    ANNE
Tel. No. :01-424-5647

Name :    MARGARET
Tel. No. :0231-985645

Name :    ELIZABETH
Tel. No. :01-545-9867

Name :    DIANA
Tel. No. :0705-827681

Hit any key to continue  X <---(or anything else)
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
1. EXTEND the Directory
2. MODIFY a Directory Entry
3. SEARCH the Directory
4. LIST the Directory
5. QUIT and SAVE Directory

Which Function (1-5)?3

SEARCHING for which name?MARGARET

Name :    MARGARET
Tel. No. :0231-985645

Hit any key to continue X
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
1. EXTEND the Directory
2. MODIFY a Directory Entry
3. SEARCH the Directory
4. LIST the Directory
5. QUIT and SAVE Directory

Which Function (1-5)?2

Entry to be MODIFIED?DIANA

Name?DIANA & CHARLES
Tel. No.?Ex-directory
```

```
ooooooooooooooooooooooooooooooooooooooooooScreen Cleared
1. EXTEND the Directory
2. MODIFY a Directory Entry
3. SEARCH the Directory
4. LIST the Directory
5. QUIT and SAVE Directory

Which Function (1-5)?3

SEARCHING for which name?PHILIP

Name not found

Hit any key to continue X
ooooooooooooooooooooooooooooooooooooooooooScreen Cleared
1. EXTEND the Directory
2. MODIFY a Directory Entry
3. SEARCH the Directory
4. LIST the Directory
5. QUIT and SAVE Directory

Which Function (1-5)?5

Exit from TELLY
>
```

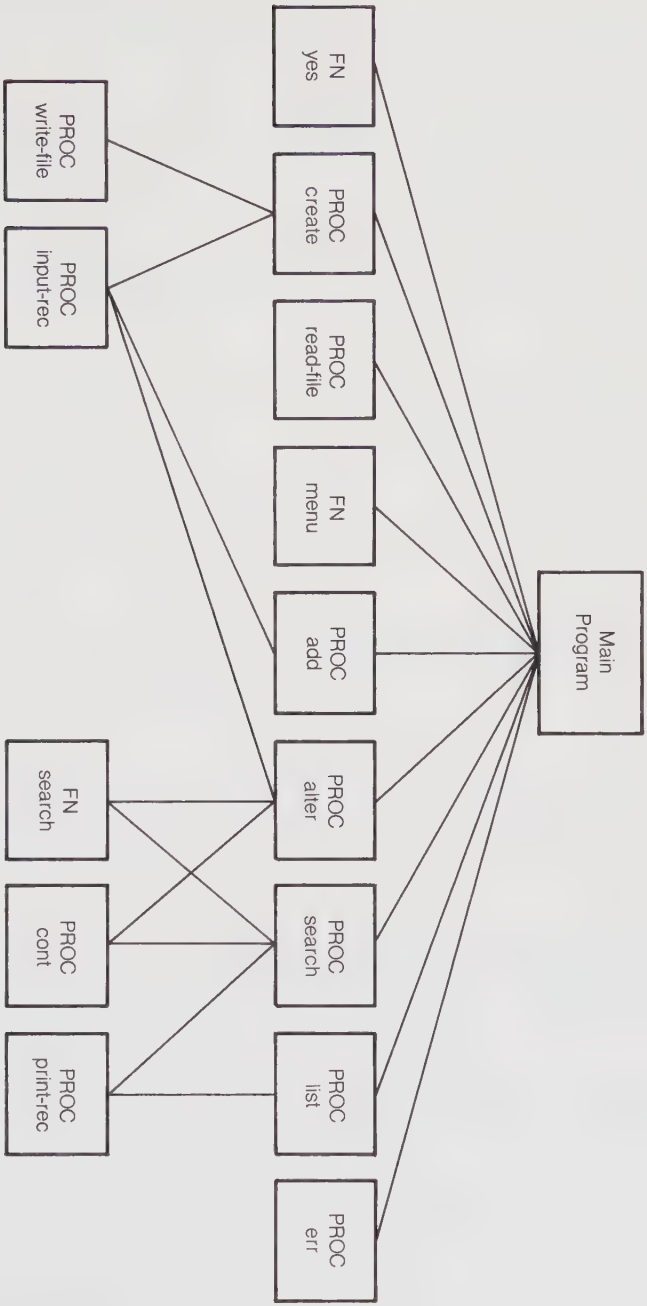
TELLY - METHOD USED

Since this is a program illustrating the use of a serial file, we are constrained to reading the entries in sequence. Each entry comprises two strings, first the name then the telephone number. Given that the facility for modification and searching are to be provided, the most efficient technique is to read the whole of the data file into RAM and operate on it there. This approach obviously means that the size of the file is constrained to the amount of available RAM.

When the file is created the user inputs a number of data-pairs comprising name and telephone number. Both are input as character strings. The information is held in a pair of string arrays until input is complete, at which point the number of pairs is known. A file is then created with its contents being as follows:

```
Number of pairs, N
Name 1
Telephone Number 1
Name 2
Telephone Number 2

        |
        |
        |

Name N
Telephone Number N
```

Any operation other than file creation will cause this sequence to take place in reverse - the 'number of pairs' value is input, and then the data-pairs read into the arrays from which they will be accessed.


Main Program
FN yes
PROC create
PROC read-file
FN menu
PROC add
PROC alter
PROC search
PROC list
PROC err
PROC write-file
PROC input-rec
FN search
PROC cont
PROC print-rec

TELLY - FULL PROGRAM LISTING

```
 10 REM****************
 20 REM* CASE STUDY 1 *
 30 REM****************
 40 max%=200: DIM name$(max%+1),tel$(max%+1)
 50 ON ERROR PROCerr: END

 60 CLS
 70  IF  FNyes("CREATE  a  New  File")  THEN  PROCcreate
      ELSE PROCread_file
 80 REPEAT
 90   func%=FNmenu
100   IF func%=1 PROCadd
110   IF func%=2 PROCalter
120   IF func%=3 PROCsearch
130   IF func%=4 PROClist
140   UNTIL func%=5
150 PROCwrite_file(fnam$): CLOSE#fo%: PRINT'"Exit from TELLY"': END

160 DEFPROCadd
170 REPEAT
180   n%=n%+1: PROCinput_rec(n%)
190   UNTIL name$(n%)="": n%=n%-1
200 ENDPROC

210 DEFPROCalter
220 INPUT'"Entry to be MODIFIED",snam$
230 pos%=FNsearch(snam$):  IF pos%<>0 PROCinput_rec(pos%)
      ELSE PROCcont
240 ENDPROC

250 DEFPROCsearch
260 INPUT'"SEARCHING for which name",snam$
270 pos%=FNsearch(snam$): IF pos%<>0 PROCprint_rec(pos%)
280 PROCcont
290 ENDPROC

300 DEFPROClist
310 CLS: VDU14: PRINT'"LISTING of Telephone Directory"'
320 FOR i%=1 TO n%
330   PROCprint_rec(i%)
340   NEXT: PROCcont: VDU15
350 ENDPROC

360 DEFPROCcreate: n%=0
370 REPEAT: INPUT'"Name of File to be CREATED",fnam$: UNTIL fnam$<>""
380 PRINT'"Input data items (RETURN to finish)"
390 REPEAT
400   n%=n%+1:PROCinput_rec(n%)
```

```
  410   UNTIL name$(n%)=""
  420 n%=n%-1: PROCwrite_file(fnam$)
  430 ENDPROC

  440 DEFPROCread_file: LOCAL i%: i%=0
  450 REPEAT: INPUT'"Name of File to be used",fnam$: UNTIL fnam$<>""
  460 fi%=OPENIN(fnam$)
  470 INPUT#fi%,n%
  480 REPEAT
  490   i%=i%+1: INPUT#fi%,name$(i%),tel$(i%)
  500   UNTIL EOF#fi%: CLOSE#fi%
  510 ENDPROC

  520 DEFPROCwrite_file(fn$): LOCAL i%
  530 fo%=OPENOUT(fn$)
  540 PRINT#fo%,n%
  550  FOR i%=1 TO n%
  560   PRINT#fo%,name$(i%),tel$(i%)
  570   NEXT: CLOSE#fo%
  580 ENDPROC

  590 DEFPROCinput_rec(p%)
  600  IF  p%>max%   PRINT'"**   No  more  room  **":
       PROCcont: name$(p%)="": ENDPROC
  610 INPUT'"Name",name$(p%)
  620 IF name$(p%)<>"" THEN INPUT"Tel. No.",tel$(p%)
  630 ENDPROC

  640 DEFPROCprint_rec(p%)
  650 PRINT'"Name :    ";name$(p%)'"Tel. No. :";tel$(p%)'
  660 ENDPROC

  670 DEFFNsearch(s$): LOCAL i%: i%=0
  680 REPEAT
  690   i%=i%+1
  700   UNTIL name$(i%)=s$ OR i%=n%
  710 IF i%=n% AND name$(i%)<>s$ PRINT'"Name  not found"':
      =0: ELSE =i%

  720 DEFFNmenu: LOCAL fc%
  730 CLS: PRINT'"1. EXTEND the Directory"
  740 PRINT"2. MODIFY a Directory Entry"
  750 PRINT"3. SEARCH the Directory"
  760 PRINT"4. LIST the Directory"
  770 PRINT"5. QUIT and SAVE Directory"'
  780 REPEAT: INPUT"Which Function (1-5)",fc%
  790   UNTIL fc%>0 AND fc%<6: =fc%

  800 DEFFNyes(st$): LOCAL ans$
  810 REPEAT: PRINT'st$;" (Y or N)";: INPUTans$
```

```
  820   UNTIL ans$="Y" OR ans$="y" OR ans$="N" OR ans$="n"
  830 =ans$="Y" OR ans$="y"

  840 DEFPROCcont: LOCAL key%
  850 PRINT'"Hit any alpha key to continue";: key%=GET
  860 ENDPROC

  870 DEFPROCerr
  880 PRINT'"Error ";ERR;" in Line "; ERL: REPORT: PRINT
  890 CLOSE#0
  900 ENDPROC
```

TELLY - ANNOTATED PROGRAM LISTING

List of Global Variables used:

ARRAYS

name$	These two arrays contain names
tel$	and corresponding telephone numbers

STRINGS

fnam$	Filename for OPENIN/OPENOUT
snam$	Name for SEARCH/MODIFY functions

INTEGERS

n%	Position in arrays of last entry
fi%	Channel number of input file
fo%	Channel number of output file
func%	Menu function selected
max%	Maximum number of entries
pos%	Position of located entry after search

```
   10 REM***************
   20 REM* CASE STUDY 1 *
   30 REM***************
```

(The arrays used for holding file entries are declared first. Variable max% is all that has to be altered to give files of different sizes.)

```
   40 max%=200: DIM name$(max%+1),tel$(max%+1)
   50 ON ERROR PROCerr: END
```

(Either a new file is to be created, or an existing file is to be used. After either of the two, the arrays name$ and tel$ contain the file entries.)

```
   60 CLS
   70 IF FNyes("CREATE a New File") THEN PROCcreate
        ELSE PROCread_file
```

(The remainder of the main program repeatedly presents the menu, requiring a function to be selected. This function is carried out by calling the appropriate procedure, the whole process finishing when QUIT (function 5) is selected.)

```
   80 REPEAT
   90   func%=FNmenu
  100   IF func%=1 PROCadd
  110   IF func%=2 PROCalter
  120   IF func%=3 PROCsearch
  130   IF func%=4 PROClist
  140   UNTIL func%=5
```

```
  150 PROCwrite_file(fnam$): CLOSE#fo%: PRINT'"Exit from TELLY"': END
```

(The procedures to perform the different menu operations begin here. Procedure 'add' extends the directory by adding new names and telephone numbers into the arrays from position n% onwards. The list is terminated by a 'null' name.)

```
  160 DEFPROCadd
  170 REPEAT
  180   n%=n%+1: PROCinput_rec(n%)
  190   UNTIL name$(n%)="": n%=n%-1
  200 ENDPROC
```

(Procedure 'alter' enables the modification of an entry. This entry has to be identified by the input of the whole of the name$ part. If the entry exists then it has to be completely re-entered. The function FNsearch will return the value 0 if snam$ cannot be found.)

```
  210 DEFPROCalter
  220 INPUT'"Entry to be MODIFIED",snam$
  230 pos%=FNsearch(snam$):  IF pos%<>0 PROCinput_rec(pos%)
        ELSE PROCcont
  240 ENDPROC
```

(Procedure 'search' works in much the same way as procedure 'alter', the difference being that the entry is only displayed when found.)

```
  250 DEFPROCsearch
  260 INPUT'"SEARCHING for which name",snam$
  270 pos%=FNsearch(snam$): IF pos%<>0 PROCprint_rec(pos%)
  280 PROCcont
  290 ENDPROC
```

(Procedure 'list' displays every entry held in the arrays as far as position n%. Since this would normally mean a display that was too rapid to read, this function employs 'Page mode' - VDU14 and VDU15 - to let the user control the listing.)

```
  300 DEFPROClist
  310 CLS: VDU14: PRINT'"LISTING of Telephone Directory"'
  320 FOR i%=1 TO n%
  330   PROCprint_rec(i%)
  340   NEXT: PROCcont: VDU15
  350 ENDPROC
```

(Procedure 'create' will be invoked on the initial entry to the program. It repeatedly asks for name/telephone number inputs until a 'null' name is received. The file, fnam$ is created by calling procedure 'write_file'; n% holds the number of entries that have been supplied.)

```
360 DEFPROCcreate: n%=0
370 REPEAT: INPUT'"Name of File to be CREATED",fnam$: UNTIL fnam$<>""
380 PRINT'"Input data items (RETURN to finish)"
390 REPEAT
400   n%=n%+1:PROCinput_rec(n%)
410   UNTIL name$(n%)=""
420 n%=n%-1: PROCwrite_file(fnam$)
430 ENDPROC
```

(Procedure 'read_file' inputs the whole of an existing file into arrays name$ and tel$. After ascertaining the file name, it opens the file and inputs the first value - the number of entries - into n%. Thereafter the data pairs are input up to the end of the disk file.)

```
440 DEFPROCread_file: LOCAL i%: i%=0
450 REPEAT: INPUT'"Name of File to be used",fnam$: UNTIL fnam$<>""
460 fi%=OPENIN(fnam$)
470 INPUT#fi%,n%
480 REPEAT
490   i%=i%+1: INPUT#fi%,name$(i%),tel$(i%)
500   UNTIL EOF#fi%: CLOSE#fi%
510 ENDPROC
```

(Procedure 'write_file' sends the relevant contents of arrays name$ and tel$ to the file specified by the parameter fn$. On completion of this operation the file is closed.)

```
520 DEFPROCwrite_file(fn$): LOCAL i%
530 fo%=OPENOUT(fn$)
540 PRINT#fo%,n%
550  FOR i%=1 TO n%
560   PRINT#fo%,name$(i%),tel$(i%)
570   NEXT: CLOSE#fo%
580 ENDPROC
```

(Procedure 'input_rec' will ask the user to input a name. If the name supplied is not 'null' then the procedure follows on to input a telephone number. Both of these items are inserted into their appropriate arrays at the position given by parameter p%. If the arrays are full an error message is output and no input requested.)

```
590 DEFPROCinput_rec(p%)
600  IF p%>max% PRINT'"**  No more room **":
       PROCcont: name$(p%)="": ENDPROC
610 INPUT'"Name",name$(p%)
620 IF name$(p%)<>"" THEN INPUT"Tel. No.",tel$(p%)
630 ENDPROC
```

(Procedure 'print_rec' displays the name/telephone number entry specified by parameter p%.)

```
  640 DEFPROCprint_rec(p%)
  650 PRINT'"Name :    ";name$(p%)'"Tel. No. :";tel$(p%)'
  660 ENDPROC
```

(Function 'search' scans through the array name$ trying to find a match for its supplied parameter s$. If it succeeds, the position in the array of the located name is returned as the function's value. If the name is not found, an error message is output and the value of the function set to zero.)

```
  670 DEFFNsearch(s$): LOCAL i%: i%=0
  680 REPEAT
  690   i%=i%+1
  700   UNTIL name$(i%)=s$ OR i%=n%
  710 IF i%=n% AND name$(i%)<>s$ PRINT'"Name  not found"':
      =0: ELSE =i%
```

(Function 'menu' displays the available operations, and asks the user to make a choice. After ensuring that the selection is valid, the function's value is set to that selection number.)

```
  720 DEFFNmenu: LOCAL fc%
  730 CLS: PRINT'"1. EXTEND the Directory"
  740 PRINT"2. MODIFY a Directory Entry"
  750 PRINT"3. SEARCH the Directory"
  760 PRINT"4. LIST the Directory"
  770 PRINT"5. QUIT and SAVE Directory"'
  780 REPEAT: INPUT"Which Function (1-5)",fc%
  790   UNTIL fc%>0 AND fc%<6: =fc%
```

(A final group of utilities. Function 'yes' displays its string parameter st$ until a response 'Y/y' or 'N/n' is made by the user; the function's value is TRUE if 'Y/y' is input.
Procedure 'cont' outputs a little message and waits for a key to be pressed before returning to the main program. A look at the sample dialogue will show where this is needed - normally to allow the user to read what has been output before the screen is cleared!
Procedure 'err' is only called if the BBC Operating System detects that something is wrong. If this happens the standard error information is output and, most importantly, all files are closed.)

```
  800 DEFFNyes(st$): LOCAL ans$
  810 REPEAT: PRINT'st$;" (Y or N)";: INPUTans$
  820   UNTIL ans$="Y" OR ans$="y" OR ans$="N" OR ans$="n"
  830 =ans$="Y" OR ans$="y"
  840 DEFPROCcont: LOCAL key%
  850 PRINT'"Hit any alpha key to continue";: key%=GET
  860 ENDPROC
  870 DEFPROCerr
  880 PRINT'"Error ";ERR;" in Line "; ERL: REPORT: PRINT
```

```
  890 CLOSE#0
  900 ENDPROC
```

TELLY - ROOMS FOR IMPROVEMENT

Some possibilities for improving this program will no doubt have occurred to even the most somnolent of readers. There follows a short list which may give one or two additional ideas:

It is only possible to search/modify by quoting a full name. Allowing references which quote only the first characters of a name could be implemented.

+++++

There may be duplicate names in the list, in which case the program as it stands will only locate the first. Ensure that all matching names are found.

+++++

It is not possible to search for a given telephone number; provide this facility.

+++++

The telephone number is held as a string variable. Store space can be saved by holding it as an integer variable. Is this worth doing?

CASE STUDY 2
RTELLY – A program for storing names and telephone numbers in a random access file

RTELLY - FUNCTION

The program enables a user to create a random access file containing up to 200 entries. Each entry comprises two items of data:

(i) A name, of maximum length 15 characters
(ii) A telephone number, of maximum length 11 characters

Once created, a number of different operations may be carried out upon this file of data:

* EXTEND the number of entries
* MODIFY an entry
* SEARCH for a particular entry
* LIST all of the entries

RTELLY - SAMPLE DIALOGUE

(The user's view of this Case Study program is really the same as that for Case Study 1. What is presented below, therefore, is just that part of the dialogue which shows what happens when the directory becomes full. To achieve this output with minimal effort the Case Study program was modified (in line 40) so as to limit the size of the directory file to five entries only.)

```
>CHAIN "RTELLY"

○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○Screen Cleared
CREATE a New File (Y or N)?Y

Name of File to be CREATED?RDIRECT

Input data items (RETURN to finish)

Name?ANNE
Tel. No.?01-424-5647

Name?MARGARET
Tel. No.?0231-985645

Name?ELIZABETH
Tel. No.?01-545-9867

Name?DIANA & CHARLES
Tel. No.?Ex-Directory

Name?JENNIFER
Tel. No.?3290-898762

** No more room **   (At this point the directory is full - any further
                       attempt to extend it must be prevented.)

Hit any alpha key to continue   X
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○Screen Cleared
1. EXTEND the Directory
2. MODIFY a Directory Entry
3. SEARCH the Directory
4. LIST the Directory
5. QUIT

Which Function (1-5)?2

Entry to be MODIFIED?JENNIFER
```

(This is the last entry in the file. The program must of course allow it to be modified, but this does make checking for a full directory that much harder - a '**No more room**' message is not wanted at this point.)

```
Name?JENNIFER MARY COLEMAN
Tel. No.?3290-898762

ooooooooooooooooooooooooooooooooooooooooooooScreen Cleared
1. EXTEND the Directory
2. MODIFY a Directory Entry
3. SEARCH the Directory
4. LIST the Directory
5. QUIT

Which Function (1-5)?1

** No more room **

Hit any alpha key to continue
ooooooooooooooooooooooooooooooooooooooooooooScreen Cleared
1. EXTEND the Directory
2. MODIFY a Directory Entry
3. SEARCH the Directory
4. LIST the Directory
5. QUIT

Which Function (1-5)?5

>RUN                             (Run the program again)

ooooooooooooooooooooooooooooooooooooooooooooScreen Cleared
CREATE a New File (Y or N)?N

Name of File to be used?RDIRECT   (Use the file just created)

ooooooooooooooooooooooooooooooooooooooooooooScreen Cleared
1. EXTEND the Directory
2. MODIFY a Directory Entry
3. SEARCH the Directory
4. LIST the Directory
5. QUIT

Which Function (1-5)?1

** No more room **   (This can only be achieved by performing a check
                       as soon as the file is opened.)

Hit any alpha key to continue  X
```

```
ooooooooooooooooooooooooooooooooooooooooooooScreen Cleared
1. EXTEND the Directory
2. MODIFY a Directory Entry
3. SEARCH the Directory
4. LIST the Directory
5. QUIT

Which Function (1-5)?4

ooooooooooooooooooooooooooooooooooooooooooooScreen Cleared
LISTING of Telephone Directory

Name :    ANNE
Tel. No. :01-424-5647

Name :    MARGARET
Tel. No. :0231-985645

Name :    ELIZABETH
Tel. No. :01-545-9867

Name :    DIANA & CHARLES
Tel. No. :Ex-Director      <-- (Truncated to)

Name :    JENNIFER MARY C  <-- (15 characters!)
Tel. No. :3290-898762

Hit any alpha key to continue   X
ooooooooooooooooooooooooooooooooooooooooooooScreen Cleared
1. EXTEND the Directory
2. MODIFY a Directory Entry
3. SEARCH the Directory
4. LIST the Directory
5. QUIT

Which Function (1-5)?5
Exit from RTELLY
>
```

RTELLY - METHOD USED

The telephone directory is held on disk as a random access file, its contents being

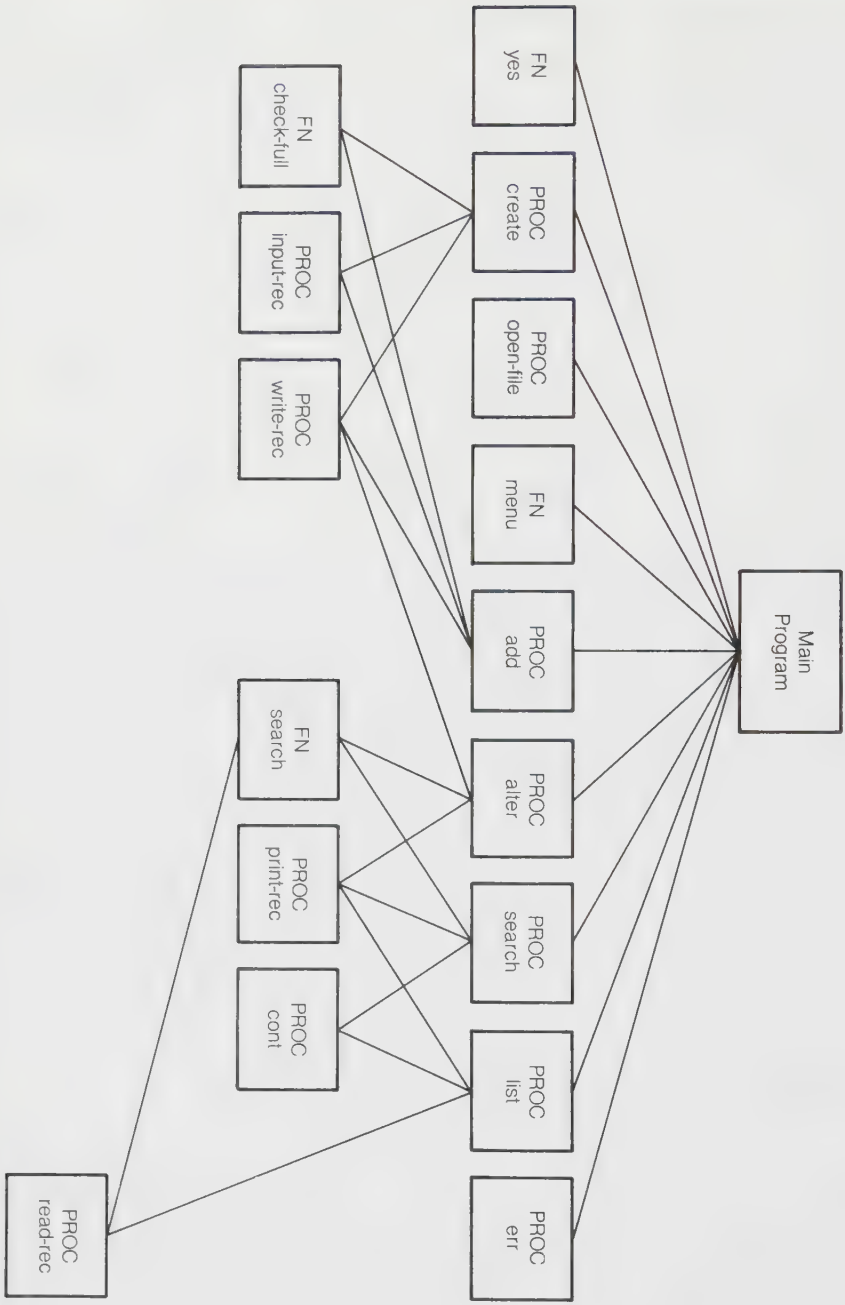
```
Number of pairs, N
Name 1
Telephone Number 1
Name 2
Telephone Number 2

        |
        |
        |

Name N
Telephone Number N
```

A name is constrained to a maximum length of 15 bytes, a telephone number to a maximum of 11 bytes. A file entry for each of these items is created as a fixed-length record, whether or not the full amount of space is used. Since both are string items an additional two bytes is required in each case, so that the total disk space occupied by a name is 17 bytes and by a telephone number 13 bytes. Data pairs are thus spaced at 30-byte intervals throughout the file. Since the integer value, N, giving the number of defined data pairs, is the first item of the file (needing the five bytes for an integer) the content of the file can be illustrated thus:

Setting PTR# by means of a statement like

```
PTR#fu%=5+(no%-1)*30
```

enables any particular name to be randomly accessed by means of its numerical position in the directory (no%). The program of Case Study 2 works on this basis. Only the name and corresponding telephone number which are currently needed are held in store at any time, being read from the disk file when required. If any item is to be modified, PTR# is set using exactly the same technique before the new data pair are written to the appropriate place on disk.


Main Program
FN yes
PROC create
PROC open-file
FN menu
PROC add
PROC alter
PROC search
PROC list
PROC err
FN check-full
PROC input-rec
PROC write-rec
FN search
PROC print-rec
PROC cont
PROC read-rec

RTELLY - FULL PROGRAM LISTING

```
 10 REM****************
 20 REM* CASE STUDY 2 *
 30 REM****************
 40 max%=200: start%=5: FULL%=FALSE: fu%=0
 50 ON ERROR PROCerr: END
 60 CLS

 70 IF FNyes("CREATE a New File") THEN PROCcreate  ELSE
     PROCopen_file
 80 REPEAT
 90   func%=FNmenu
100   IF func%=1 PROCadd
110   IF func%=2 PROCalter
120   IF func%=3 PROCsearch
130   IF func%=4 PROClist
140   UNTIL func%=5
150 CLOSE#fu%: PRINT'"Exit from RTELLY"': END

160 DEFPROCadd
170 IF FULL% PROCcheck_full: ENDPROC
180 REPEAT
190   PROCinput_rec
200   IF name$<>"" THEN n%=n%+1: PROCwrite_rec(n%):
       PROCcheck_full
210   UNTIL name$="" OR FULL%: PTR#fu%=0: PRINT#fu%,n%
220 ENDPROC

230 DEFPROCalter
240 INPUT'"Entry to be MODIFIED",snam$
250 pos%=FNsearch(snam$):  IF  pos%<>0  PROCinput_rec:
     PROCwrite_rec(pos%)
260 ENDPROC

270 DEFPROCsearch
280 INPUT'"SEARCHING for which name",snam$
290 pos%=FNsearch(snam$): IF pos%<>0 PROCprint_rec
300 PROCcont
310 ENDPROC

320 DEFPROClist
330 CLS: VDU14: PRINT'"LISTING of Telephone Directory"'
340 FOR i%=1 TO n%
350   PROCread_rec(i%): PROCprint_rec
360   NEXT: PROCcont: VDU15
370 ENDPROC
```

```
380 DEFPROCcreate: n%=0
390 REPEAT: INPUT'"Name of File to be CREATED",fnam$: UNTIL fnam$<>""
400 fo%=OPENOUT(fnam$): PRINT#fo%,0
410 FOR i%=1 TO max%
420   PRINT#fo%,STRING$(15,"?"),STRING$(11,"!")
430   NEXT: CLOSE#fo%
440 fu%=OPENUP(fnam$): REM use OPENIN with BASIC I
450 PRINT'"Input data items (RETURN to finish)"
460 REPEAT
470   PROCinput_rec
480   IF name$<>"" n%=n%+1: PROCwrite_rec(n%): PROCcheck_full
490   UNTIL name$="" OR FULL%
500 PTR#fu%=0: PRINT#fu%,n%
510 ENDPROC

520 DEFPROCopen_file
530 REPEAT: INPUT'"Name of File to be used",fnam$: UNTIL fnam$<>""
540 fu%=OPENUP(fnam$): REM use OPENIN with BASIC I
550 INPUT#fu%,n%: IF start%+n%*30>=EXT#fu% THEN FULL%=TRUE
560 ENDPROC

570 DEFPROCread_rec(no%): LOCAL ptr%
580 ptr%=start%+(no%-1)*30: PTR#fu%=ptr%
590 INPUT#fu%,name$: PTR#fu%=ptr%+17: INPUT#fu%,tel$
600 ENDPROC

610 DEFPROCwrite_rec(no%): LOCAL ptr%
620 ptr%=start%+(no%-1)*30: PTR#fu%=ptr%
630 PRINT#fu%,LEFT$(name$,15): PTR#fu%=ptr%+17:
     PRINT#fu%,LEFT$(tel$,11)
640 ptr%=ptr%+30: IF ptr%>=EXT#fu% THEN FULL%=TRUE
650 ENDPROC

660 DEFPROCinput_rec
670 INPUT'"Name",name$
680 IF name$<>"" THEN INPUT"Tel. No.",tel$
690 ENDPROC

700 DEFPROCprint_rec
710 PRINT'"Name :    ";name$'"Tel. No. :";tel$
720 ENDPROC

730 DEFFNsearch(s$): LOCAL i%: i%=0
740 REPEAT
750   i%=i%+1: PROCread_rec(i%)
760   UNTIL name$=s$ OR i%=n%
770 IF i%=n% AND name$<>s$ THEN PRINT'"Name not found"':
     =0: ELSE =i%
```

```
780 DEFFNmenu: LOCAL fc%
790 CLS: PRINT'"1. EXTEND the Directory"
800 PRINT"2. MODIFY a Directory Entry"
810 PRINT"3. SEARCH the Directory"
820 PRINT"4. LIST the Directory"
830 PRINT"5. QUIT"'
840 REPEAT: INPUT"Which Function (1-5)",fc%
850   UNTIL fc%>0 AND fc%<6: =fc%

860 DEFPROCcheck_full
870 IF FULL% PRINT'"** No more room **": PROCcont
880 ENDPROC

890 DEFFNyes(st$): LOCAL ans$
900 REPEAT: PRINT'st$;" (Y or N)";: INPUTans$
910   UNTIL ans$="Y" OR ans$="y" OR ans$="N" OR ans$="n"
920 =ans$="Y" OR ans$="y"

930 DEFPROCcont: LOCAL key%
940 PRINT'"Hit any alpha key to continue";: key%=GET
950 ENDPROC

960 DEFPROCerr
970 PRINT'"Error ";ERR;" in Line "; ERL: REPORT: PRINT
980 IF fu%<>0 THEN PTR#fu%=0: PRINT#fu%,n%: CLOSE#0
990 ENDPROC
```

RTELLY - ANNOTATED PROGRAM LISTING

List of Global Variables used:

STRINGS	
name$	Name, and corresponding
tel$	telephone number
fnam$	Filename for OPENIN/OPENOUT/OPENUP
snam$	Name for SEARCH/MODIFY functions
INTEGERS	
n%	Number of defined directory entries
fu%	Channel number of file to be updated
fo%	Channel number of file to be created
func%	Menu function selected
max%	Maximum number of entries
pos%	Number of located entry after search
start%	Start position of 1st data pair
LOGICAL	
FULL%	TRUE if the directory is full, FALSE otherwise

```
   10 REM****************
   20 REM* CASE STUDY 2 *
   30 REM****************
```

(Not surprisingly, sections of this program are either identical or very similar to the program of Case Study 1. Comments are only provided, therefore, for those parts which are markedly different in either form or function.)

```
   40 max%=200: start%=5: FULL%=FALSE: fu%=0
```

(FULL% has to be initialised to indicate that the file is presumed non-full unless found to be otherwise. Variable start% is used to set PTR# to the start position of the first data pair in the file and is thus set to 5. Use of a variable name in the program, rather than a value, makes the places where this value needs to be used a little more comprehensible.)

```
   50 ON ERROR PROCerr: END
   60 CLS
```

(If an existing file is to be used it does not need to be input, just opened for updating.)

```
   70 IF FNyes("CREATE a New File") THEN PROCcreate  ELSE
       PROCopen_file
   80 REPEAT
   90   func%=FNmenu
```

```
  100   IF func%=1 PROCadd
  110   IF func%=2 PROCalter
  120   IF func%=3 PROCsearch
  130   IF func%=4 PROClist
  140   UNTIL func%=5
  150 CLOSE#fu%: PRINT'"Exit from RTELLY"': END
```

(Procedure 'add' checks to ensure that space is available in the file. If all is well, then the new data pair is input by the user and written to the disk file at the position calculated for entry n%. The counter value is also updated and written to the file at its position, byte 0.)

```
  160 DEFPROCadd
  170 IF FULL% PROCcheck_full: ENDPROC
  180 REPEAT
  190   PROCinput_rec
  200   IF name$<>"" THEN n%=n%+1: PROCwrite_rec(n%):
        PROCcheck_full
  210   UNTIL name$="" OR FULL%: PTR#fu%=0: PRINT#fu%,n%
  220 ENDPROC
```

(Procedure 'alter' carries out the modification of an entry by searching the file for the name which the user inputs. Assuming that it is found, the revised data pair is written back to the same disk position.)

```
  230 DEFPROCalter
  240 INPUT'"Entry to be MODIFIED",snam$
  250 pos%=FNsearch(snam$): IF pos%<>0 PROCinput_rec:
      PROCwrite_rec(pos%)
  260 ENDPROC
```

(Procedure 'search' locates the entry for which the user supplies a name, and displays both parts of it.)

```
  270 DEFPROCsearch
  280 INPUT'"SEARCHING for which name",snam$
  290 pos%=FNsearch(snam$): IF pos%<>0 PROCprint_rec
  300 PROCcont
  310 ENDPROC
```

(Procedure 'list' outputs the directory listing by reading individual entries from the file and displaying each in turn.)

```
  320 DEFPROClist
  330 CLS: VDU14: PRINT'"LISTING of Telephone Directory"'
  340 FOR i%=1 TO n%
  350   PROCread_rec(i%): PROCprint_rec
  360   NEXT: PROCcont: VDU15
  370 ENDPROC
```

(Procedure 'create' initially establishes a file of max% dummy data pairs having lengths 15 and 11 characters respectively. This file is closed (since it was opened for output only) and then reopened for updating. User-input entries are then written to the disk file at their calculated positions until the process is terminated or the file becomes full. Note that the file is not closed at this point; it remains open for further operations to be selected from the menu.)

```
380 DEFPROCcreate: n%=0
390 REPEAT: INPUT'"Name of File to be CREATED",fnam$: UNTIL fnam$<>""
400 fo%=OPENOUT(fnam$): PRINT#fo%,0
410 FOR i%=1 TO max%
420   PRINT#fo%,STRING$(15,"?"),STRING$(11,"!")
430   NEXT: CLOSE#fo%
440 fu%=OPENUP(fnam$): REM use OPENIN with BASIC I
450 PRINT'"Input data items (RETURN to finish)"
460 REPEAT
470   PROCinput_rec
480   IF name$<>"" n%=n%+1: PROCwrite_rec(n%): PROCcheck_full
490   UNTIL name$="" OR FULL%
500 PTR#fu%=0: PRINT#fu%,n%
510 ENDPROC
```

(Procedure 'open_file' asks the user for a filename and then opens that file for updating. In addition it discovers whether or not the file is full and sets FULL% accordingly.)

```
520 DEFPROCopen_file
530 REPEAT: INPUT'"Name of File to be used",fnam$: UNTIL fnam$<>""
540 fu%=OPENUP(fnam$): REM use OPENIN with BASIC I
550 INPUT#fu%,n%: IF start%+n%*30>=EXT#fu% THEN FULL%=TRUE
560 ENDPROC
```

(Procedure 'read_rec', using its parameter which provides the number of the entry needed, sets PTR# and takes the appropriate name$ from the file. PTR# is then positioned at the start of the corresponding telephone number, and that is read into tel$.)

```
570 DEFPROCread_rec(no%): LOCAL ptr%
580 ptr%=start%+(no%-1)*30: PTR#fu%=ptr%
590 INPUT#fu%,name$: PTR#fu%=ptr%+17: INPUT#fu%,tel$
600 ENDPROC
```

(Procedure 'write_rec' is the converse of procedure 'read-rec', sending name$ and tel$ to the position on disk for entry no%. Note that it also forces these data items to be truncated if their length is in excess of that allowed.)

```
610 DEFPROCwrite_rec(no%): LOCAL ptr%
620 ptr%=start%+(no%-1)*30: PTR#fu%=ptr%
```

```
  630 PRINT#fu%,LEFT$(name$,15): PTR#fu%=ptr%+17:
       PRINT#fu%,LEFT$(tel$,11)
  640 ptr%=ptr%+30: IF ptr%>=EXT#fu% THEN FULL%=TRUE
  650 ENDPROC
```

(The remainder of the program is either the same as Case Study 1, and documented there, or else is slightly new but obvious in the way it works.)

```
  660 DEFPROCinput_rec
  670 INPUT'"Name",name$
  680 IF name$<>"" THEN INPUT"Tel. No.",tel$
  690 ENDPROC
  700 DEFPROCprint_rec
  710 PRINT'"Name :    ";name$'"Tel. No. :";tel$
  720 ENDPROC
  730 DEFFNsearch(s$): LOCAL i%: i%=0
  740 REPEAT
  750   i%=i%+1: PROCread_rec(i%)
  760   UNTIL name$=s$ OR i%=n%
  770 IF i%=n% AND name$<>s$ THEN PRINT'"Name not found"':
       =0: ELSE =i%
  780 DEFFNmenu: LOCAL fc%
  790 CLS: PRINT'"1. EXTEND the Directory"
  800 PRINT"2. MODIFY a Directory Entry"
  810 PRINT"3. SEARCH the Directory"
  820 PRINT"4. LIST the Directory"
  830 PRINT"5. QUIT"'
  840 REPEAT: INPUT"Which Function (1-5)",fc%
  850   UNTIL fc%>0 AND fc%<6: =fc%
  860 DEFPROCcheck_full
  870 IF FULL% PRINT'"** No more room **": PROCcont
  880 ENDPROC
  890 DEFFNyes(st$): LOCAL ans$
  900 REPEAT: PRINT'st$;" (Y or N)";: INPUTans$
  910   UNTIL ans$="Y" OR ans$="y" OR ans$="N" OR ans$="n"
  920 =ans$="Y" OR ans$="y"
  930 DEFPROCcont: LOCAL key%
  940 PRINT'"Hit any alpha key to continue";: key%=GET
  950 ENDPROC
  960 DEFPROCerr
  970 PRINT'"Error ";ERR;" in Line "; ERL: REPORT: PRINT
  980 IF fu%<>0 THEN PTR#fu%=0: PRINT#fu%,n%: CLOSE#0
  990 ENDPROC
```

ROOMS FOR IMPROVEMENT

The suggestions given at the end of Case Study 1 apply equally well to this program also. In addition:

Certain constant values are used throughout the program, viz:

15 - maximum length of name$
11 - maximum length of tel$
17 - length of name field on disk
30 - length of name/telephone number record on disk

These have been left as constants in the program to highlight the many instances in which they need to be used. An improvement to the readability of the program would be to assign these values to variables, e.g.

```
 45 rec_length%=30
640 ptr%=ptr%+rec_length%: IF ptr%>=EXT#fu% THEN FULL%=TRUE
```

This would have the added benefit of making the program much easier to modify. If for instance, an increase to the maximum record length was desired, only the new line 45 would need to be altered.

+++++

Allow the user to define the maximum lengths of name$ and tel$ when the directory file is created. This is not quite so simple as it sounds. Values for the two lengths could certainly be requested of the user when a new file is to be created, and employed without too much trouble in setting up the initial file. However, in order for those lengths to be used subsequently they have to be available every time the program is used. Asking the user to input them every time the program is run is rather messy - and confusing if a number of different directories have been set up. No, the obvious place to store the lengths of the data items is in the file itself, and that means a certain amount of rearrangement. As a clue, look at how the number of data pairs (n%) is stored and used. Good luck. It is no coincidence that the next chapter is about diagnostic aids!

+++++

Names and telephone numbers which are greater than the maximum lengths of 15 and 11 characters respectively have to be truncated (line 630). Modified records are given the same treatment. On the other hand, for records which are shorter than the maximum lengths spurious characters will be found in the unused positions in the name/telephone number fields. An improvement to the program which will make the file easier to examine would be to pad with spaces any fields which are shorter than the maximum length.

CASE STUDY 3
PATCHER – A friendly program for the dumping and patching of disk files

PATCHER - FUNCTION

The program aims to perform two complementary functions:

(i) to generate a file dump in which integer, real and string elements are clearly identifiable

(ii) to provide a simple means, with some error protection, for file patching

Format of the File Dump

As will have already been seen, the output produced by the *DUMP utility is fine if one is fluent in hexadecimal-speak, but needs some concentrated effort to interpret if one is normal. Since data files - with which we are primarily concerned - contain identifiable items, it is perfectly possible to produce a dump that prints these items in a more readable form. Thus integers, reals and strings are all displayed as themselves, together with the address at which their first (identifying) byte is located, and an indicator giving the type of item. For example,

```
0 i  5          (address 0, integer value 5)
5 s  ANNE       (address 5, string "ANNE")
```

```
B r  2.1          (address B, real value 2.1)
```

In addition, though, it is also necessary to display individual bytes. This is because, as we have seen with our example of a file dump produced by the *DUMP utility, an individual byte need not always be part of one of the three data types. Strings will not always use the whole of a fixed-length field, for instance, and yet the odd bytes at the end of the field have to be shown in a file dump. The Case Study program treats odd bytes by displaying them in the 2-digit hexadecimal form used by *DUMP, grouped eight or less to a line. For example,

```
10 b    3F 3F 3F      (a sequence of 3 bytes,
                       containing ASCII for "?")
```

(NOTE: The program is designed for use with files which contain sensible but incorrect data items of the types string, integer and real. Files which have totally random contents in this respect, such as machine-code programs, do not qualify. PATCHER will react to such files by generating equally random displays.)

Patching the File

A simple menu offers the user a choice of three functions:

(i) Display next section of the file
(ii) Patch over an item
(iii) Exit from the program

This part of the program tries to minimise the danger of making a file unreadable by incorrect patching. It constrains the user so that an item can only be replaced by another item of the same type, i.e. an integer by an integer, a real by a real, a string by a string. In addition a new string cannot occupy more room than the string that it replaces. Bytes can be patched individually and replaced by any other byte value.

PATCHER - SAMPLE DIALOGUE

```
>CHAIN "PATCHER"

°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
Name of file?RDIRECT   (The same file as used to show the output from
                           *DUMP in Chapter 7)
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
(The first section of  the  file is displayed, using the top 15 lines of
the display screen.)

   0 i  5                    (Address 0000, integer value 5)
   5 s  ANNE                 (Address 0005, string "ANNE")
   B b    3F 3F 3F 3F 3F     (Addresses 000B to 0015 contain
  10 b    3F 3F 3F 3F 3F 3F    the ASCII code 3F)
  16 s  01-424-5647
  23 s  MARGARET
  2D b    3F 3F 3F
  30 b    3F 3F 3F 3F
  34 s  0231-985645
  41 s  ELIZABETH
  4C b    3F 3F 3F 3F
  50 b    3F 3F
  52 s  01-545-9867
  5F s  DIANA & CHARLES
||||||||||||||||||||||||||||||||||||||| (Means more to come)

0. NEXT SECTION OF FILE RDIRECT
1. MAKE A CHANGE
2. QUIT

Which function (0-2)?1

Which (Start) Address?5   (This address is in hexadecimal)

Input a STRING:ANNIE

  4 chars maximum at this address  (Because the original string was
                                       4-character "ANNE")

Input a STRING:ANNA
```

```
ooooooooooooooooooooooooooooooooooooooooooScreen Cleared
   0 i  5
   5 s  ANNA              (New string now written to file)
   B b    3F 3F 3F 3F 3F
  10 b    3F 3F 3F 3F 3F 3F
  16 s  01-424-5647
  23 s  MARGARET
  2D b    3F 3F 3F
  30 b    3F 3F 3F 3F
  34 s  0231-985645
  41 s  ELIZABETH
  4C b    3F 3F 3F 3F
  50 b    3F 3F
  52 s  01-545-9867
  5F s  DIANA & CHARLES
||||||||||||||||||||||||||||||||||||||

0. NEXT SECTION OF FILE RDIRECT
1. MAKE A CHANGE
2. QUIT

Which function (0-2)?1

Which (Start) Address?0

Input an INTEGER value:4    (Only an integer is allowed)

ooooooooooooooooooooooooooooooooooooooooooScreen Cleared
   0 i  5
   5 s  ANNA
   B b    3F 3F 3F 3F 3F
  10 b    3F 3F 3F 3F 3F 3F
  16 s  01-424-5647
  23 s  MARGARET
  2D b    3F 3F 3F
  30 b    3F 3F 3F 3F
  34 s  0231-985645
  41 s  ELIZABETH
  4C b    3F 3F 3F 3F
  50 b    3F 3F
  52 s  01-545-9867
  5F s  DIANA & CHARLES
||||||||||||||||||||||||||||||||||||||

0. NEXT SECTION OF FILE RDIRECT
1. MAKE A CHANGE
2. QUIT

Which function (0-2)?0   (or RETURN)
```

```
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
  70 s  Ex-Director
  7D s  JENNIFER MARY C
  8E s  3290-898762
*END OF FILE**END OF FILE**END OF FILE*

0. NEXT SECTION OF FILE RDIRECT
1. MAKE A CHANGE
2. QUIT

Which function (0-2)?1

Which (Start) Address?7D

Input a STRING:JENNY COLEMAN

°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
  70 s  Ex-Director
  7D s  JENNY COLEMAN
  8C b    45 4A
  8E s  3290-898762
*END OF FILE**END OF FILE**END OF FILE*

0. NEXT SECTION OF FILE RDIRECT
1. MAKE A CHANGE
2. QUIT

Which function (0-2)?2

°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
Exit from PATCHER
>
```

PATCHER - METHOD USED

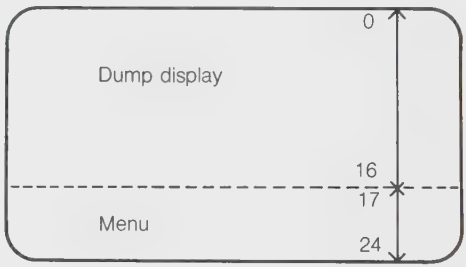
As far as techniques for disk programming are concerned, this Case Study shows how it is possible to manipulate the file pointer so as to read information from, and write information to, any position in the file being used. The heart of the program involves the recognition of integer, real and string quantities in the file by looking for the appropriate leading byte, namely

40	for an integer
FF	for a real value
00	for a string

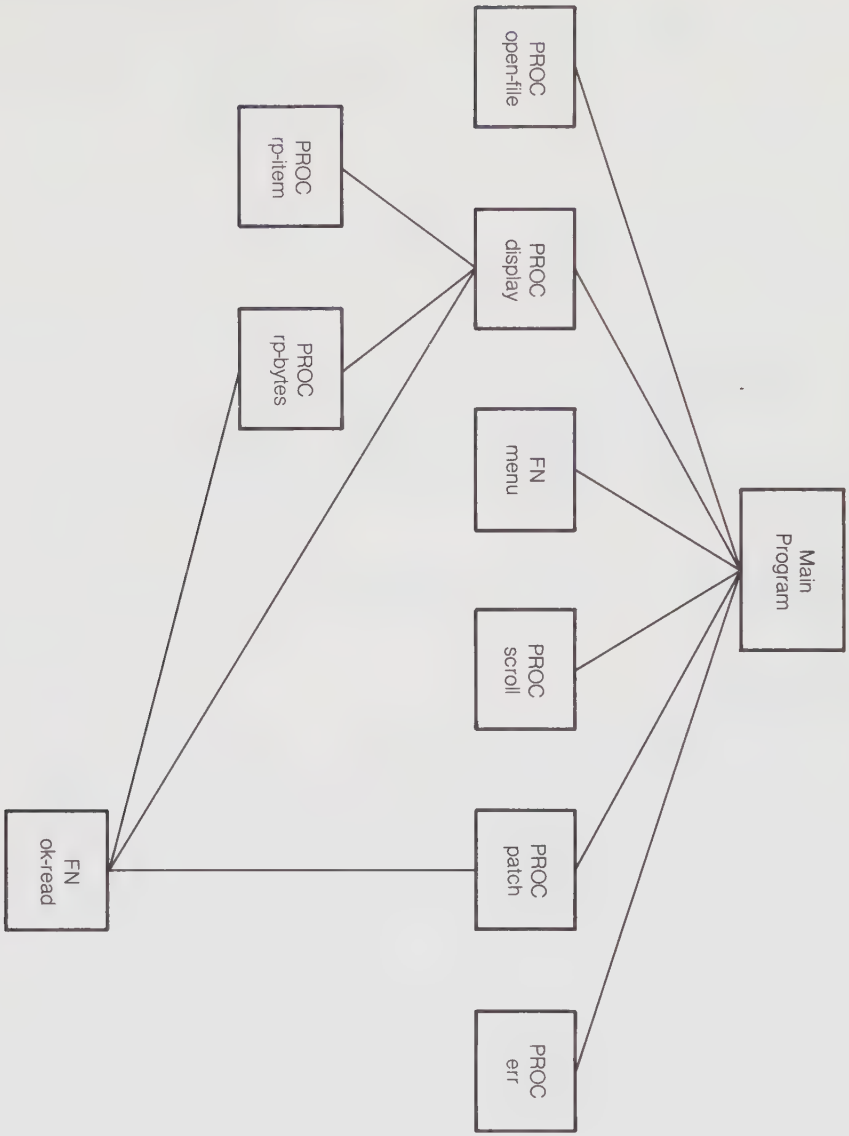
Once it is ascertained that a data item is of a specific type, then it can be read from the disk and displayed accordingly. Conversely, failure to discover one of the identification bytes at a nominated address enables the appropriate treatment of an individual byte both for display and patching.

With regard to patching, recognition of the type of data item allows control over what can be patched into what position since the constraint can easily be imposed that like must replace like.

Use of the display screen with this exercise poses an interesting problem, since the selection of a menu function really needs to be made on the basis of what is being presented by way of a file dump. The implication, therefore, is that the display and the menu must be managed so as to share the display screen in some way. The method used is as illustrated:



For the separate parts of the program - dump display, and menu - the facilities of the BBC Micro for redefinition of the current text area are utilised. In this way the two sections of the screen can be used as independent areas, each containing different types of information in their own right.


Main Program
PROC open-file
PROC rp-item
PROC display
PROC rp-bytes
FN menu
PROC scroll
FN ok-read
PROC patch
PROC err

PATCHER - FULL PROGRAM LISTING

```
 10 REM****************
 20 REM* CASE STUDY 3 *
 30 REM****************

 40 ON ERROR PROCerr: END
 50 @%=4: fu%=0: CLS
 60 PROCopen_file: sptr%=0
 70 REPEAT
 80   PROCdisplay(sptr%)
 90   func%=FNmenu
100   IF func%=0 PROCscroll
110   IF func%=1 PROCpatch
120   UNTIL func%=2
130 VDU26: CLS: CLOSE#fu%: PRINT'"Exit from PATCHER"'
140 END

150 DEFPROCscroll
160 IF PTR#fu%<EXT#fu% THEN sptr%=PTR#fu%
170 ENDPROC

180 DEFPROCpatch
190 CLS: REPEAT
200   INPUT"Which (Start) Address",add$:
       ptr%=EVAL("&"+add$)
210   UNTIL ptr%>=0 AND ptr%<=EXT#fu%
220 PTR#fu%=ptr%
230 IF NOT FNok_read THEN INPUT'"Input BYTE(Hex):" byt$:
     byte%=EVAL("&"+byt$)MOD256: BPUT#fu%,byte%: ENDPROC
240 IF type%=&40 INPUT'"Input an INTEGER value:" integer%:
     PRINT#fu%,integer%: ENDPROC
250 IF type%=&FF THEN INPUT'"Input a REAL value:" real:
     PRINT#fu%,real: ENDPROC
260 INPUT#fu%,string$: sl%=LENstring$: PTR#fu%=ptr%
270 INPUT'"Input a STRING:" string$
280 IF LENstring$>sl% THEN PRINT'sl%;" chars maximum at
     this address": GOTO270
290 PRINT#fu%,string$: ENDPROC

300 DEFPROCopen_file
310 REPEAT: INPUT'"Name of file",fnam$: UNTIL fnam$<>""
320 fu%=OPENUP(fnam$):   REM Use OPENIN with BASIC I
330 ENDPROC

340 DEFPROCdisplay(start%): LOCAL line%: line%=1
350 VDU28,0,16,39,0: CLS
360 PTR#fu%=start%
370 REPEAT
```

```
  IF FNok_read THEN PROCrp_item ELSE PROCrp_bytes
  UNTIL line%>=15 OR EOF#fu%
IF EOF#fu% THEN PRINT'STRING$(3,"*END OF FILE*")
 ELSE PRINT'STRING$(39,"|")
ENDPROC

DEFFNok_read
type%=BGET#fu%: PTR#fu%=PTR#fu%-1
= (type%=&00 OR type%=&40 OR type%=&FF)

DEFPROCrp_item
PRINT'~PTR#fu%;" ";
IF type%=&00 THEN INPUT#fu%,string$ ELSE
 IF type%=&40 INPUT#fu%,integer% ELSE INPUT#fu%,real
IF type%=&00 PRINT "s  ";string$; ELSE IF type%=&40
 THEN PRINT "i  ";integer%; ELSE PRINT "r  ";real;
IF type%=&00 THEN line%=line%+((LENstring$-32)DIV 40)+1
 ELSE line%=line%+1
ENDPROC

DEFPROCrp_bytes: LOCAL k%
k%=PTR#fu% MOD 8: PRINT'~PTR#fu%;" b    ";
REPEAT
  line%=line%+1
  REPEAT
    @%=2: PRINT~type%;" ";: @%=4: k%=k%+1
    type%=BGET#fu%
    UNTIL k%=8 OR FNok_read
    IF line%<15 AND NOT FNok_read THEN
     PRINT'~PTR#fu%;" b    ";: k%=0
  UNTIL line%=15 OR FNok_read
ENDPROC

DEFFNmenu: LOCAL fc%
VDU28,0,24,39,18: CLS
PRINT"0. NEXT SECTION OF FILE ";fnam$
PRINT"1. MAKE A CHANGE"
PRINT"2. QUIT"
REPEAT
  INPUT'"Which function (0-2)",fc%
  UNTIL fc%>-1 AND fc%<3: =fc%

DEFPROCerr
VDU26: CLS
PRINT'"Error ";ERR;" in line ";ERL':REPORT:PRINT
IF fu%<>0 CLOSE#fu%
ENDPROC
```

PATCHER - ANNOTATED PROGRAM LISTING

List of Global Variables used:

```
STRINGS
add$            Hexadecimal file address
byt$            Hexadecimal byte to be patched into the file
fnam$           Filename
string$         String to be patched into the file

INTEGERS
fu%             Channel number of file to be updated
func%           Menu function selected
integer%        Integer to be patched into the file
line%           Line counter for display
ptr%            Value for PTR# to access data item
sl%             Length of string$
sptr%           Start value for PTR# to display file section
type%           Type (Integer, Real, String) of data item found

REAL
real            Real value to be patched into the file
```

```
   10 REM****************
   20 REM* CASE STUDY 3 *
   30 REM****************
```

(After opening the required file for updating, the program repeatedly executes a loop which comprises just two elements: display a section of the file, and ask - through a menu - for the user to select a function to be performed. This loop terminates when the QUIT function is selected. Since numbers are to be output in the dump, the number output format is initialised to a four-digit display using @%=4.)

```
   40 ON ERROR PROCerr: END
   50 @%=4: fu%=0: CLS
   60 PROCopen_file: sptr%=0
   70 REPEAT
   80   PROCdisplay(sptr%)
   90   func%=FNmenu
  100   IF func%=0 PROCscroll
  110   IF func%=1 PROCpatch
  120   UNTIL func%=2
  130 VDU26: CLS: CLOSE#fu%: PRINT'"Exit from PATCHER"'
  140 END
```

(Procedure 'scroll' is quite trivial. The displayed section begins at the address given by variable sptr%, and continues until either the 15 lines of display area are full or the last section of the file is reached. The code to implement the 'next section' function must merely

set sptr% to wherever PTR# finished up, so long as it was not at the end of the file.)

```
  150 DEFPROCscroll
  160 IF PTR#fu%<EXT#fu% THEN sptr%=PTR#fu%
  170 ENDPROC
```

(Procedure 'patch' performs the CHANGE function if selected. Firstly the byte address of the item to be changed is input. Since this is taken to be a hexadecimal address, the EVAL function is employed to convert it to decimal. Then the byte at this address is inspected (using FNok_read) to determine whether it is an identifying byte for an integer, real or string value, or just any old byte. Depending upon the result of this examination, an appropriate prompt is displayed, and the user required to input the element that is to be patched into the file. When received, the new item is written to the file at the address previously specified. In the case of a string input, its length is checked against the maximum possible; if it is too long then it is not sent to the file but another string is requested.)

```
  180 DEFPROCpatch
  190 CLS: REPEAT
  200   INPUT"Which (Start) Address",add$:
        ptr%=EVAL("&"+add$)
  210   UNTIL ptr%>=0 AND ptr%<=EXT#fu%
  220 PTR#fu%=ptr%
  230 IF NOT FNok_read THEN INPUT'"Input BYTE(Hex):" byt$:
      byte%=EVAL("&"+byt$)MOD256: BPUT#fu%,byte%: ENDPROC
  240 IF type%=&40 INPUT'"Input an INTEGER value:" integer%:
      PRINT#fu%,integer%: ENDPROC
  250 IF type%=&FF THEN INPUT'"Input a REAL value:" real:
      PRINT#fu%,real: ENDPROC
  260 INPUT#fu%,string$: sl%=LENstring$: PTR#fu%=ptr%
  270 INPUT'"Input a STRING:" string$
  280 IF LENstring$>sl% THEN PRINT'sl%;" chars maximum at
      this address": GOTO270
  290 PRINT#fu%,string$: ENDPROC
```

(Procedure 'open_file' opens the specified file for updating. Remember that this will mean using either OPENIN or OPENUP depending on whether BASIC I or BASIC II is available.)

```
  300 DEFPROCopen_file
  310 REPEAT: INPUT'"Name of file",fnam$: UNTIL fnam$<>""
  320 fu%=OPENUP(fnam$):   REM Use OPENIN with BASIC I
  330 ENDPROC
```

(Procedure 'display' controls the screen output of a section of the file, starting from the point given by its parameter start%. After setting the display area to the first 17 lines of the screen, items are

read and displayed by using one of two other procedures - rp_item for integers, reals, and strings; rp_bytes for odd bytes. The final line of the output is either a row of '|' characters if more of the file still remains to be displayed, or an *END OF FILE* banner.)

```
340 DEFPROCdisplay(start%): LOCAL line%: line%=1
350 VDU28,0,16,39,0: CLS
360 PTR#fu%=start%
370 REPEAT
380   IF FNok_read THEN PROCrp_item ELSE PROCrp_bytes
390   UNTIL line%>=15 OR EOF#fu%
400 IF EOF#fu% THEN PRINT'STRING$(3,"*END OF FILE*")
     ELSE PRINT'STRING$(39,"|")
410 ENDPROC
```

(Function 'ok_read' produces a result TRUE if the next item to be input from the disk file is an identified type, i.e. integer, real or string; FALSE if it is a UFO - Unidentified File Object! This is achieved by the simple method of reading the byte at which PTR# is aimed, and setting the result on whether or not that value (type%) is 00, 40, or FF. Note that PTR# is repositioned so that the whole item can be read in subsequently.)

```
420 DEFFNok_read
430 type%=BGET#fu%: PTR#fu%=PTR#fu%-1
440 = (type%=&00 OR type%=&40 OR type%=&FF)
```

(Procedure 'rp_item' reads from disk and then displays the data item at which PTR# is pointing. The only tricky part is in keeping count of how many lines of the display area have been used. Integer and real values, since they only occupy one line, present no problem. For a string, which may spread across more than one line, a little extra effort is needed.)

```
450 DEFPROCrp_item
460 PRINT'~PTR#fu%;" ";
470 IF type%=&00 THEN INPUT#fu%,string$ ELSE
     IF type%=&40 INPUT#fu%,integer% ELSE INPUT#fu%,real
480 IF type%=&00 PRINT "s  ";string$; ELSE IF type%=&40
     THEN PRINT "i  ";integer%; ELSE PRINT "r  ";real;
490 IF type%=&00 THEN line%=line%+((LENstring$-32)DIV 40)+1
     ELSE line%=line%+1
500 ENDPROC
```

(Procedure 'rp_bytes' reads and then displays a series of individual bytes until an identified data item is met. The disk input part is straightforward - one byte is read at a time and, assuming that the byte is not one of either 00, 40 or FF, it is then displayed. The procedure organises this output so as to place a maximum of eight bytes onto a line, each being output as a 2-digit hexadecimal value. In order to establish some sort of compatibility with a *DUMP output, the first row

of bytes is reduced if necessary, so as to ensure that the next line begins with an address ending in 0 or 8.)

```
  510 DEFPROCrp_bytes: LOCAL k%
  520 k%=PTR#fu% MOD 8: PRINT'~PTR#fu%;" b    ";
  530 REPEAT
  540   line%=line%+1
  550   REPEAT
  560     @%=2: PRINT~type%;" ";: @%=4: k%=k%+1
  570     type%=BGET#fu%
  580     UNTIL k%=8 OR FNok_read
  590     IF line%<15 AND NOT FNok_read THEN
           PRINT'~PTR#fu%;" b    ";: k%=0
  600   UNTIL line%=15 OR FNok_read
  610 ENDPROC
```

(Function 'menu' follows the same pattern as its counterparts in the earlier Case Studies. The only difference in this case is the use of the VDU28 command to set up a small window for the menu below the section of the disk file currently being displayed.)

```
  620 DEFFNmenu: LOCAL fc%
  630 VDU28,0,24,39,18: CLS
  640 PRINT"0. NEXT SECTION OF FILE ";fnam$
  650 PRINT"1. MAKE A CHANGE"
  660 PRINT"2. QUIT"
  670 REPEAT
  680   INPUT'"Which function (0-2)",fc%
  690   UNTIL fc%>-1 AND fc%<3: =fc%
```

(Procedure 'err' is entered should any unforeseen errors occur during operation. It re-establishes the whole screen as the display area, and closes the file being patched.)

```
  700 DEFPROCerr
  710 VDU26: CLS
  720 PRINT'"Error ";ERR;" in line ";ERL':REPORT:PRINT
  730 IF fu%<>0 CLOSE#fu%
  740 ENDPROC
```

PATCHER - ROOMS FOR IMPROVEMENT

This Case Study program is a quite unsophisticated affair which seeks to provide a useful facility without frills. There are a few suggestions below as to ways in which it could be embellished; the reader can no doubt think of many others in addition. It is a moot point, however, as to how far one should go to produce a super-powerful program for an activity - disk file patching - which really should not become a regular activity!

Individual bytes are only output in hexadecimal. These could be displayed in character form also, as they are with the *DUMP display. Alternatively, hexadecimal/character output could be a choice made by the user when the program is entered.

+++++

Once an early section of the file has been superseded, there is no clean way of getting back to it, only of moving further on through the file. It is possible to go backwards simply by selecting an earlier address but this is rather hit-and-miss since that location is not currently being displayed. The program could be altered so that the 'Next section of file' option moved either forwards or backwards depending upon a user input. This will not be a trivial change, however - the need to limit the output of any section to 15 lines of display means that some analysis would need to be done in order to determine the address at which the previous section should begin.

+++++

The patching facility is rigidly constrained - an integer can only be replaced by an integer, a real by a real, a string by another string of the same or shorter length. It might be worth, since we all have the utmost faith in our ability not to make mistakes, changing the program so as to relax these restrictions and allow items to be written at addresses already occupied by data of a different type. One simple approach would be to issue a warning, rather than to prevent such a patch from being made, with the operation being carried out if the user confirmed that this was what was intended.

CASE STUDY 4
LOADER – A technique for implementing a BASIC Procedure Library

LOADER - FUNCTION

Chapter 8 describes how a *SPOOL file containing lines of BASIC can be appended to a portion of program that is already in RAM, the general technique being based on the idea of treating the former as an EXEC file. Now there is no reason why this technique could not be used a number of times, adding the contents of a different file on each occasion. The basis exists, therefore, for the user to be able to create a small collection of 'library' files containing chunks of BASIC that are used regularly.

This Case Study provides an automatic method for implementing such a library. Naturally, these library files could contain any program lines at all. However, with BBC BASIC offering the capability for procedures handling, the most obvious approach is to construct a Procedure Library. In this way, a 'main program' can be written which contains calls on some of the procedures held in the library files, this main program being written to disk using SAVE as usual. Subsequently it can be brought into RAM using LOAD, the required procedures being added by taking selected files from the Procedure Library with repeated EXEC calls. The example dialogue demonstrates this approach.

Program LOADER serves to automate this process. It requires the user to input

(a) name of file containing 'main program',

(b) names of required files from the Procedure Library.

It then proceeds to create a !BOOT file which, when the user presses SHIFT+BREAK will initiate the sequence of loading the main program and its selected Procedure Library files.

LOADER - SAMPLE DIALOGUE

(As far as this Case Study program is concerned, a certain amount of activity will have taken place beforehand. In particular, the BASIC library files will have been created using *SPOOL in the manner that has been described in Chapter 8. Also another file will have been created using SAVE, and will hold the opening part of the program or, as it is usually termed, the 'main program'. For this example it is assumed that a file called MAIN, and three BASIC library files P1, P2 and P3, have been set up as follows:)

```
>10 PROC              (Creating file "MAIN")
>20 PROCtwo
>30 PROCthree
>40 END
>SAVE "MAIN"

>NEW                  (Creating file "P1")
>30000 DEFPROCone
>30010 PRINT"HERE I AM"
>30020 ENDPROC
>*SPOOL P1
>LIST
30000 DEFPROCone
30010 PRINT"HERE I AM"
30020 ENDPROC
>*SPOOL

>NEW                  (Creating file "P2")
>30000 DEFPROCtwo
>30010 PRINT"HERE I AM AGAIN"
>30020 ENDPROC
>*SPOOL P2
>LIST
30000 DEFPROCtwo
30010 PRINT"HERE I AM AGAIN"
30020 ENDPROC
>*SPOOL

>NEW                    (Creating file "P3")
>30000 DEFPROCthree
>30010 PRINT"HOW DO YOU DO"
>30020 ENDPROC
>*SPOOL P3
>LIST
30000 DEFPROCthree
30010 PRINT"HOW DO YOU DO"
30020 ENDPROC
>*SPOOL
```

(There now follows an example of how the Case Study program is used to automatically load into RAM the main program MAIN, together with the three BASIC Library files P1, P2 and P3.)

```
>CHAIN "LOADER"
```

(The program asks firstly for the name of the file containing the opening section of the program. As with MAIN, this file must always have been created by using the SAVE command.)

```
Name of Main Program?MAIN
```

(Then a series of BASIC library files are specified. Each name input is taken to be a reference to a file of BASIC lines created by using the *SPOOL command. Between the input of each filename an amount of disk activity will take place as the links between the files are set up (described in detail under METHOD USED). This sequence of names is terminated by pressing RETURN in answer to the prompt.)

```
Library file (RETURN to finish)?P1
Library file (RETURN to finish)?P2
Library file (RETURN to finish)?P3
Library file (RETURN to finish)?   <----(RETURN i.e. no more files.)
```

(The program sets up a !BOOT file and then invites the user to do something with it.)

```
Press SHIFT+BREAK to Load Program
```

(The user obediently presses SHIFT + BREAK. The output which follows is generated as each EXEC file is called in turn, starting with the !BOOT file which the case study program created for itself.)

```
>LOAD "MAIN"                    (EXEC file !BOOT entered)
>*E. P1

>REN.                           (EXEC file P1 entered)
>30000 DEFPROCone
>30010 PRINT"HERE I AM"
>30020 ENDPROC
>*E. P2

>REN.                           (EXEC file P2 entered)
>30000 DEFPROCtwo
>30010 PRINT"HERE I AM AGAIN"
>30020 ENDPROC
>*E. P3
```

```
>REN.                          (EXEC file P3 entered)
>30000 DEFPROCthree
>30010 PRINT"HOW DO YOU DO"
>30020 ENDPROC
>REN.
```

(At the end of the chain, the whole program - comprising main program and any specified BASIC library files - will be in RAM. From there it can be RUN, SAVEd and generally treated in the usual sort of way.) e.g.

```
>LIST
   10 PROC
   20 PROCtwo
   30 PROCthree
   40 END
   50 DEFPROCone
   60 PRINT"HERE I AM"
   70 ENDPROC
   80 DEFPROCtwo
   90 PRINT"HERE I AM AGAIN"
  100 ENDPROC
  110 DEFPROCthree
  120 PRINT"HOW DO YOU DO"
  130 ENDPROC

>RUN

HERE I AM
HERE I AM AGAIN
HOW DO YOU DO
```

LOADER - METHOD USED

It would have been very nice if LOADER could have simply created a !BOOT file which contained (using the files mentioned in SAMPLE DIALOGUE)

```
*EXEC MAIN
*EXEC P1
*EXEC P2
*EXEC P3
```

Unfortunately, this is not a viable proposition. The reason is that, when an EXEC file is being obeyed, the appearance of an *EXEC statement causes the new EXEC file to be opened, and the current file to be closed. Thus with the above example a call of

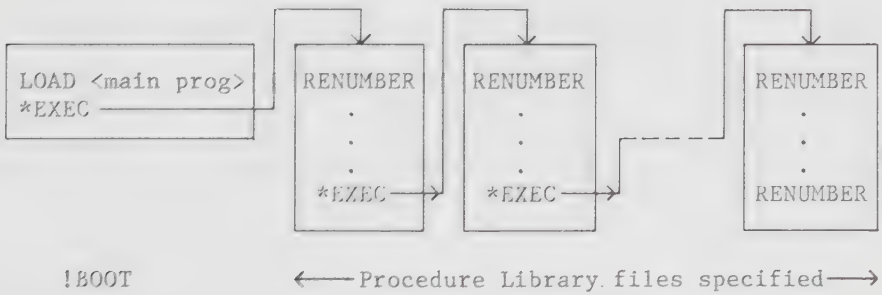
```
*EXEC !BOOT  (which is what SHIFT+BREAK effects)
```

would start quite happily. When

```
*EXEC MAIN
```

is encountered, however, file !BOOT is closed - which means that the remaining *EXEC calls in that file do not get obeyed. This is a little disappointing; nevertheless, all is not lost. It is possible, as the above explanation indicates, for one EXEC file to terminate with an *EXEC command to invoke a second EXEC file. Since there is no reason why this facet cannot be used a number of times, it is thus possible for a 'chain' of EXEC files to be set up with one file linking to another, and so on.

For our Procedure Library application then, a way forward is for each library file to end with an *EXEC command which invokes the next library file. Diagrammatically, the organisation may be represented in the following way:



The !BOOT file loads the main program in the normal way and then uses an

*EXEC to commence the addition of the first quoted procedure file. This file begins with a RENUMBER command, and ends with an *EXEC to link to the next required library file, the process continuing until the final library file is input. This final file terminates with a RENUMBER which leaves the final loaded program in a tidy form.

Since the aim of this Case Study is to automate the whole process of loading these different Procedure Library files, not only does program LOADER have to set up the !BOOT, but also to 'plant' into each of the library files the opening and closing RENUMBER and *EXEC commands. In general terms LOADER operates as follows:

1. It creates, after obtaining from the user the name of the file containing the 'main program', a file called !BOOT which contains two lines:

```
LOAD "<Main program file>"
*********
```

   The second of these two lines is simply there to occupy a bit of space; it is overwritten a little later.

2. Then, for every Procedure Library file (Pn, say) specified:

   (a) write '*EXEC Pn' to become the final line of the currently open file (in the first instance this replaces the ********* line of the !BOOT file);
   (b) close this file;
   (c) open file "Pn";
   (d) write RENUMBER to become the first line of this file.

3. For the final library file only:

   (a) write RENUMBER to become the final line of this file;
   (b) close the file.

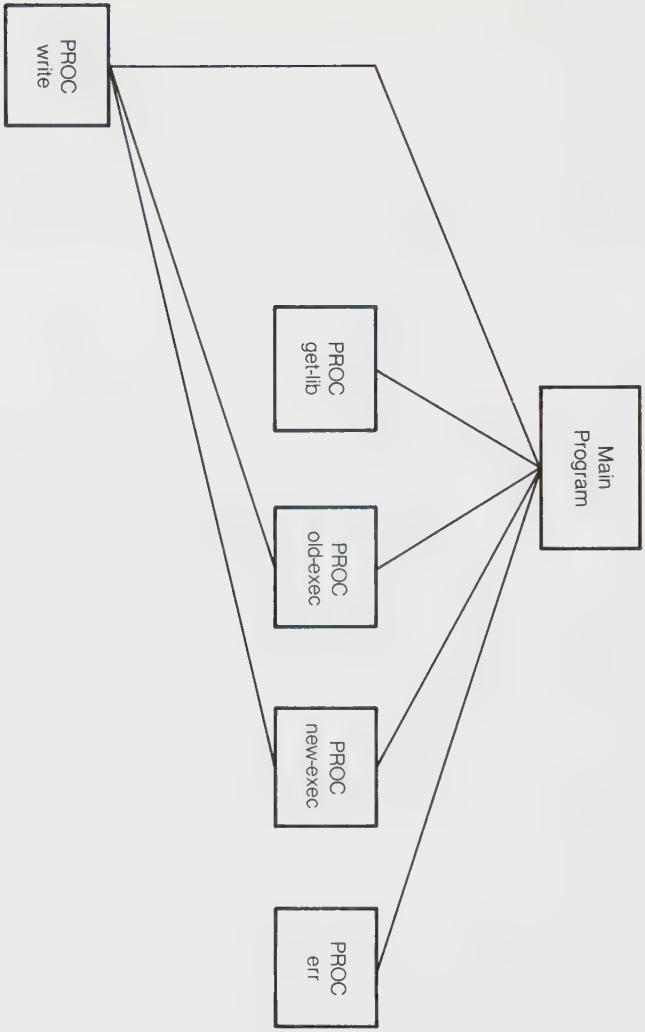
Now you may well be thinking, doesn't all this planting of RENUMBERs and *EXECs mean that the Procedure Library files will become larger? And what happens if the files cannot be extended? Well done - these are very good questions and in appropriate circumstances the points you have raised could certainly play havoc with the whole process. The simple way out here is to impose one or two simple restrictions on how a library file is to be created with *SPOOL. Taking our library file P1 from the Sample Dialogue, we can see that after its creation its contents are as follows:

```
>*DUMP P1

0000  3E 4C 49 53 54 0A 0D 33  >LIST..3
0008  30 30 30 30 20 44 45 46  0000 DEF
0010  50 52 4F 43 6F 6E 65 0A  PROCone.
0018  0D 33 30 30 31 30 20 50  .30010 P
0020  52 49 4E 54 22 48 45 52  RINT"HER
0028  45 20 49 20 41 4D 22 0A  E I AM".
0030  0D 33 30 30 32 30 20 45  .30020 E
0038  4E 44 50 52 4F 43 0A 0D  NDPROC..
0040  3E 2A 53 50 4F 4F 4C 0A  >*SPOOL.
0048  0D ** ** ** ** ** ** **  ........
```

This indicates that there are five bytes (0000 to 0004) at the start of the file, containing >LIST, which could be overwritten. Similarly the seven bytes (0040 to 0046) containing >*SPOOL at the end of the file can also be used. Now these two areas will always be available if, when creating the spooled file, the commands LIST and *SPOOL are used and not their abbreviations (L. and *SP.). Given that this has occurred, LOADER is able to replace the characters >LIST in this file by the characters REN., i.e. the abbreviated form of the command RENUMBER. Likewise, if we restrict the name of any Procedure Library file to 4 characters, then LOADER can replace >*SPOOL by *E.xxxx, where *E. is the short form of *EXEC, and xxxx is the file name. In this way the links from one library file to another, and the necessary line renumbering, can be effected without the library file needing to be extended in any way.

File P1, after being doctored by LOADER in the way described, would therefore look as follows:

```
>*DUMP P1

0000  52 45 4E 2E 20 0A 0D 33  REN. ..3
0008  30 30 30 30 20 44 45 46  0000 DEF
0010  50 52 4F 43 6F 6E 65 0A  PROCone.
0018  0D 33 30 30 31 30 20 50  .30010 P
0020  52 49 4E 54 22 48 45 52  RINT"HER
0028  45 20 49 20 41 4D 22 0A  E I AM".
0030  0D 33 30 30 32 30 20 45  .30020 E
0038  4E 44 50 52 4F 43 0A 0D  NDPROC..
0040  2A 45 2E 50 32 0A 0D 0A  *E.P2...
0048  0D ** ** ** ** ** ** **  ........
```


PROC
write
PROC
get-lib
PROC
old-exec
PROC
new-exec
PROC
err
Main
Program

LOADER - FULL LISTING

```
 10 REM****************
 20 REM* CASE STUDY 4 *
 30 REM****************

 40 ON ERROR PROCerr: END
 50 CLS: fu%=0: INPUT"Name of Main Program",main$
 60 fu%=OPENOUT(":0.$.!BOOT")
 70 PROCwrite("LOAD "+CHR$34+main$+CHR$34)
 80 PROCwrite("*******")
 90 REPEAT
100   PROCget_lib
110   PROCold_exec
120   PROCnew_exec
130   UNTIL lib$=""
140 PTR#fu%=EXT#fu%-9: PROCwrite("REN. ")
150 CLOSE#fu%
160 PRINT "Press SHIFT+BREAK to Load the Program"'
170 END

180 DEFPROCwrite(s$): LOCAL byte%
190 REPEAT
200   byte%=ASC(s$): BPUT#fu%,byte%
210   s$=MID$(s$,2)
220   UNTIL s$=""
230 BPUT#fu%,&0A: BPUT#fu%,&0D
240 ENDPROC

250 DEFPROCget_lib
260 INPUT"Library file (RETURN to finish)",lib$
270 ENDPROC

280 DEFPROCold_exec
290 IF lib$="" ENDPROC
300 PTR#fu%=EXT#fu%-9
310 PROCwrite("*E."+lib$)
320 ENDPROC

330 DEFPROCnew_exec
340 IF lib$="" ENDPROC
350 CLOSE#fu%
360 fu%=OPENUP(lib$): REM Use OPENIN with BASIC I
370 PROCwrite("REN. ")
380 ENDPROC

390 DEFPROCerr
400 PRINT '"Error ";ERR;" in line ";ERL': REPORT: PRINT
410 IF fu%<>0 CLOSE#fu%
420 ENDPROC
```

LOADER - ANNOTATED PROGRAM LISTING

List of Global Variables used:

STRINGS
lib$ Name of library file
main$ Name of main program file

INTEGER
fu% Channel number of !BOOT file

```
   10 REM****************
   20 REM* CASE STUDY 4 *
   30 REM****************
   40 ON ERROR PROCerr: END
```

(After obtaining the name of the file holding the 'main program', the EXEC file !BOOT is opened. Two lines are written to it: LOAD "<main program file>", and a dummy string of seven asterisks (which effectively reserves nine bytes on disk. This dummy string will subsequently be overwritten by a line of the form, *E.<library file name>.)

```
   50 CLS: fu%=0: INPUT"Name of Main Program",main$
   60 fu%=OPENOUT(":0.$.!BOOT")
   70 PROCwrite("LOAD "+CHR$34+main$+CHR$34)
   80 PROCwrite("*******")
```

(Following this, LOADER cycles round a loop which asks for the name of a library file, patches an *EXEC call to this file into the file previously opened, and then patches a RENUMBER command into the start of this file. The cycle is terminated by the input of a null file name.)

```
   90 REPEAT
  100   PROCget_lib
  110   PROCold_exec
  120   PROCnew_exec
  130   UNTIL lib$=""
```

(The last library file has a RENUMBER command patched to its end, and LOADER finishes with a reminder to the user of how to invoke the !BOOT file to commence the loading of the complete program.)

```
  140 PTR#fu%=EXT#fu%-9: PROCwrite("REN. ")
  150 CLOSE#fu%
  160 PRINT "Press SHIFT+BREAK to Load the Program"'
  170 END
```

(Procedure 'write' takes the string, s$, supplied as its parameter, and sends this string as a series of ASCII bytes to the currently open file.

This character stream is terminated by the output of 'line feed/return' - the ASCII codes &0A AND &0D. This output is thus in the form required for future input as an EXEC file.)

```
  180 DEFPROCwrite(s$): LOCAL byte%
  190 REPEAT
  200   byte%=ASC(s$): BPUT#fu%,byte%
  210   s$=MID$(s$,2)
  220   UNTIL s$=""
  230 BPUT#fu%,&0A: BPUT#fu%,&0D
  240 ENDPROC
```

(Procedure 'get_lib' asks for the name of a Procedure Library file, putting the input name into lib$.)

```
  250 DEFPROCget_lib
  260 INPUT"Library file (RETURN to finish)",lib$
  270 ENDPROC
```

(Procedure 'old_exec' locates the file pointer to the start of the character string >*SPOOL in the currently opened file, and then writes the stream *E.<lib$> at that position.)

```
  280 DEFPROCold_exec
  290 IF lib$="" ENDPROC
  300 PTR#fu%=EXT#fu%-9
  310 PROCwrite("*E."+lib$)
  320 ENDPROC
```

(Procedure 'new_exec' closes the current file and opens the newly specified library file. This new file then has the character stream "REN. " sent to byte 00000 onwards.)

```
  330 DEFPROCnew_exec
  340 IF lib$="" ENDPROC
  350 CLOSE#fu%
  360 fu%=OPENUP(lib$): REM Use OPENIN with BASIC I
  370 PROCwrite("REN. ")
  380 ENDPROC
```

(Procedure 'err' has been met before - it is obviously a prime candidate for being converted into a Procedure Library file!)

```
  390 DEFPROCerr
  400 PRINT '"Error ";ERR;" in line ";ERL': REPORT: PRINT
  410 IF fu%<>0 CLOSE#fu%
  420 ENDPROC
```

LOADER - ROOMS FOR IMPROVEMENT

Some suggestions for changes to the version of LOADER that has been described:

The !BOOT file that is created by LOADER could be invoked directly from that program, rather than asking the user to press SHIFT+BREAK. The change is trivial: line 160 is replaced by

```
160 *EXEC :0.$.!BOOT
```

which causes the !BOOT file to be accessed once LOADER has set up all the links between the files. In this instance line 170 (END) is redundant because it is never reached.

+++++

The restriction of four characters for a Procedure Library file name is to ensure that the *E.xxxx does not take up more room on the disk than the >*SPOOL that it replaces. This restriction could be removed if the program checked how much space was in fact available for use. Since the DFS always allocates whole numbers of sectors on disk to any file, program or data, there may well be a few bytes that can be used in the last sector. The calculation

```
free%=EXT#fu% MOD 256: IF free%<>0 THEN free%=256-free%
```

will indicate how much.

+++++

The scheme implemented by LOADER could be extended by one more step. A 'closedown' file, call it FINI, could also be created, with the final library file having an *EXEC statement to this file at its end rather than a RENUMBER. This closedown file might then comprise,

```
RENUMBER
LIST
*DELETE :0.$.BOOT
```

This still leaves the problem of getting rid of the file FINI. Solutions, on a postcard please, to the author!

CASE STUDY 5
A database package
MAKEDB – A database creation program
USEDB – A program for accessing the database

INTRODUCTION

This is a major and challenging Case Study. It attempts, through the implementation of a database package, to draw together many of the techniques that have been introduced for programming with random access files. It incorporates, for instance, the use of both fixed- and variable-length records; the data is structured by means of an indexing system; and a form of garbage collecton is implemented.

The database package is designed to be one that is of general applicability in any situation that requires the storage and retrieval of sets of related data. The basic unit of data is a variable-length record which may comprise a number of 'fields' and 'keywords'; the exact form of this record is defined by the user. In addition it is possible to effect subdivisions of the database content by defining any record as being a member of a particular category. In this way retrieval of information from the database is made possible under a variety of search criteria, formed from a combination of selected category and keywords.

The database package comprises two programs reflecting the two distinct aspects of such systems - definition, and use. The first, program MAKEDB, allows the user to define the categories of record that are to be held in the database, and the form that each data record will take. The database file is then created. Thereafter operations on the database - additions, deletions, searches - are carried out by using the second program: USEDB. It is this program which could be said to 'maintain' the database.

Whilst stressing that the package may be used in a variety of

ways it is nevertheless most convenient to describe its operation by means of a specific application. For this reason the documentation for the Case Study will employ as an example the use of the package to set up a database containing information about articles published in a popular journal which concentrates on the BBC Micro: 'The Micro User'. For obvious reasons only a small amount of the database content will be revealed in what follows, but a fully constructed 'Micro User' database is included on the disk which has been produced to accompany this book.

MAKEDB - FUNCTION

It is the role of the MAKEDB program to create an empty database. This might sound as though it is a contradiction in terms, but an empty database has in fact got quite a lot in it! As the final example of Chapter 9 shows, any file which is based on an indexed organisation has to have an index stored as part of the file. One of the functions of MAKEDB then is to create this index.

Our example database is going to hold records which describe articles that have been published in the 'Micro User' journal. Since the object of an index is to subdivide the database content so as to speed up the searching process, it seems sensible for our example to define an index which is based on major subject CATEGORIES, making them as distinct as possible. The chosen categories are Devices, Education, Graphics, Languages, Operating Systems, Programming, Reviews, Software, Sound, and an odds-and-sundries Miscellany.

In addition every entry in the database will comprise a number of fields - that is, the component parts which go to make up a complete record. When the database is being used (to add a new record for instance) prompts need to be output to the user to indicate what sort of information he has to supply. The nature of the field content, numeric or string, is also useful to know since it enables some simple validity checks to be made when a field is input. This information, termed FIELD NAME and TYPE, is also defined with MAKEDB and stored in the database file. For our example database, which will hold records about published articles, the headings and types are Author (Type = String), Title (String), Volume (Numeric), Number (Numeric), Page (String), Date (String).

MAKEDB - SAMPLE DIALOGUE

```
>CHAIN "MAKEDB"

°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
       DATABASE CREATION PROGRAM
       *************************
```

(The program assumes that a file of suitable size has been created on disk already, probably with the *SAVE command. Our example file was set to a size of 5000(Hex) bytes using the command: *SAVE MUSE 0 +5000.)

```
Name of Database File?MUSE
```

(MAKEDB now asks for the different categories of record which the database is to hold. As a general rule these categories should be natural subdivisions of the different types of record, since the idea of this scheme is to speed up database accesses. For certain applications, however, it may be that only one category is defined.)

```
Input CATEGORIES (Max. 12 Chars)
RETURN to finish

Category 1?DEVICES
Category 2?EDUCATION
Category 3?GRAPHICS
Category 4?LANGUAGES
Category 5?OP.SYSTEMS
Category 6?PROGRAMMING
Category 7?REVIEWS
Category 8?SOFTWARE
Category 9?SOUND
Category 10?MISCELLANY
Category 11?   <--------(RETURN to finish)
```

(The field names are now input, together with the type of data which the field is to represent. Remember that the field names are to be used as prompts when the database is used. The names are held on disk as variable-length records and so there is no reason for undue brevity!)

```
Input FIELD NAMES and TYPES
RETURN to finish

Field Name?AUTHOR
Type: S=String, N=Numeric?S
Field Name?TITLE
Type: S=String, N=Numeric?S
Field Name?VOLUME
Type: S=String, N=Numeric?N
```

```
Field Name?NUMBER
Type: S=String, N=Numeric?N
Field Name?PAGE(S)
Type: S=String, N=Numeric?S
Field Name?DATE
Type: S=String, N=Numeric?S
Field Name?     <--------------(RETURN to finish)

Database MUSE Created
Exit from MAKEDB
>
```

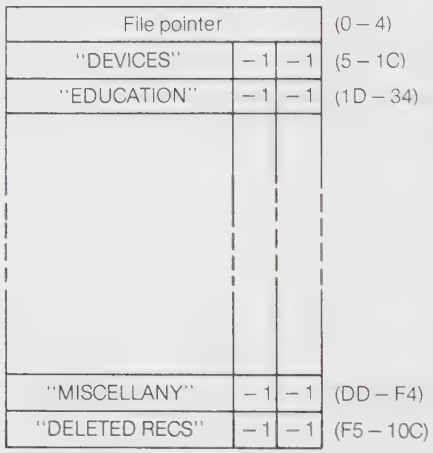
(The 'empty' database is now established with the above information held in its opening sectors. See METHOD USED for the exact format of these sectors.)

MAKEDB - METHOD USED

Two sections at the beginning of the database file are set up by MAKEDB. The first, which starts in the first sector of the file, contains the category names which the user has supplied. They are used as part of the database index, with each name being associated with two pointers, 'chain start' and 'chain end'. These are addresses of the first and last entries in the database for their category and, as explained in Chapter 9, are initially set to negative values. The category names are held as fixed-length strings of 12 characters (14 bytes), the pointers as integer values (of 5 bytes). In addition to the user-supplied category names, an additional entry is appended to this index. This entry is DELETED_RECS, and is the index entry which is used by the database package to mark the start and end of the chain of 'free space' left behind by records which have been deleted (see Section 9.6).

The content of part of this index is shown below, both in diagrammatic and *DUMP form. Bytes 0-4 are set by MAKEDB as a pointer to the start of the database entries proper; in our example this address is &300.

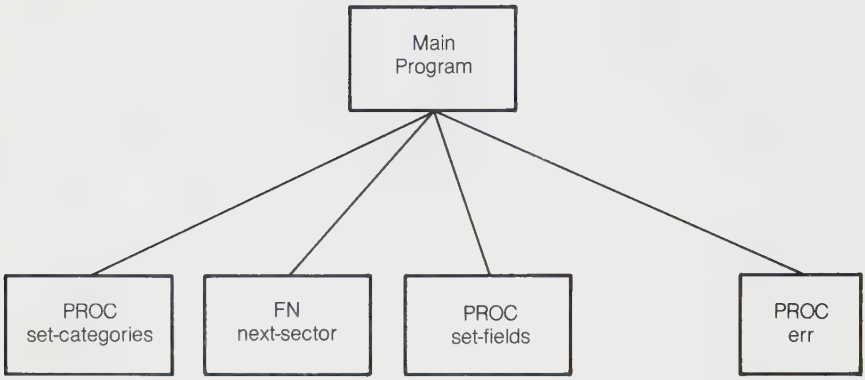
```
>*DUMP MUSE
0000 40 00 00 03 00 00 07 53 @.......S
0008 45 43 49 56 45 44 00 00 ECIVED..
0010 00 00 00 40 FF FF FF FF ...@....
0018 40 FF FF FF FF 00 09 4E @.......N
0020 4F 49 54 41 43 55 44 45 OITACUDE
0028 00 00 00 40 FF FF FF FF ...@....
0030 40 FF FF FF FF -- -- -- @....---
                    |
                    |
00D8 -- -- -- -- -- 00 0A 59 -------Y
00E0 4E 41 4C 4C 45 43 53 49 NALLECSI
00E8 4D 00 00 40 FF FF FF FF M..@....
00F0 40 FF FF FF FF 00 0C 53 @.......S
00F8 43 45 52 5F 44 45 54 45 CER_DETE
0100 4C 45 44 40 FF FF FF FF LED@....
0108 40 FF FF FF FF -- -- -- @....---
```



The second part of the file established by MAKEDB holds the field names and types. These entries comprise a single byte character, 'S' or 'N', depending on whether the field is defined as 'String' or 'Numeric' together with the variable-length field name. Because there is no limit to the number of field names that can be defined this list ends with a single byte terminating character, '|'. Again the content of this part of the file is given in both diagram and *DUMP forms.

```
0200 53 00 06 52 4F 48 54 55 S..ROHTU
0208 41 53 00 05 45 4C 54 49 AS..ELTI
0210 54 4E 00 07 45 4D 55 4C TN..EMUL
0218 4F 56 4E 00 07 52 45 42 OVN..REB
0220 4D 55 4E 53 00 07 29 53 MUNS..(S
0228 28 45 47 41 50 53 00 04 )EGAPS..
0230 45 54 41 44 7C -- -- -- ETAD|---
```

S	"AUTHOR"	(200 – 208)
S	"TITLE"	(209 – 210)
N	"VOLUME"	(211 – 219)
N	"NUMBER"	(21A – 222)
S	"PAGE(S)"	(223 – 22C)
S	"DATE"	(22D – 233)
I		(234)


Main
Program
PROC
set-categories
FN
next-sector
PROC
set-fields
PROC
err

MAKEDB - FULL LISTING

```
 10 REM*****************
 20 REM* CASE STUDY 5A *
 30 REM*****************

 40 ON ERROR PROCerr: END
 50 CLS: PRINT'TAB(7);"DATABASE CREATION PROGRAM"'TAB(7);
    STRING$(25,"*")'': fu%=0
 60 INPUT"Name of Database File",fnam$
 70 fu%=OPENUP(fnam$): REM Use OPENIN with BASIC I
 80 PTR#fu%=PTR#fu%+5
 90 PROCset_categories
100 PTR#fu%=FNnext_sector
110 PROCset_fields
120 nx%=FNnext_sector: PTR#fu%=0: PRINT#fu%,nx%
130 CLOSE#fu%: PRINT'"Database ";fnam$;" Created"'"Exit from MAKEDB"
140 END

150 DEFPROCset_categories: LOCAL cat$,cn%: cn%=0: @%=3
160 PRINT'"Input CATEGORIES (Max. 12 Chars)"'"RETURN to finish"'
170 REPEAT: cn%=cn%+1
180   PRINT"Category ";cn%;: INPUT cat$
190   ptr%=PTR#fu%
200   IF cat$<>"" THEN PRINT#fu%,LEFT$(cat$,12):
        PTR#fu%=ptr%+14: PRINT#fu%,-1,-1
210   UNTIL cat$=""
220 PRINT#fu%,"DELETED_RECS",-1,-1
230 ENDPROC

240 DEFPROCset_fields: LOCAL field$,type$
250 PRINT'"Input FIELD NAMES and TYPES"'"RETURN to finish"'
260 REPEAT
270   INPUT"Field Name",field$:
       IF field$="" BPUT#fu%,ASC("|"): GOTO 320
280   REPEAT
290     INPUT"Type: S=String, N=Numeric",type$
300     UNTIL type$="S" OR type$="N"
310   BPUT#fu%,ASCtype$: PRINT#fu%,field$
320   UNTIL field$=""
330 ENDPROC

340 DEFFNnext_sector
350 =(PTR#fu%DIV256+1)*256

360 DEFPROCerr
370 PRINT'"Error ";ERR;" in Line ";ERL: REPORT: PRINT
380 IF fu%<>0 CLOSE#fu%
390 ENDPROC
```

MAKEDB - ANNOTATED PROGRAM LISTING

List of Global Variables used:

STRINGS
fnam$ Filename

INTEGERS
fu% Channel number of database file
nx% Address of 'next free'

```
   10 REM*****************
   20 REM* CASE STUDY 5A *
   30 REM*****************
```

(The program, having opened the nominated database file, leaves five bytes for 'next free', and proceeds to set up the index of category names. Following this, at the start of the next sector, it writes field names and their types.)

```
   40 ON ERROR PROCerr: END
   50 CLS: PRINT'TAB(7);"DATABASE CREATION PROGRAM"'TAB(7);
       STRING$(25,"*")'': fu%=0
   60 INPUT"Name of Database File",fnam$
   70 fu%=OPENUP(fnam$): REM Use OPENIN with BASIC I
   80 PTR#fu%=PTR#fu%+5
   90 PROCset_categories
  100 PTR#fu%=FNnext_sector
  110 PROCset_fields
```

(The address of the start of the first full sector following the two tables that have been created will be that at which the database entries proper begin. This address, 'next free', is stored in bytes 0-4 of the file.)

```
  120 nx%=FNnext_sector: PTR#fu%=0: PRINT#fu%,nx%
  130 CLOSE#fu%: PRINT'"Database ";fnam$;" Created"'"Exit from MAKEDB"
  140 END
```

(Procedure 'set_categories' cycles round a loop which, after asking the user for a category name, writes this name, together with two negative 'pointers' to the file. The loop terminates on input of a null category upon which a final DELETED_RECS entry is set up for the 'free space' chain.)

```
  150 DEFPROCset_categories: LOCAL cat$,cn%: cn%=0: @%=3
  160 PRINT'"Input CATEGORIES (Max. 12 Chars)"'"RETURN to finish"'
  170 REPEAT: cn%=cn%+1
  180   PRINT"Category ";cn%;: INPUT cat$
  190   ptr%=PTR#fu%
```

```
  200   IF cat$<>"" THEN PRINT#fu%,LEFT$(cat$,12):
          PTR#fu%=ptr%+14: PRINT#fu%,-1,-1
  210   UNTIL cat$=""
  220 PRINT#fu%,"DELETED_RECS",-1,-1
  230 ENDPROC
```

(Procedure 'set_fields' also cycles round a loop asking each time for a field name and a type. If the field name is not null then both items are written to the database file, type% as a single byte ASCII character ('S' or 'N'). On input of a null field name, the single terminating byte, ASCII '|', is output.)

```
  240 DEFPROCset_fields: LOCAL field$,type$
  250 PRINT'"Input FIELD NAMES and TYPES"'"RETURN to finish"'
  260 REPEAT
  270   INPUT"Field Name",field$:
         IF field$="" BPUT#fu%,ASC("|"): GOTO 320
  280   REPEAT
  290     INPUT"Type: S=String, N=Numeric",type$
  300     UNTIL type$="S" OR type$="N"
  310   BPUT#fu%,ASCtype$: PRINT#fu%,field$
  320   UNTIL field$=""
  330 ENDPROC
```

(Function 'next_sector' calculates the start address of the next full sector from the current position of the file pointer PTR#. Procedure 'err' fulfils its usual role.)

```
  340 DEFFNnext_sector
  350 =(PTR#fu%DIV256+1)*256
  360 DEFPROCerr
  370 PRINT'"Error ";ERR;" in Line ";ERL: REPORT: PRINT
  380 IF fu%<>0 CLOSE#fu%
  390 ENDPROC
```

USEDB - FUNCTION

Program USEDB is a database management program which provides, in brief, the following facilities:

Add New Entry

New records may be added to the database. The program prompts the user for each field in turn by using the field names defined when the database was created with MAKEDB. The fields are of variable length. In addition up to three 'keywords' can be supplied by the user for inclusion with the entry in the database file.

Alter Existing Entry

Existing records may be modified, either in terms of the content of any field, or of their related keywords. The facility exists for modified records to occupy more disk space than their original versions.

Delete Entry

Records may be deleted from the database. In such cases the disk space released is added onto a chain of 'free space' which might subsequently be used either for new records, or modified records which need extra room.

Search the Database

A variety of searches can be conducted. Essentially any search is based on the keywords that are stored with each record. On this basis, the program can be requested to produce a list of entries which possess up to three nominated 'search keywords'. A search keyword can be either 'positive' or 'negative'. If it is a 'positive' keyword then a matching record must have the given keyword as one of its keywords; if 'negative' (indicated by a '-' character before the keyword itself), then a match is only made with those records which do NOT possess the given keyword. Searches can be conducted either on a particular category, or on the whole of the database.

List Database Contents

The database can be listed in its entirety. The listing is conducted category by category, with records displayed at two-second intervals unless the user halts this cycle at any point.

As far as the list of functions is concerned, this is a Case Study which is quite typical of the unsophisticated database management programs often available for personal microcomputers such as the BBC Micro.

As will be seen under METHOD USED, however, the internal organisation of the database is a different matter altogether. Variable-length records are used, together with an indexing technique of the sort discussed in Chapter 9. In addition a method of garbage collection is employed so that space left by deleted records may be re-used. Given this sort of approach, and the fact that the Case Study is intended to supplement the text in helping the reader understand the techniques of using linked records, certain items of information are displayed whenever a record's contents are output. In particular the address of the record in the file and its pointer fields are output - such data would not normally be displayed with commercial packages.

For this reason we will need to depart briefly from the normal mode of presenting a Case Study to look at the format of a record in the database. This, although more appropriate to the METHOD USED section, is necessary if the SAMPLE DIALOGUE is to make complete sense.

Database Records

Every record in the database is of the following general form:

where <PTR> is the file address of the record itself, and where the different parts of the record are

ID	Identification byte
CAT	Category number
RL	Record length
PRED	Pointer to 'predecessor' record
SUCC	Pointer to 'successor' record
RECORD DATA	Data items for each field
KEYWORDS	3 keywords

Each of these components, for the purposes of illustration, forms part of the output display for each record of our example, thus:

```
<PTR>= &xxxx <PRED>= &xxxx  <SUCC>= &xxxx
Record Length= xxx (&xx) bytes

AUTHOR: xxxxxxxxxxxxx
TITLE: xxxxxxxxxxxxxxxxx
VOL.: xx
NO.: xx
PAGE(S): xx-xx
```

```
DATE: xxx/83
Keywords:
   1. xxxxxxx
   2. xxxxxxxxx
   3. xxxxx
```

This sort of display, especially when used in conjunction with *DUMP for the database file, enables a good picture to be built up of how the many records are being linked together. A number of actual examples appear in the following sample dialogue for program USEDB.

USEDB - SAMPLE DIALOGUE

```
>CHAIN "USEDB"
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
      DATABASE MANAGEMENT PROGRAM
      ***************************


Name of Database File?MUSE
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
Database MUSE Open
```

(The menu display below is the point to which the program always returns on completion of its current function or if ESCAPE is pressed. When the program is first called, this menu is presented after the database file has been opened and the index and field information read. In what follows, examples of the use of each of the menu functions will be given. They are presented in the order, ADD, LIST, ALTER, DELETE and SEARCH.)

```
1. ADD New Entry
2. ALTER Entry
3. DELETE Entry
4. SEARCH Database MUSE
5. LIST Database MUSE
6. QUIT

Which  function  (1-6)?1
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
                           (ADD new entry)
                           (=============)
ADD new entry

  0. *ANY* (SEARCH only)
  1. DEVICES
  2. EDUCATION
  3. GRAPHICS
  4. LANGUAGES
  5. OP.SYSTEMS
  6. PROGRAMMING
  7. REVIEWS
  8. SOFTWARE
  9. SOUND
 10. MISCELLANY

Which Category(0 -10)?6
```

(The user is now prompted for the data items which comprise the record. The prompts used are those which were defined as field names when the database was created with the MAKEDB program. It is not essential to

input data to every field; simply typing RETURN will cause the field content to be stored as blank. After the specified fields have been dealt with, the program then asks for three keywords to be associated with this particular entry. The idea here is that the keywords will be used as the basis of the database search facility. Careful specification of keywords at this stage, therefore, will be well rewarded later.)

```
Input Fields - RETURN to omit

AUTHOR?BIBBY,Pete
TITLE?Programming is easier than you think
VOLUME?1
NUMBER?1
PAGE(S)?14-18
DATE?MAR/83
Keyword 1?KEYBOARD
Keyword 2?BASIC
Keyword 3?   <---- RETURN (means no keyword 3)

Save this Entry(Y or N)?Y
New entry ADDED to Database MUSE

Any more ADDITIONS(Y or N)?Y
```

(As many additions can be made as are desired in one sitting. It is perhaps sensible, though, to limit the number in order to take periodic copies of the database file.)

```
ooooooooooooooooooooooooooooooooooooooScreen Cleared
ADD new entry

  0. *ANY* (SEARCH only)
  1. DEVICES
  2. EDUCATION
  3. GRAPHICS
  4. LANGUAGES
  5. OP.SYSTEMS
  6. PROGRAMMING
  7. REVIEWS
  8. SOFTWARE
  9. SOUND
 10. MISCELLANY

Which Category(0 -10)?1

Input Fields - RETURN to omit

AUTHOR?SHAW,Mike
TITLE?Build your own games paddle
VOLUME?1
```

```
NUMBER?1
PAGE(S)?20-22
DATE?MAR/83
Keyword 1?GAMES PADDLE
Keyword 2?ANALOGUE
Keyword 3?    <-------RETURN

Save this Entry(Y or N)?Y
New entry ADDED to Database MUSE

Any more ADDITIONS(Y or N)?N      (Returns to Menu)
```

(LIST Database)
(=============)

```
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
1. ADD New Entry
2. ALTER Entry
3. DELETE Entry
4. SEARCH Database MUSE
5. LIST Database MUSE
6. QUIT

Which function (1-6)?5
```

(This function lists the content of the database record by record in order of category. Each record is displayed for two seconds, and then the next record appears automatically. Typing any key during the two seconds will freeze this sequence so that the currently displayed record can be examined at length.

By way of example, a listing of our 'Micro User' database was taken after completing the entries for the first edition of the journal; this listing is given in Fig. 1.

```
Category:DEVICES

(PTR)= &36B (PRED)= FIRST (SUCC)
= &4F7
Record Length= 104 (&68) bytes

AUTHOR: SHAW,Mike
TITLE: Build your own games paddle
VOLUME: 1
NUMBER: 1
PAGE(S): 20-22
DATE: MAR/83
Keywords:
   1. GAMES_PADDLE
   2. ANALOGUE
   3.

Category:DEVICES

(PTR)= &4F7 (PRED)= &36B (SUCC)
= &6CB
Record Length= 105 (&69) bytes

AUTHOR: COOK,Mike
TITLE: Cut out those cassette
capers
VOLUME: 1
NUMBER: 1
PAGE(S): 38-39
DATE: MAR/83
Keywords:
   1. CASSETTE_TAPE
   2. ERRORS
   3.

Category:DEVICES

(PTR)= &6CB (PRED)= &4F7 (SUCC)
= &805
Record Length= 95 (&5F) bytes

AUTHOR: COOK,Mike
TITLE: Beeb Body building course
VOLUME: 1
NUMBER: 1
PAGE(S): 50-54
DATE: MAR/83
Keywords:
   1. UPGRADE
   2. A_TO_B
   3.

Category:DEVICES

(PTR)= &805 (PRED)= &6CB (SUCC)
= LAST
Record Length= 97 (&61) bytes

AUTHOR: NOELS,Michael
TITLE: How does your monitor check
out?
VOLUME: 1
NUMBER: 1
PAGE(S): 63
DATE: MAR/83
Keywords:
   1. MONITOR
   2.
   3.

Category:EDUCATION

(PTR)= &611 (PRED)= FIRST (SUCC)
= LAST
Record Length= 121 (&79) bytes

AUTHOR: CHANTRY-PRICE,Roger
(Interview)
TITLE: Spreading the micro gospel
in education
VOLUME: 1
NUMBER: 1
PAGE(S): 46-48
DATE: MAR/83
Keywords:
   1. MEP
   2.
   3.

Category:GRAPHICS

(PTR)= &429 (PRED)= FIRST (SUCC)
= &72A
Record Length= 97 (&61) bytes

AUTHOR: JONES,Paul
TITLE: A touch of the Van Goghs
VOLUME: 1
NUMBER: 1
PAGE(S): 26-31
DATE: MAR/83
Keywords:
   1. COLOUR
   2. MODE
   3. VDU19

Category:GRAPHICS

(PTR)= &72A (PRED)= &429 (SUCC)
= LAST
Record Length= 112 (&70) bytes

AUTHOR: NOELS,Michael
TITLE: Triangular Technicolour
Tangram
VOLUME: 1
NUMBER: 1
PAGE(S): 55-58
DATE: MAR/83
Keywords:
   1. SHAPES
   2. COLOUR
   3. SOFTWARE

Category:OP.SYSTEMS

(PTR) = &48A (PRED) = FIRST (SUCC)
= LAST
Record Length = 109 (&6D) bytes

AUTHOR: BEVERLEY,Paul
TITLE: Let's get down to those
basic building blocks
VOLUME: 1
NUMBER: 1
PAGE(S): 32-36
DATE: MAR/83
Keywords:
   1. ROM
   2.
   3.
```

```
Category:PROGRAMMING

(PTR)= &300 (PRED)= FIRST (SUCC)
= &79A
Record Length= 107 (&6B) bytes

AUTHOR: BIBBY,Mike
TITLE: Programming is easier than
you think
VOLUME: 1
NUMBER: 1
PAGE(S): 14-19
DATE: MAR/83
Keywords:
   1. KEYBOARD
   2. BASIC
   3.

Category:PROGRAMMING

(PTR)= &79A (PRED)= &300 (SUCC)
= &866
Record Length= 107 (&6B) bytes

AUTHOR: DAVIDSON,Peter
TITLE: Pick a Random Number ..
VOLUME: 1
NUMBER: 1
PAGE(S): 60-62
DATE: MAR/83
Keywords:
   1. BASIC
   2. RANDOM_NUMBER
   3. GAME

Category:PROGRAMMING

(PTR)= &866 (PRED)= &79A (SUCC)
= &8C5
Record Length= 95 (&5F) bytes

AUTHOR: THORPE,John
TITLE: Sorting things out
VOLUME: 1
NUMBER: 1
PAGE(S): 64-66
DATE: MAR/83
Keywords:
   1. BASIC
   2. PROCEDURE
   3. SORT

Category:PROGRAMMING

(PTR)= &8C5 (PRED)= &866 (SUCC)
= LAST
Record Length= 143 (&8F) bytes

AUTHOR: Programmer's Workshop
TITLE: Testing for function keys in
machine code routines
VOLUME: 1
NUMBER: 1
PAGE(S): 67-68
DATE: MAR/83
Keywords:
   1. MACHINE_CODE
   2. FUNCTION_KEY
   3.

Category:REVIEWS

(PTR)= &3D3 (PRED)= FIRST (SUCC)
= &68A
Record Length= 86 (&56) bytes

AUTHOR: JAMES,Paul
TITLE: ALPHABETA
VOLUME: 1
NUMBER: 1
PAGE(S): 24
DATE: MAR/83
Keywords:
   1. SOFTWARE
   2. WORD_PROCESSOR
   3.

Category:REVIEWS

(PTR)= &68A (PRED)= &3D3 (SUCC)
= LAST
Record Length= 65 (&41) bytes

AUTHOR: various
TITLE:
VOLUME: 1
NUMBER: 1
PAGE(S): 49
DATE: MAR/83
Keywords:
   1. SOFTWARE
   2. GAMES
   3.

Category:SOFTWARE

(PTR)= &560 (PRED)= FIRST (SUCC)
= LAST
Record Length= 86 (&56) bytes

AUTHOR: CLARK,Brian&Marian
TITLE: DEATHWATCH
VOLUME: 1
NUMBER: 1
PAGE(S): 40-41,73-75
DATE: MAR/83
Keywords:
   1. GAME
   2.
   3.

Category:DELETED RECS

(PTR)= &5B6 (PRED)= FIRST (SUCC)
= LAST
Record Length= 91 (&5B) bytes

AUTHOR: various
TITLE: Read all about it ..
VOLUME: 1
NUMBER: 1
PAGE(S): 43-44
DATE: MAR/83
Keywords:
   1. BOOKS
   2. PROGRAMMING
   3.
```

Fig. 1

(ALTER entry)
(===========)

```
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
1. ADD New Entry
2. ALTER Entry
3. DELETE Entry
4. SEARCH Database MUSE
5. LIST Database MUSE
6. QUIT

Which function (1-6)?2
```

(This function allows an existing database entry to be altered. It is necessary, before any changes can be made, for the program to locate the required record. Normally this entails the user quoting a record number, where this number is added by the system to form part of the record stored on disk. However, the technique used here, which highlights once more the organisation of the database file, is for the user to input the <PTR> value for the record - that is, the file address at which the record is located. For users not having a printer, the simplest way of finding the <PTR> value is through either a SEARCH or LIST operation.)

```
<PTR> of Entry to be ALTERED: &300
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
```

(If a record is found to start at this (hex) address it is input and displayed as a check.)

```
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
Category:PROGRAMMING

<PTR>= &300 <PRED>= FIRST <SUCC>= LAST
Record Length= 107 (&6B) bytes

AUTHOR: BIBBY,Pete
TITLE: Programming is easier than you think
VOLUME: 1
NUMBER: 1
PAGE(S): 14-18
DATE: MAR/83
Keywords:
   1. KEYBOARD
   2. BASIC
   3.

ALTER this Entry(Y or N)?Y
```

(Alteration of any field is by re-typing the complete field. RETURN is used with any field for which no change is needed.)

Input fields - RETURN leaves unchanged

AUTHOR?BIBBY,Mike
TITLE? <---RETURN
VOLUME? <---RETURN
NUMBER? <---RETURN
PAGE(S)?14-19
DATE? <---RETURN
Keyword 1? <---RETURN
Keyword 2? <---RETURN
Keyword 3? <---RETURN

(The new record is now writen to disk. There is no restriction on its length; it may be longer or shorter than the original.)

ALTERATION Done

Any more ALTERATIONS(Y or N)?N (Returns to Menu)

(DELETE Entry)
(============)

°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
1. ADD New Entry
2. ALTER Entry
3. DELETE Entry
4. SEARCH Database MUSE
5. LIST Database MUSE
6. QUIT

Which function (1-6)?3

(As with the ALTER function, DELETE needs to have the unwanted record specified in some way. In like fashion the user has to provide the <PTR> value for the record to be deleted.)

°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
<PTR> of Entry to be DELETED: &5B6

°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
Category:REVIEWS

<PTR>= &5B6 <PRED>= &3D3 <SUCC>= &68A
Record Length= 91 (&5B) bytes

AUTHOR: various
TITLE: Read all about it ..
VOLUME: 1
NUMBER: 1

```
PAGE(S): 43-44
DATE: MAR/83
Keywords:
   1. BOOKS
   2. PROGRAMMING
   3.

DELETE this entry(Y or N)?Y

DELETION Complete

Any more DELETIONS(Y or N)?N     (Returns to Menu)
```

(USEDB incorporates a garbage collection mechanism. Thus when a record is deleted, the space that it occupied is added to a chain of 'deleted records' which can be re-used. This is illustrated by showing this chain as part of the database listing. For instance, using the LIST operation (function 5) at this point would produce a listing, at the end of which would appear the entry,

```
Category:DELETED_RECS

<PTR>= &5B6 <PRED>= FIRST <SUCC>= LAST
Record Length= 91 (&5B) bytes

AUTHOR: various
TITLE: Read all about it ..
... etc
```

(Note that <PRED> and <SUCC> indicate that in this case there is just the one 'deleted record' in the chain.)

(SEARCH Database)
(===============)

```
oooooooooooooooooooooooooooooooooooooooScreen Cleared
1. ADD New Entry
2. ALTER Entry
3. DELETE Entry
4. SEARCH Database MUSE
5. LIST Database MUSE
6. QUIT

Which function (1-6)?4
```

(Once the database has been set up, the most common operation will be that of searching the database for records which match certain requirements. With the organisation used here, an obvious criterion could be that the search be confined to a particular category of record. Also by using the keywords associated with each record, the search

criteria may be narrowed further if the user wishes.)

```
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
Input Details for SEARCH:

  0. *ANY* (SEARCH only)
  1. DEVICES
  2. EDUCATION
  3. GRAPHICS
  4. LANGUAGES
  5. OP.SYSTEMS
  6. PROGRAMMING
  7. REVIEWS
  8. SOFTWARE
  9. SOUND
 10. MISCELLANY

Which Category(0 -10)?6
Keyword 1?PROCEDURE
Keyword 2?  <---RETURN
Keyword 3?  <---RETURN
```

(These inputs specify that the program should search through all records in the PROGRAMMING category of the database for any which have PROCEDURE as one of their keywords. All such records are to be displayed. In our example only one such record is to be found.)

```
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
Category:PROGRAMMING

<PTR>= &866 <PRED>= &79A <SUCC>= &8C5
Record Length= 95 (&5F) bytes

AUTHOR: THORPE,John
TITLE: Sorting things out
... etc
Keywords:
   1. BASIC
   2. PROCEDURE
   3. SORT

Press any key to continue X
```

(The next record which is found to match the search criteria would then be displayed. This continues until all such records have been displayed, in which case a return is made to the menu.

It is possible to stipulate that, for a match to be made, a record should NOT include a particular keyword. For instance, the way to search the database for any record having the keyword BASIC should be obvious. If, however, one is allergic to any trace of machine code

within a BASIC program, then a further keyword could specify -MACHINE_CODE, where the leading '-' is taken as the indication that the given keyword must not be associated with a record if it is to match.)

```
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°Screen Cleared
Input Details for SEARCH:

  0. *ANY* (SEARCH only)
  1. DEVICES
  2. EDUCATION
  3. GRAPHICS
  4. LANGUAGES
  5. OP.SYSTEMS
  6. PROGRAMMING
  7. REVIEWS
  8. SOFTWARE
  9. SOUND
 10. MISCELLANY

Which Category(0 -10)?0 (or RETURN)
Keyword 1?BASIC
Keyword 2?-MACHINE CODE
Keyword 3?
```

(The reader is invited to verify from Fig. 1 that only the following records would be matched:)

```
**     AUTHOR: BIBBY,Mike
       TITLE: Programming is easier than you think
       ... etc
       Keywords:
          1. KEYBOARD
          2. BASIC
          3.

**     AUTHOR: DAVIDSON,Peter
       TITLE: Pick a Random Number ..
       ... etc
       Keywords:
          1. BASIC
          2. RANDOM_NUMBER
          3. GAME

**     AUTHOR: THORPE,John
       TITLE: Sorting things out
       ... etc
       Keywords:
          1. BASIC
          2. PROCEDURE
          3. SORT
```

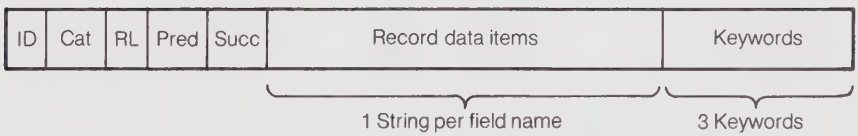
USEDB - METHOD USED

From the sample dialogue the reader should have gained the impression that it is the method for storage and retrieval of database records which is the key part of the whole program. This section, therefore, concentrates on this aspect and the way in which it applies to the different functions carried out by USEDB.

We will consider each function in turn. Before doing so, however, one general point has to be made. It is that, in one way or another, all of the functions need to access the information in the database index. For this reason the index data is held by the program throughout its execution in three arrays. These will always reflect the current state of the database, being copied to the file whenever they change in any way. Pictorially the arrays are

cat$	cstart%	cfin%
DEVICES	-1	-1
EDUCATION	-1	-1
.	.	.
.	.	.
.	.	.
MISCELLANY	-1	-1
DELETED_RECS	-1	-1

Array cat$ holds the categories, and cstart% and cfin% hold pointers to the records which start and finish the chain of records for their particular category. As a reminder, the format of a record is



We now consider each of the functions in turn.

ADD New Entry

When a new record is added to the database, its physical placement in the file will depend on whether or not an area of suitable size exists on the DELETED_RECS chain. If this is not the case, which will be the norm, then the sequence of events is as follows (use of cstart% and cfin% for the correct category is assumed):

(a) write the new record to disk starting from the next free position (nxfree%), i.e. add it to the end of the file;
(b) use the current cfin% entry to set a <SUCC> pointer to this new record into what was previously the last record of the chain;
(c) reset cfin% to point to the new record.

If there is an entry in the DELETED_RECS chain that is big enough to use, then the sequence becomes a little more complicated:

(a) unlink the deleted record from the DELETED_RECS chain, which means altering the <SUCC> and <PRED> pointers of its neighbours in this chain;
(b) using the <PRED> and <SUCC> pointers from the deleted record, locate the neighbours that it had in its category chain, and re-establish their <SUCC> and <PRED> values so as to put this record back into the chain;
(c) write the new record to the disk position of the deleted record.

ALTER Entry

On the face of it, altering an existing entry is no trouble at all. The record is input and displayed to the user, fields are either left alone or altered, and the modified record is then written back to its position in the database file. This is exactly how things work, if the modified record will fit into the disk space occupied by the original record. If this is not the case, the record having been made longer, then alternative steps have to be taken:

(a) delete the existing record, adding it to the DELETED_RECS chain, i.e. exactly as for 'DELETE Entry', below;
(b) add the modified version of the record to the file, i.e. exactly as for 'ADD New Entry', above.

DELETE Entry

To remove an existing record from its category chain, and then add it to the DELETED_RECS chain, involves the following steps:

(a) using the <PRED> and <SUCC> pointers of the record to be deleted, change the <SUCC> and <PRED> pointers of its neighbours so as to unlink the record from its category chain;

(b) add this record to the end of the DELETED_RECS chain by use of the same approach as given with steps (b) and (c) of 'ADD New Entry', above.

SEARCH Database

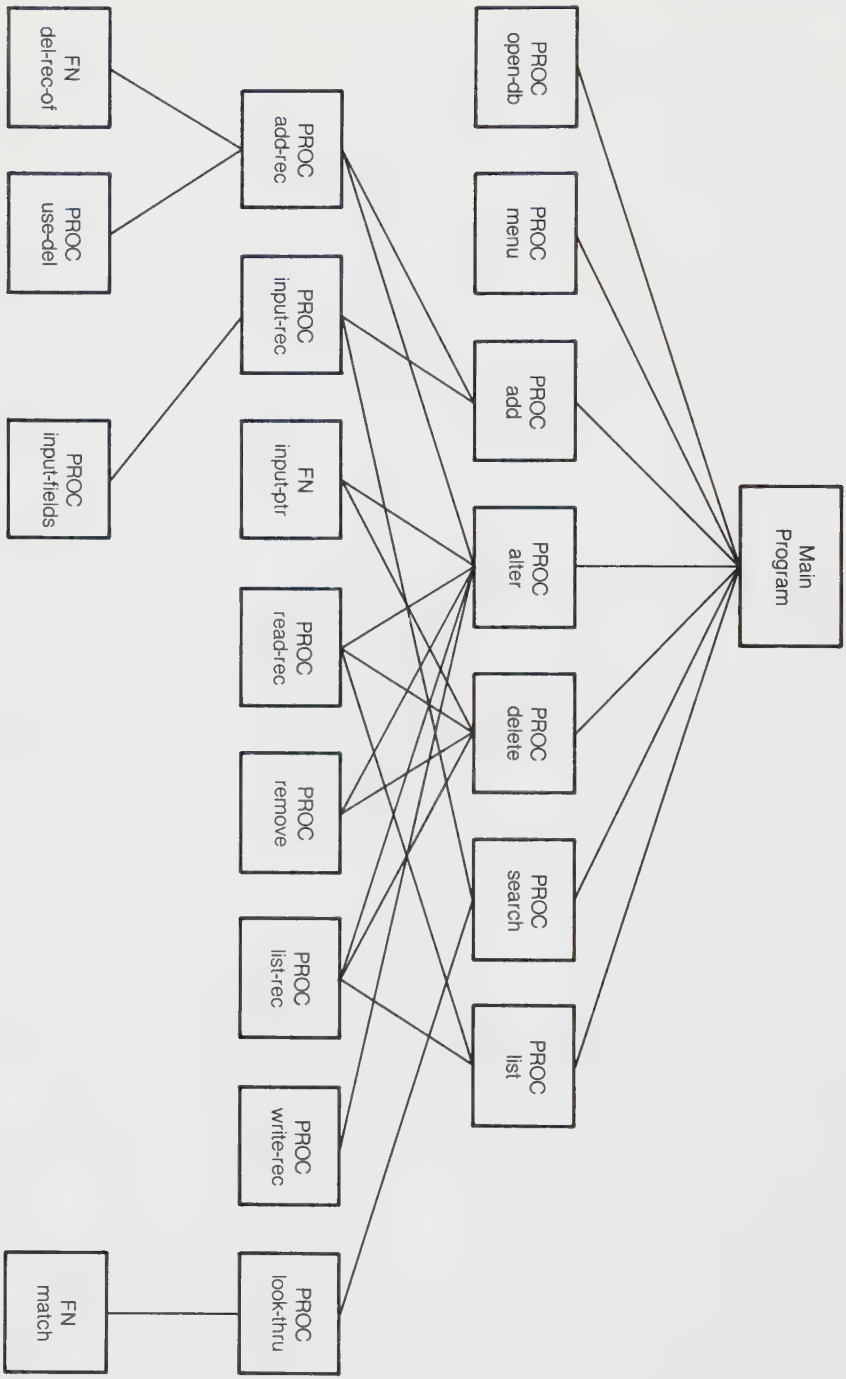
A search will either be conducted on a specified category, or on the whole of the database, i.e. all categories in turn! Whatever the case, the technique involves using the cstart% entry to locate the first record of the category chain, and <SUCC> pointers thereafter to work through the chain to its end.

For every record retrieved, a comparison has to be made of the stored keywords for the record against the keywords specified for the search, with the record being displayed if a match is found. This part is a little tricky, since the keywords can be given in any order, and because 'negative' keywords are allowed. The whole technique (see DEFFNmatch in the program) is based on the use of the Boolean types TRUE and FALSE, together with the logical operators AND, NOT and EOR - a spot of revision with the BBC User Guide might be advisable here!

LIST Database

No special technique is involved to list a record itself. What does happen, though, is that the listing is produced category by category, since this is the most natural way of retrieving items from the database - think of a listing as a search which matches every record!

With USEDB no specific attempt has been made to cater for the use of a printer (but see ROOMS FOR IMPROVEMENT). For this reason a delay mechanism is implemented so as to hold each record on the screen for two seconds during which the user can press a key to freeze the display. No key depression during this period means automatic display of the next record in sequence.


Main Program
PROC open-db
PROC menu
PROC add
PROC alter
PROC delete
PROC search
PROC list
PROC add-rec
PROC input-rec
FN input-ptr
PROC read-rec
PROC remove
PROC list-rec
PROC write-rec
PROC look-thru
FN del-rec-of
PROC use-del
PROC input-fields
FN match

USEDB - FULL PROGRAM LISTING

```
 10 REM*****************
 20 REM* CASE STUDY 5B *
 30 REM*****************

 40 fu%=0: ON ERROR PROCerr: IF ERR=17 AND fu%<>0 GOTO 100 ELSE END
 50 DIMcat$(40),cstart%(40),cfin%(40)
 60 DIMtype%(20),field$(20)
 70 DIMrec$(20),key$(3),mkey$(3),key_in%(3)
 80 CLS: PRINT'TAB(6);"DATABASE MANAGEMENT PROGRAM"'TAB(6);
    STRING$(27,"*")''
 90 PROCopen_db
100 REPEAT
110   func%=FNmenu: CLS
120   IF func%=1 PROCadd
130   IF func%=2 PROCalter
140   IF func%=3 PROCdelete
150   IF func%=4 PROCsearch
160   IF func%=5 PROClist
170   UNTIL func%=6: PRINT'"Exit from USEDB"'
180 CLOSE#fu%: END

190 DEFPROCopen_db: LOCAL i%,pt%: i%=0
200 INPUT'"Name of Database File",fnam$
210 fu%=OPENUP(fnam$): REM Use OPENIN  with BASIC I
220 INPUT#fu%,nxfree%
230  REPEAT: i%=i%+1: pt%=PTR#fu%
240   INPUT#fu%,cat$(i%): PTR#fu%=pt%+14
250   INPUT#fu%,cstart%(i%),cfin%(i%)
260   UNTIL cat$(i%)="DELETED_RECS": nc%=i%-1: del%=i%
270 PTR#fu%=FNnext_sector: i%=0
280 REPEAT: i%=i%+1
290    type%(i%)=BGET#fu%
300   IF type%(i%)<>ASC("|") INPUT#fu%,field$(i%)
310    UNTIL type%(i%)=ASC("|"): nf%=i%-1
320 ENDPROC

330 DEFFNmenu: LOCAL fc%,E$: E$=" Entry"
340 CLS: PRINT'"Database ";fnam$;" Open"'
350 PRINT"1. ADD New";E$
360 PRINT"2. ALTER";E$
370 PRINT"3. DELETE";E$
380 PRINT"4. SEARCH Database ";fnam$
390 PRINT"5. LIST Database ";fnam$
400 PRINT"6. QUIT"'
410 REPEAT
420   INPUT"Which function (1-6)",fc%
430   UNTIL fc%>0 AND fc%<7: =fc%
```

```
440 DEFPROCadd
450 REPEAT
460   CLS:PRINT'"ADD new entry"'
470   PROCinput_rec("ADD")
480   IF FNyes("Save this Entry") PROCadd_rec:
       PRINT"New entry ADDED to Database ";fnam$
490   UNTIL NOT FNyes("Any more ADDITIONS")
500 ENDPROC

510 DEFPROCalter: LOCAL old_rl%
520 REPEAT
530   CLS: ptr%=FNinput_ptr("Entry to be ALTERED")
540   PROCread_rec(ptr%)
550   PROClist_rec(ptr%)
560   IF NOT FNyes("ALTER this Entry") GOTO 590
570   PROCinput_fields("MOD"):
       IF new_rl%<=rl%THEN rl%=new_rl%:PROCwrite_rec(ptr%)
       ELSE PROCremove(ptr%): rl%=new_rl%: PROCadd_rec
580   PRINT'"ALTERATION Done"'
590   UNTIL NOT FNyes("Any more ALTERATIONS")
600 ENDPROC

610 DEFPROCdelete
620 REPEAT
630   CLS: ptr%=FNinput_ptr("Entry to be DELETED")
640   PROCread_rec(ptr%): PROClist_rec(ptr%)
650   IF FNyes("DELETE this entry") PROCremove(ptr%):
       PRINT'"DELETION Complete"
660   UNTIL NOT FNyes("Any more DELETIONS")
670 ENDPROC

680 DEFPROCsearch: LOCAL i%
690 CLS: PRINT'"Input Details for SEARCH:"'
700 PROCinput_rec("FIND")
710 FOR i%=1 TO 3: mkey$(i%)=key$(i%)
720   IF LEFT$(mkey$(i%),1)="-" THEN mkey$(i%)=MID$(mkey$(i%),2):
        key_in%(i%)=FALSE ELSE key_in%(i%)=TRUE
730   NEXT
740 IF cno%<>0 THEN PROClook_thru(cno%) ELSE
     FOR cno%=1 TO nc%: PROClook_thru(cno%): NEXT
750 ENDPROC

760 DEFPROClist: LOCAL cat%,more%,pt%: cat%=1: more%=TRUE
770 REPEAT
780   pt%=cstart%(cat%)
790   IF pt%=-1 GOTO 860
800   REPEAT
810     PROCread_rec(pt%): PROClist_rec(pt%)
820     PRINT'"Press any key to halt display";
830     IF INKEY(200)<>-1 THEN more%=FNyes("CONTINUE")
```

```
          ELSE more%=TRUE
  840     IF more% THEN pt%=succ%
  850     UNTIL pt%=-1 OR NOT more%
  860   cat%=cat%+1
  870   UNTIL cat%>del% OR NOT more%
  880 ENDPROC

  890 DEFPROCinput_rec(s$): LOCAL i%,ok%: @%=3
  900 PRINT'"  0. *ANY* (SEARCH only)"
  910  FOR i%=1 TO nc%: PRINTi%;". ";cat$(i%): NEXT: PRINT
  920 REPEAT
  930   PRINT"Which Category(0 -";nc%;")";: INPUTcno%
  940   ok%=(cno%>=0 AND cno%<=nc%)
  950   IF s$="ADD" AND cno%=0
         PRINT"Non-zero category with ADD": ok%=FALSE
  960   UNTIL ok%: rec$(0)=cat$(cno%)
  970 PROCinput_fields(s$)
  980 ENDPROC

  990 DEFPROCinput_fields(s$): LOCAL i%,x$
 1000 IF s$="FIND" GOTO1070
 1010 PRINT'"Input fields - RETURN ";
        IF s$="MOD" PRINT"leaves unchanged"' ELSE PRINT"to omit"'
 1020 FOR i%=1 TO nf%
 1030   PRINTfield$(i%);: INPUTLINEx$
 1040   IF type%(i%)=ASC("N") AND x$<>"" THEN
         IF (VALx$=0 AND x$<>"0") THEN PRINT'field$(i%);
          " is NUMERIC"': GOTO 1030
 1050   IF NOT(s$="MOD" AND x$="") THEN rec$(i%)=x$
 1060   NEXT
 1070 FOR i%=1 TO 3
 1080   PRINT"Keyword ";i%;: INPUTLINEx$
 1090   IF NOT (s$="MOD" AND x$="") THEN key$(i%)=x$
 1100   NEXT
 1110 IF s$="MOD" THEN new_rl%=FNrec_len ELSE rl%=FNrec_len
 1120 ENDPROC

 1130 DEFPROCadd_rec: LOCAL pt%,i%
 1140 pt%=FNdel_rec_of(rl%): IFpt%<>-1 PROCuse_del(pt%): GOTO 1220
 1150 IF nxfree%+rl%>=EXT#fu% PRINT'"***Database ";fnam$;
       " is FULL***"': ENDPROC
 1160 succ%=-1: pred%=cfin%(cno%)
 1170 IF pred%<>-1 THEN PTR#fu%=pred%+12: PRINT#fu%,nxfree%
 1180 cfin%(cno%)=nxfree%: IF cstart%(cno%)=-1
       THEN cstart%(cno%)=nxfree%
 1190 pt%=nxfree%: PROCwrite_rec(pt%)
 1200 nxfree%=pt%+rl%
 1210 PTR#fu%=0: PRINT#fu%,nxfree%
 1220 PTR#fu%=FNcpos(cno%): PRINT#fu%,cstart%(cno%),cfin%(cno%)
 1230 ENDPROC
```

```
1190 DEFFNdel_rec_of(needed%): LOCAL pt%,size%
1200 pt%=cstart%(del%): IF pt%=-1 THEN =pt%
1260 REPEAT
1270   PTR#fu%=pt%+1: INPUT#fu%,size%
1280   IF size%<needed% THEN PTR#fu%=PTR#fu%+5:INPUT#fu%,pt%
1290   UNTIL size%>=needed% OR pt%=-1
1300 =pt%

1310 DEFPROCuse_del(pt%)
1320 PTR#fu%=pt%+1: INPUT#fu%,pred%,succ%
1330 IF pred%<>-1 THEN PTR#fu%=pred%+11: PRINT#fu%,succ%
      ELSE cstart%(del%)=succ%
1340 IF succ%<>-1 THEN PTR#fu%=succ%+7: PRINT#fu%,pred%
      ELSE cfin%(del%)=pred%
1350 PTR#fu%=fvpos(del%): PRINT#fu%,cstart%(del%),cfin%(del%);
      succ%=-1: pred%=cfin%(cno%)
1360 IF pred%<>-1 THEN PTR#fu%=pred%+12: PRINT#fu%,pt%
      ELSE cstart%(cno%)=pt%
1370 cfin%(cno%)=pt%: PROCwrite_rec(pt%)
1380 ENDPROC

1390 DEFPROCwrite_rec(pt%): LOCAL i%
1400 PTR#fu%=pt%
1410 BPUT#fu%,&FE: BPUT#fu%,cno%: PRINT#fu%,rl%,pred%,succ%
1420 FOR i%=1 TO nf%: PRINT#fu%,rec$(i%): NEXT
1430 FOR i%=1 TO 3: PRINT#fu%,key$(i%): NEXT
1440 ENDPROC

1450 DEFFNinput_ptr(s$): LOCAL pt%,pt$,id%
1460 REPEAT
1470   PRINT'"<PTR> of ";s$;:INPUT": &" pt$: pt$="&"+pt$:
         pt%=EVAL(pt$): PTR#fu%=pt%: id%=BGET#fu%
1480   IF id%<>&FE PRINT'"Record not found at this position"'
1490   UNTIL id%=&FE
1500 =pt%

1510 DEFPROCread_rec(pt%): LOCAL i%
1520 PTR#fu%=pt%+1: cno%=BGET#fu%
1530 INPUT#fu%,rl%,pred%,succ%
1540 FOR i%=1 TO nf%: INPUT#fu%,rec$(i%): NEXT
1550 FOR i%=1 TO 3: INPUT#fu%,key$(i%): NEXT
1560 ENDPROC

1570 DEFPROClist_rec(pt%): LOCAL i%: @%=4
1580 CLS: PRINT'"Category:";cat$(cno%)
1590 PRINT'"<PTR>= &";~pt%;" <PRED>= ";:
      IF pred%=-1 PRINT"FIRST";ELSE PRINT"&";~pred%;
1600 PRINT" <SUCC>= ";:
      IF succ%=-1 PRINT"LAST" ELSE PRINT"&";~succ%
1610 PRINT"Record Length= ";rl%;" (&";~rl%;") bytes"'
```

```
1620 FOR i%=1 TO nf%
1630   PRINTfield$(i%);": ";rec$(i%)
1640   NEXT
1650 PRINT"Keywords:":FOR i%=1 TO 3: PRINTi%;". ";key$(i%):
       NEXT
1660 ENDPROC

1670 DEFPROCremove(pt%)
1680 IF pred%<>-1 THEN PTR#fu%=pred%+12: PRINT#fu%,succ%
      ELSE cstart%(cno%)=succ%
1690 IF succ%<>-1 THEN PTR#fu%=succ%+7: PRINT#fu%,pred%
      ELSE cfin%(cno%)=pred%
1700 succ%=-1: pred%=cfin%(del%)
1710 IF pred%<>-1 THEN PTR#fu%=pred%+12: PRINT#fu%,pt%
      ELSE cstart%(del%)=pt%
1720 cfin%(del%)=pt%: PTR#fu%=pt%: BPUT#fu%,&FD: BPUT#fu%,del%:
      PRINT#fu%,rl%,pred%,succ%
1730 PTR#fu%=FNcpos(del%): PRINT#fu%,cstart%(del%),cfin%(del%)
1740 PTR#fu%=FNcpos(cno%): PRINT#fu%,cstart%(cno%),cfin%(cno%)
1750 ENDPROC

1760 DEFPROClook_thru(cat%): LOCAL pt%,x%
1770 pt%=cstart%(cat%)
1780 IF pt%=-1 ENDPROC
1790 REPEAT
1800   PROCread_rec(pt%)
1810   IF FNmatch THEN PROClist_rec(pt%):
        PRINT'"Press any key to continue"';: x%=GET
1820   pt%=succ%
1830   UNTIL pt%=-1
1840 ENDPROC

1850 DEFFNmatch: LOCAL i%,j%,found%,ok%: ok%=TRUE
1860 FOR i%=1 TO 3
1870   IF mkey$(i%)="" GOTO 1930
1880   found%=FALSE
1890   FOR j%=1 TO 3
1900     IF mkey$(i%)=key$(j%) THEN found%=TRUE
1910     NEXT
1920   ok%=ok% AND NOT (found% EOR key_in%(i%))
1930   NEXT
1940 =ok%

1950 DEFFNcpos(cno%): =5+(cno%-1)*24+14

1960 DEFFNrec_len: LOCAL rl%: rl%=0
1970 FOR i%=1 TO nf%: rl%=rl%+LENrec$(i%)+2: NEXT
1980 FOR i%=1 TO 3: rl%=rl%+LENkey$(i%)+2: NEXT
1990 =rl%+17
```

```
2000 DEFFNyes(s$): LOCAL x$
2010 REPEAT
2020   PRINT's$;"(Y or N)";: INPUTx$
2030   UNTIL x$="Y"ORx$="y"ORx$="N"ORx$="n"
2040 =(x$="Y"ORx$="y")

2050 DEFFNnext_sector
2060 =(PTR#fu%DIV256+1)*256

2070 DEFPROCerr: IF ERR=17 AND fu%<>0 ENDPROC
2080 PRINT'"Error ";ERR;" in Line ";ERL: REPORT: PRINT
2090 IF fu%<>0 CLOSE#fu%
2100 ENDPROC
```

USEDB - ANNOTATED PROGRAM LISTING

List of Global Variables used:

STRING ARRAYS

cat$	Category names
field$	Field names
key$	Keywords for current record
mkey$	Keywords to be matched during search
rec$	Current record, one field per array item

INTEGER ARRAYS

cstart%,cfin%	Start/finish of record chains
key_in%	Search keyword positive/negative flags
type%	Field type (ASCII code for "S"/"N")

STRINGS

fnam$	Database file name

INTEGERS

del%	Pointer to DELETED_RECS entry of index
func%	Menu function selected
fu%	Channel number for database file
nc%	Number of categories
new_rl%	Length of altered record
nf%	Number of fields per record
nxfree%	Pointer to next free byte of database file
pred%,succ%	Predecessor, successor pointers
rl%	Length of current record

```
10 REM*****************
20 REM* CASE STUDY 5B *
30 REM*****************
```

(The arrays used by the program are set here to have dimensions which are fairly arbitrary. There is no reason why any database should not have more than 40 categories for its records, or why any record should not have more than 20 fields (including id_byte, record length and pointers). What is important is that any dimension change from those given should be applied to all arrays sharing that dimension.)

```
40 fu%=0: ON ERROR PROCerr: IF ERR=17 AND fu%<>0 GOTO 100 ELSE END
50 DIMcat$(40),cstart%(40),cfin%(40)
60 DIMtype%(20),field$(20)
70 DIMrec$(20),key$(3),mkey$(3),key_in%(3)
```

(Following a self-introduction, the program opens the specified database file for updating. It then uses the standard menu presentation loop, performing selected operations until the user wishes to go to bed. At this point the database file is closed.)

```
 80 CLS: PRINT'TAB(6);"DATABASE MANAGEMENT PROGRAM"'TAB(6);
    STRING$(27,"*")''
 90 PROCopen_db
100 REPEAT
110   func%=FNmenu: CLS
120   IF func%=1 PROCadd
130   IF func%=2 PROCalter
140   IF func%=3 PROCdelete
150   IF func%=4 PROCsearch
160   IF func%=5 PROClist
170   UNTIL func%=6: PRINT'"Exit from USEDB"'
180 CLOSE#fu%: END
```

(Procedure 'open_db' opens a selected database file. It then reads from that file the stored index - category names (into array cat$), start and finish pointers (into cstart% and cfin%) for each category chain of records. A marker (nc%) is set to the number of categories, with another (del%) set for convenience as an index to the DELETED_RECS entry. A second loop reads the stored field names (into field$) and their respective data types (into type%), with a marker (nf%) being set to the number of fields per record.)

```
190 DEFPROCopen_db: LOCAL i%,pt%: i%=0
200 INPUT'"Name of Database File",fnam$
210 fu%=OPENUP(fnam$): REM Use OPENIN  with BASIC I
220 INPUT#fu%,nxfree%
230  REPEAT: i%=i%+1: pt%=PTR#fu%
240   INPUT#fu%,cat$(i%): PTR#fu%=pt%+14
250   INPUT#fu%,cstart%(i%),cfin%(i%)
260   UNTIL cat$(i%)="DELETED_RECS": nc%=i%-1: del%=i%
270 PTR#fu%=FNnext_sector: i%=0
280 REPEAT: i%=i%+1
290    type%(i%)=BGET#fu%
300   IF type%(i%)<>ASC("|") INPUT#fu%,field$(i%)
310    UNTIL type%(i%)=ASC("|"): nf%=i%-1
320 ENDPROC
```

(Function 'menu' displays the available operations and asks the user to take a pick. After a check the selection number is returned as the value of the function.)

```
330 DEFFNmenu: LOCAL fc%,E$: E$=" Entry"
340 CLS: PRINT'"Database ";fnam$;" Open"'
350 PRINT"1. ADD New";E$
360 PRINT"2. ALTER";E$
370 PRINT"3. DELETE";E$
380 PRINT"4. SEARCH Database ";fnam$
390 PRINT"5. LIST Database ";fnam$
400 PRINT"6. QUIT"'
```

```
  410 REPEAT
  420   INPUT"Which function (1-6)",fc%
  430   UNTIL fc%>0 AND fc%<7: =fc%
```

(Procedure 'add' is called to add a new entry to the database. It prompts the user to input the details of the new record (PROCinput_rec) and, after getting confirmation from the user that this record should be added to the file, it calls PROCadd_rec to do the job. These steps are repeated until the user has completed the current batch of new entries and returns to the menu.)

```
  440 DEFPROCadd
  450 REPEAT
  460   CLS:PRINT'"ADD new entry"'
  470   PROCinput_rec("ADD")
  480   IF FNyes("Save this Entry") PROCadd_rec:
        PRINT"New entry ADDED to Database ";fnam$
  490   UNTIL NOT FNyes("Any more ADDITIONS")
  500 ENDPROC
```

(Procedure 'alter' is called to modify an existing record in the database. The <PTR> value for this record is requested, and the selected record input (PROCread_rec) and displayed (PROClist_rec). Assuming that the user decides to alter this entry, he is prompted for fields in turn (PROCinput_fields). If the modified record is not longer than its original, it is written back to the file (PROCwrite_rec); if it is, the original is deleted (PROCremove) and the new version treated as an addition (PROCadd_rec). Again this sequence loops until terminated by the user.)

```
  510 DEFPROCalter: LOCAL old_rl%
  520 REPEAT
  530   CLS: ptr%=FNinput_ptr("Entry to be ALTERED")
  540   PROCread_rec(ptr%)
  550   PROClist_rec(ptr%)
  560   IF NOT FNyes("ALTER this Entry") GOTO 590
  570   PROCinput_fields("MOD"):
        IF new_rl%<=rl%THEN rl%=new_rl%:PROCwrite_rec(ptr%)
        ELSE PROCremove(ptr%): rl%=new_rl%: PROCadd_rec
  580   PRINT'"ALTERATION Done"'
  590   UNTIL NOT FNyes("Any more ALTERATIONS")
  600 ENDPROC
```

(Procedure 'delete', entered for obvious reasons, starts by asking for the <PTR> value of the victim. It then reads and displays this record in case a stay of execution is to be granted. If not, PROCremove is called to carry out the deed.)

```
  610 DEFPROCdelete
  620 REPEAT
```

```
  630   CLS: ptr%=FNinput_ptr("Entry to be DELETED")
  640   PROCread_rec(ptr%): PROClist_rec(ptr%)
  650   IF FNyes("DELETE this entry") PROCremove(ptr%):
         PRINT'"DELETION Complete"
  660   UNTIL NOT FNyes("Any more DELETIONS")
  670 ENDPROC
```

(Procedure 'search' starts by asking (PROCinput_rec) for the search category (cno%) and search keywords (mkey$) to be input. If cno% is returned as 0, all categories are to be searched. Array elements key_in% are set to TRUE/FALSE depending on whether the mkey$ are defined as 'positive' or 'negative'; if the latter, the '-' character is removed from the keyword. The actual search is conducted by PROClook_thru, operating on one or all of the category chains as required.)

```
  680 DEFPROCsearch: LOCAL i%
  690 CLS: PRINT'"Input Details for SEARCH:"'
  700 PROCinput_rec("FIND")
  710 FOR i%=1 TO 3: mkey$(i%)=key$(i%)
  720   IF LEFT$(mkey$(i%),1)="-" THEN mkey$(i%)=MID$(mkey$(i%),2):
         key_in%(i%)=FALSE ELSE key_in%(i%)=TRUE
  730   NEXT
  740 IF cno%<>0 THEN PROClook_thru(cno%) ELSE
       FOR cno%=1 TO nc%: PROClook_thru(cno%): NEXT
  750 ENDPROC
```

(Procedure 'list' is basically a REPEAT loop which works through each category chain in turn displaying every record that it meets. Note that the DELETED_RECS chain is included in the loop. After a record is displayed the user has two seconds in which to depress any key to cause the loop to pause on a CONTINUE? prompt. No action on the part of the user means that the loop continues on its merry way.)

```
  760 DEFPROClist: LOCAL cat%,more%,pt%: cat%=1: more%=TRUE
  770 REPEAT
  780   pt%=cstart%(cat%)
  790   IF pt%=-1 GOTO 860
  800   REPEAT
  810     PROCread_rec(pt%): PROClist_rec(pt%)
  820     PRINT'"Press any key to halt display";
  830     IF INKEY(200)<>-1 THEN more%=FNyes("CONTINUE")
           ELSE more%=TRUE
  840     IF more% THEN pt%=succ%
  850     UNTIL pt%=-1 OR NOT more%
  860   cat%=cat%+1
  870   UNTIL cat%>del% OR NOT more%
  880 ENDPROC
```

(Procedure 'input_rec' reads a record from the keyboard. It displays a list of the categories and asks for one to be selected (cno%) checking that, for a new record, *ANY* is not the option chosen. Procedure 'input_fields' is called to obtain the field contents for the record.)

```
  890 DEFPROCinput_rec(s$): LOCAL i%,ok%: @%=3
  900 PRINT'"  0. *ANY* (SEARCH only)"
  910  FOR i%=1 TO nc%: PRINTi%;". ";cat$(i%): NEXT: PRINT
  920 REPEAT
  930   PRINT"Which Category(0 -";nc%;")";: INPUTcno%
  940   ok%=(cno%>=0 AND cno%<=nc%)
  950   IF s$="ADD" AND cno%=0
         PRINT"Non-zero category with ADD": ok%=FALSE
  960   UNTIL ok%: rec$(0)=cat$(cno%)
  970 PROCinput_fields(s$)
  980 ENDPROC
```

(Procedure 'input_fields', for ADD or MOD functions, firstly prompts the user for each field of a record in turn, the responses being checked against the stored type for that field and input again if necessary. These strings are stored in the array rec$, unless the string is null and the record is being input as part of an 'ALTER Entry' operation, in which case the element of rec$ is left unchanged. Whatever the current function the three keywords are then input, the final step being the setting of the record length variable in use.)

```
  990 DEFPROCinput_fields(s$): LOCAL i%,x$
 1000 IF s$="FIND" GOTO1070
 1010 PRINT'"Input fields - RETURN ";:
       IF s$="MOD" PRINT"leaves unchanged"' ELSE PRINT"to omit"'
 1020 FOR i%=1 TO nf%
 1030   PRINTfield$(i%);: INPUTLINEx$
 1040   IF type%(i%)=ASC("N") AND x$<>"" THEN
         IF (VALx$=0 AND x$<>"0") THEN PRINT'field$(i%);
          " is NUMERIC"': GOTO 1030
 1050   IF NOT(s$="MOD" AND x$="") THEN rec$(i%)=x$
 1060   NEXT
 1070 FOR i%=1 TO 3
 1080   PRINT"Keyword ";i%;: INPUTLINEx$
 1090   IF NOT (s$="MOD" AND x$="") THEN key$(i%)=x$
 1100   NEXT
 1110 IF s$="MOD" THEN new_rl%=FNrec_len ELSE rl%=FNrec_len
 1120 ENDPROC
```

(Procedure 'add_rec' carries out the operation of adding a record to the database. It determines whether or not a deleted record exists of suitable length (FNdel_rec_of) and, if so, writes the record to that position (PROCuse_del). If not, it adds the record to the end of the file, checking firstly that the database is not full! The reader is referred to METHOD USED for the steps involved. Notice that not only is

the new record written to disk, but also the updated value of nxfree% and the relevant items from cstart% and cfin%, since they may well have changed.)

```
 1130 DEFPROCadd_rec: LOCAL pt%,i%
 1140 pt%=FNdel_rec_of(rl%): IFpt%<>-1 PROCuse_del(pt%): GOTO 1220
 1150 IF nxfree%+rl%>=EXT#fu% PRINT'""***Database ";fnam$;
      " is FULL***"'': ENDPROC
 1160 succ%=-1: pred%=cfin%(cno%)
 1170 IF pred%<>-1 THEN PTR#fu%=pred%+12: PRINT#fu%,nxfree%
 1180 cfin%(cno%)=nxfree%: IF cstart%(cno%)=-1
      THEN cstart%(cno%)=nxfree%
 1190 pt%=nxfree%: PROCwrite_rec(pt%)
 1200 nxfree%=pt%+rl%
 1210 PTR#fu%=0: PRINT#fu%,nxfree%
 1220 PTR#fu%=FNcpos(cno%): PRINT#fu%,cstart%(cno%),cfin%(cno%)
 1230 ENDPROC
```

(Function 'del_rec_of' scans through the chain of deleted records, looking at the record length (rl%) item of each. If any record is discovered with a length greater than or equal to its parameter, needed%, then the function returns the PTR# value of that record; if not, it returns the value -1.)

```
 1240 DEFFNdel_rec_of(needed%): LOCAL pt%,size%
 1250 pt%=cstart%(del%): IF pt%=-1 THEN =pt%
 1260 REPEAT
 1270   PTR#fu%=pt%+2: INPUT#fu%,size%
 1280   IF size%<needed% THEN PTR#fu%=PTR#fu%+5:INPUT#fu%,pt%
 1290   UNTIL size%>=needed% OR pt%=-1
 1300 =pt%
```

(Procedure 'use_del%' copies the current record to position pt% in the file, this being the position of a previously deleted record. The old record is unlinked from the DELETED_RECS chain and added on to the end of the chain for the category of the new record. The new record is finally written to the disk.)

```
 1310 DEFPROCuse_del(pt%)
 1320 PTR#fu%=pt%+7: INPUT#fu%,pred%,succ%
 1330 IF pred%<>-1 THEN PTR#fu%=pred%+12: PRINT#fu%,succ%
      ELSE cstart%(del%)=succ%
 1340 IF succ%<>-1 THEN PTR#fu%=succ%+7: PRINT#fu%,pred%
      ELSE cfin%(del%)=pred%
 1350 PTR#fu%=FNcpos(del%): PRINT#fu%,cstart%(del%),cfin%(del%)
      succ%=-1: pred%=cfin%(cno%)
 1360 IF pred%<>-1 THEN PTR#fu%=pred%+12: PRINT#fu%,pt%
      ELSE cstart%(cno%)=pt%
 1370 cfin%(cno%)=pt%: PROCwrite_rec(pt%)
 1380 ENDPROC
```

(Procedure 'write_rec' transfers a record to the database file at an address given by its parameter pt%. The elements of the record are output in the sequence, id_byte(&FE), category number, record length, predecessor and successor pointers, data items for each field (from rec$), and finally key$, the keywords.)

```
1390 DEFPROCwrite_rec(pt%): LOCAL i%
1400 PTR#fu%=pt%
1410 BPUT#fu%,&FE: BPUT#fu%,cno%: PRINT#fu%,rl%,pred%,succ%
1420 FOR i%=1 TO nf%: PRINT#fu%,rec$(i%): NEXT
1430 FOR i%=1 TO 3: PRINT#fu%,key$(i%): NEXT
1440 ENDPROC
```

(Function 'input_ptr' reads a hexadecimal <PTR> value for use with either the 'Alter' or 'Delete' functions. The value should be the start address of a record, and thus be the actual address of an identification byte. If this is not the case then another value is requested.)

```
1450 DEFFNinput_ptr(s$): LOCAL pt%,pt$,id%
1460 REPEAT
1470   PRINT'"<PTR> of ";s$;:INPUT": &" pt$: pt$="&"+pt$:
          pt%=EVAL(pt$): PTR#fu%=pt%: id%=BGET#fu%
1480   IF id%<>&FE PRINT'"Record not found at this position"'
1490   UNTIL id%=&FE
1500 =pt%
```

(Procedure 'read_rec' is the converse of PROCwrite_rec. It reads a record from disk to the global variables used in the program.)

```
1510 DEFPROCread_rec(pt%): LOCAL i%
1520 PTR#fu%=pt%+1: cno%=BGET#fu%
1530 INPUT#fu%,rl%,pred%,succ%
1540 FOR i%=1 TO nf%: INPUT#fu%,rec$(i%): NEXT
1550 FOR i%=1 TO 3: INPUT#fu%,key$(i%): NEXT
1560 ENDPROC
```

(Procedure 'list_rec' displays the current record. The reader is referred to the SAMPLE DIALOGUE for many examples of the sort of output produced, from which it should be easy to see how the procedure works.)

```
1570 DEFPROClist_rec(pt%): LOCAL i%: @%=4
1580 CLS: PRINT'"Category:";cat$(cno%)
1590 PRINT'"<PTR>= &";~pt%;" <PRED>= ";:
      IF pred%=-1 PRINT"FIRST";ELSE PRINT"&";~pred%;
1600 PRINT" <SUCC>= ";:
      IF succ%=-1 PRINT"LAST" ELSE PRINT"&";~succ%
1610 PRINT"Record Length= ";rl%;" (&";~rl%;") bytes"'
1620 FOR i%=1 TO nf%
1630   PRINTfield$(i%);": ";rec$(i%)
```

```
 1640   NEXT
 1650 PRINT"Keywords:":FOR i%=1 TO 3: PRINTi%;". ";key$(i%): NEXT
 1660 ENDPROC
```

(Procedure 'remove' carries out the task of deleting the record at address pt%. Basically it does its job in two stages, unlinking the record from its category chain, and then adding it to the chain of DELETED_RECS. The record is not physically moved, but it is altered - the identification byte is changed from &FE to &FD to indicate that this is now a deleted record, and the category number is set to del%.)

```
 1670 DEFPROCremove(pt%)
 1680 IF pred%<>-1 THEN PTR#fu%=pred%+12: PRINT#fu%,succ%
       ELSE cstart%(cno%)=succ%
 1690 IF succ%<>-1 THEN PTR#fu%=succ%+7: PRINT#fu%,pred%
       ELSE cfin%(cno%)=pred%
 1700 succ%=-1: pred%=cfin%(del%)
 1710 IF pred%<>-1 THEN PTR#fu%=pred%+12: PRINT#fu%,pt%
       ELSE cstart%(del%)=pt%
 1720 cfin%(del%)=pt%: PTR#fu%=pt%: BPUT#fu%,&FD: BPUT#fu%,del%:
       PRINT#fu%,r1%,pred%,succ%
 1730 PTR#fu%=FNcpos(del%): PRINT#fu%,cstart%(del%),cfin%(del%)
 1740 PTR#fu%=FNcpos(cno%): PRINT#fu%,cstart%(cno%),cfin%(cno%)
 1750 ENDPROC
```

(Procedure 'look_thru' comprises a REPEAT loop which will examine every record in the category chain specified by its parameter cat%. Each record is read, checked (FNmatch) and, if it is found to meet the search criteria, listed. The loop continues on the user's say so.)

```
 1760 DEFPROClook_thru(cat%): LOCAL pt%,x%
 1770 pt%=cstart%(cat%)
 1780 IF pt%=-1 ENDPROC
 1790 REPEAT
 1800   PROCread_rec(pt%)
 1810   IF FNmatch THEN PROClist_rec(pt%):
         PRINT'"Press any key to continue"';: x%=GET
 1820   pt%=succ%
 1830   UNTIL pt%=-1
 1840 ENDPROC
```

(Function 'match' compares each of the three keywords for the current record (key$) against each of the three search keywords that have been input (mkey$). Null keywords are omitted, and the function returns a value TRUE or FALSE, indicating that a match has or has not been made. Line 1920 might only be fully appreciated by trying some examples with pencil and paper. Remember that the element of key_in% is FALSE if the keyword must NOT be matched.)

```
 1850 DEFFNmatch: LOCAL i%,j%,found%,ok%: ok%=TRUE
 1860 FOR i%=1 TO 3
 1870   IF mkey$(i%)="" GOTO1930
 1880   found%=FALSE
 1890   FOR j%=1 TO 3
 1900     IF mkey$(i%)=key$(j%) THEN found%=TRUE
 1910     NEXT
 1920   ok%=ok% AND NOT (found% EOR key_in%(i%))
 1930   NEXT
 1940 =ok%
```

(A regular requirement is to output just the elements of cstart% and cfin% that have been altered, rather than the whole of each array. Function 'cpos' calculates the file address of these two items for category cno%.)

```
 1950 DEFFNcpos(cno%): =5+(cno%-1)*24+14
```

(Function 'rec_len' calculates the length of the current record. Remember that integers rl%, pred%, succ% need five bytes each, and every string two bytes more than the number of characters it contains.)

```
 1960 DEFFNrec_len: LOCAL rl%: rl%=0
 1970 FOR i%=1 TO nf%: rl%=rl%+LENrec$(i%)+2: NEXT
 1980 FOR i%=1 TO 3: rl%=rl%+LENkey$(i%)+2: NEXT
 1990 =rl%+17
```

(Finally, function 'yes' displays its string parameter s$ until a response 'Y/y' or 'N/n' is made by the user - it returns a value TRUE if 'Y/y' is input.
Function 'next_sector' produces the address of the start of the next full sector on from PTR#.
Procedure 'err' ensures that the database file is closed in the event of any untoward happenings.)

```
 2000 DEFFNyes(s$): LOCAL x$
 2010 REPEAT
 2020   PRINT's$;"(Y or N)";: INPUTx$
 2030   UNTIL x$="Y"ORx$="y"ORx$="N"ORx$="n"
 2040 =(x$="Y"ORx$="y")

 2050 DEFFNnext_sector
 2060 =(PTR#fu%DIV256+1)*256

 2070 DEFPROCerr: IF ERR=17 AND fu%<>0 ENDPROC
 2080 PRINT'"Error ";ERR;" in Line ";ERL: REPORT: PRINT
 2090 IF fu%<>0 CLOSE#fu%
 2100 ENDPROC
```

MAKEDB/USEDB - ROOMS FOR IMPROVEMENT

Writing a computer program is rather like making a journey. For short trips there are really only one or two routes that are worth considering. The further afield one travels, however, so the number of ways of reaching one's destination increases - just ask my wife/navigator! With a Case Study of the level just considered, therefore, the first point to make is that the system could have been tackled in many different ways. If the reader has followed things this far, quite a few of those alternatives might have suggested themselves.

Discussing a total rewrite of the Case Study is a bit beyond an 'improvement' though, however much the reader might feel it to be the best policy. Given the approach that has been taken, some likely areas of development are as follows:

MAKEDB

The database file has to be created in the first instance by using the *SAVE command. The disadvantage here is that the file will thus hold quite random bytes to begin with; only some of these will be overwritten with real data and this will make interpretation of a *DUMP output quite difficult. Extend MAKEDB so as to create the database file from within the program (see Case Study 2 for an example), setting its content to zero bytes. The file size should be a user input.

+++++

Category names are automatically truncated to 12 characters. Modify this so as to allow the user to 'rethink' a category name that is found to be too long.

USEDB

Modify PROClist (and PROClist_rec) so that it is possible to use a printer for output of a database listing or of search results. One obvious line of attack would be to add a 'Printer ON/OFF' function to the menu, which sets a flag that the listing procedures check to see whether a VDU2 or VDU3 is called for. A nice refinement, where normal record lengths permit, is to provide the option for a printed record to be output on a single line. This might well be feasible if the <PTR> information etc. is omitted from the output when the program is engaged in a listing or a search.

+++++

The number of keywords associated with each stored record is fixed at three. This means that for a lot of records null keywords are being stored, and for some records the choice of keywords is being constrained. One change could be to allow a variable number of keywords

to be part of any record. For this to be achieved, obvious changes would need to be made to those parts of the program dealing with the input and output of records. Also the keyword array would have to be handled in the same sort of way as the field array, i.e. using a marker (nk%, say) which holds the number of keywords for the record.

+++++

A search is conducted by carrying out a comparison of wanted keywords with those associated with each record. There is no reason why this comparison could not be extended to include the data fields of each record, which could enable a more comprehensive search to be conducted ("all records with these keywords, but not THAT Author"!). The major alterations would be the need to obtain from the user search data for fields as well as keywords, and an extension to PROCmatch.

+++++

Database management systems will normally have some sort of facility for sorting records. That program USEDB does not is an indication of the desire of the author to confine the Case Study strictly to its objective of reflecting the techniques for disk programming covered in the text, rather than a suggestion that a sort capability is not a desirable feature. The addition of some facility for sorting records would certainly be very worthwhile.

It would be possible, using the method of ordering records outlined in Chapter 9, to store records on disk which were sorted on one particular field - 'Author', in all probability. This would increase the complexity of such aspects as record insertion and deletion however, and would increase the time taken to carry out each such task.

An alternative, with a search for instance, is as follows. When a match is found, do not display the record but store its <PTR> value and just the field on which the sort is being made (Author,say). When the search is completed, all of the stored 'Author' names can be sorted. It is then just a matter of using the <PTR> values for each Author in the now sorted sequence to retrieve and display the records. The time taken to conduct a search would increase of course. Since, with any sizeable database, a search is a lengthy operation anyway, it just means that one is now able to drink that cup of tea as well as make it.

+++++

If the USEDB program has been studied in detail, it will have been noticed that the pointer items of the database index, cstart% and cfin%, are written to the file whenever they alter - which is quite often. There is no technical reason why this should occur, and it would be quite simple to remove those parts from the code and just write the whole of the cstart% and cfin% arrays to the file both as part of the QUIT sequence and of PROCerr. A very big warning, however. If for any reason (power failure, say) the program is not left via either of these

two exits, then the database will contain the same index that it started the session with. If records have been added during the session, they will not be reflected in the index and so, unless the index is correctly patched, they will be lost. Perverse as it might seem, therefore, this particular 'improvement' is definitely NOT RECOMMENDED!

CASE STUDY 6
SOAK – A 'disk soak' program for testing a suspect disk drive

SOAK - FUNCTION

This is really quite a simple program which aims to force a suspect disk drive to carry out repeatedly those operations which are likely to lead to the corruption of a floppy disk's formatting.

It will have been realised by now that when any data transfer to disk is initiated, the drive has to seek out the location on the disk at which that data is to be written. For the majority of drives used with the BBC Microcomputer, this operation is preceded by the engagement of the drive motor and the drive's read/write mechanism. (Oh yes - the little red light comes on as well!) These activities have to be synchronised precisely - if a drive's timing is occasionally a fraction out it is almost certain that the format information on the disk will be corrupted, 'Disk Fault' errors being the inevitable consequence.

The Case Study program then endeavours to exercise a drive by asking it to write information to totally random positions in a file. Each write operation is delayed so as to ensure that the drive disengages, and then has to re-engage, the drive motor. A simple screen display shows the position in the file to which a transfer has been made and the number of times that particular location has been used. The program can be left to run for as long as necessary. Experience with a faulty drive has shown that, for a file of 100x30-byte records as used by the sample program, 30 minutes is long enough to demonstrate that a fault has occurred.

SOAK - SAMPLE DIALOGUE

(Since it is the aim of the program to show that a disk has been corrupted by a faulty drive, THE PROGRAM SHOULD BE USED WITH A SCRATCH DISK! Format and verify the disk, to ensure that it is error-free to begin with, and then copy the 'SOAK' program to it.)

```
>CHAIN "SOAK"
```

(The first action of the program is to create a file called DATA. This file, in our example, will contain 100x30-byte fixed-length records comprising the string 'Disk Soak Test' and an integer value, initially zero. As the file is being set up, a series of '.' characters is produced to indicate that the program is still alive and well - this is a useful ploy for any program that, because of a large data transfer, might mistakenly be thought to have got itself into an infinite loop.)

```
Creating Test File
.....................................
.....................................
.....................................
```

(Now the program enters its second phase. It generates a random file address and reads the data items at that position. One of these items is the string 'Disk Soak Test'; the other, an integer value, is incremented by one, so as to count the number of accesses made to that particular address. The two data items are then written back to the disk. Every time this sequence is conducted four lines of output are displayed as shown below.)

```
Access at File Address &AC8
Reading
Disk Soak Test - Access number:1
Writing
```

(A delay occurs between each of these accesses. This is to force the drive to stop and then start up again, since it is this particular aspect of operation which could well be the cause of any trouble.)

```
Access at File Address &30C
Reading
Disk Soak Test - Access number:1
Writing

Access at File Address &3C
Reading
Disk Soak Test - Access number:1
Writing
```

```
Access at File Address &AC8
Reading
Disk Soak Test - Access number:2
Writing

Access at File Address &21C
Reading
Disk Soak Test - Access number:1
Writing

                |
                |
                |
```

(And so on, either until you are satisfied that your disk drive is performing correctly and press any key, or until an error occurs. In this latter case it is most likely that the dreaded

Disk Fault <number> at <Track> <Sector>

will bring the program to an untimely end. If so, repeat the test - preferably obtaining a hard copy of the output - and present the accumulated evidence to your supplier.)

SOAK - METHOD USED

The program creates a standard file of fixed-length records just as described in Chapter 6. Each record comprises 30 bytes made up as follows:

```
String: 16 bytes ("Disk Soak Test" = 14+2)
Not used: 8 bytes
Integer:  5 bytes (Integer value, initially 0)
Not used: 1 byte
```

The program creates a file called DATA into which it writes 100 of these records.

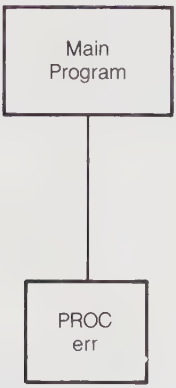
When the second phase of the program runs, a random file address is generated by using the BBC BASIC function RND to produce a random number in the range 1-100. This value is applied with the usual sort of formula to calculate the start address of a record in the file:

```
ptr%=(RND(100)-1)*30
```

The data item is read in the usual way after setting PTR# to this file address. The counter value is incremented by 1 and, after a delay of sufficient length to enable the disk drive to stop, the whole record is written back to the disk file.

If all is well the program continues indefinitely - or at least until any key is pressed. If any fault occurs during one of the data transfers, however, the DFS will cause the program to be terminated, a suitable error message being generated.

(See SOAK - FUNCTION: "This is really quite a simple program ..")



SOAK - FULL PROGRAM LISTING

```
 10 REM***************
 20 REM* CASE STUDY 6 *
 30 REM***************
 40 ON ERROR PROCerr: END

 50 fo%=OPENOUT("DATA")
 60 CLS: PRINT"Creating Test File"'
 70 FOR i%=0 TO 99: PRINT".";
 80   PTR#fo%=i%*30: PRINT#fo%,"Disk Soak Test"
 90   PTR#fo%=PTR#fo%+8: PRINT#fo%,0
100   NEXT
110 CLOSE#fo%

120 fu%=OPENUP("DATA"): REM Use OPENIN with BASIC I
130 @%=4
140 REPEAT
150   ptr%=(RND(100)-1)*30
160   PRINT'"Access at File Address &";~ptr%'"Reading"
170   PTR#fu%=ptr%: INPUT#fu%,s$
180   PTR#fu%=PTR#fu%+8: INPUT#fu%,count%
190   PRINTs$;" - Access number:";count%+1'"Writing"

200   FOR delay=1 TO 5000: NEXT

210   PTR#fu%=ptr%: PRINT#fu%,s$
220   PTR#fu%=PTR#fu%+8: PRINT#fu%,count%+1
230   UNTIL INKEY(0)<>-1: CLOSE#0: PRINT'"Exit from SOAK"'
240 END

250 DEFPROCerr
260 PRINT'"Error ";ERR;" in Line ";ERL': REPORT: PRINT
270 CLOSE#0
280 ENDPROC
```

SOAK - ANNOTATED PROGRAM LISTING

List of Global Variables used:

INTEGERS

count%	Integer value from disk record
fo%	Channel number of file "DATA" during creation
fu%	Channel number of file "DATA" during updating
i%	Counter
ptr%	Value for PTR# to access data items
s$	String part of data (i.e. "Disk Soak Test")

REAL

delay	Counter for delay between disk accesses

```
   10 REM****************
   20 REM* CASE STUDY 6 *
   30 REM****************
   40 ON ERROR PROCerr: END
```

(The disk file DATA is created. It is constructed to contain 100x30 byte records comprising a string and an integer value, with intermediate bytes remaining unused. The integer value is set to zero initially. For every record that is created, a '.' is output to the display to indicate to the user that the program is operating correctly.)

```
   50 fo%=OPENOUT("DATA")
   60 CLS: PRINT"Creating Test File"'
   70 FOR i%=0 TO 99: PRINT".";
   80   PTR#fo%=i%*30: PRINT#fo%,"Disk Soak Test"
   90   PTR#fo%=PTR#fo%+8: PRINT#fo%,0
  100   NEXT
  110 CLOSE#fo%
```

(Having been closed for output, the file is now re-opened for updating. A record is selected at random by using the RND function to produce a value in the range 1-100. This value is employed to calculate the PTR# address in the file. The record is input and messages, which incorporate that data just read, are sent to the display to indicate progress.)

```
  120 fu%=OPENUP("DATA"): REM Use OPENIN with BASIC I
  130 @%=4
  140 REPEAT
  150   ptr%=(RND(100)-1)*30
  160   PRINT'"Access at File Address &";~ptr%'"Reading"
  170   PTR#fu%=ptr%: INPUT#fu%,s$
  180   PTR#fu%=PTR#fu%+8: INPUT#fu%,count%
  190   PRINTs$;" - Access number:";count%+1'"Writing"
```

(To ensure that the disk drive mechanism has to stop and start again a delay loop is inserted at this point. Note that the loop limit would need to be a lot higher than 5000 if 'delay' was an INTEGER rather than a REAL variable, since integer arithmetic is performed much faster than floating point arithmetic. As it is, the upper limit may well need to be adjusted for individual disk drives.)

```
  200   FOR delay=1 TO 5000: NEXT
```

(The record is now written back to its position on disk, with the integer part increased by one. This read/write cycle will be repeated either until the program fails - through some disk fault one assumes - or any key is pressed.)

```
  210   PTR#fu%=ptr%: PRINT#fu%,s$
  220   PTR#fu%=PTR#fu%+8: PRINT#fu%,count%+1
  230   UNTIL INKEY(0)<>-1: CLOSE#0: PRINT'"Exit from SOAK"'
  240 END

  250 DEFPROCerr
  260 PRINT'"Error ";ERR;" in Line ";ERL': REPORT: PRINT
  270 CLOSE#0
  280 ENDPROC
```

SOAK - ROOMS FOR IMPROVEMENT

Bearing in mind that the object of the program is to exercise the disk drive as much as possible, the following are not so much 'improvements' but rather changes in the parameters with which the program operates.

The file size is defined within the program. So too is the size of the record written, and its format. Any or all of these elements could be changed in some way - possibly they could all be input by the user before the program starts a test run. Making file DATA much larger would certainly lengthen the time taken to find any particular sector - but would also lessen the chances of any particular sector being used. Changes to the size of the record should be such that it is possible for a record either to finish in the last byte of a file sector, or to cross a sector boundary.

+++++

Line 5000 sets a delay which is designed to cause the disk drive to stop and then have to start again. A little variant might be introduced here:

```
200   FOR delay=1 TO RND(7000): NEXT
```

would make this a variable delay. In this way the drive would stay on more often than not, but would still need to disengage and re-engage at regular intervals.

CASE STUDY 7
MAYDAY — A program for recovering sections of a corrupt file

MAYDAY - FUNCTION

When a disk is corrupted it is most unlikely that the whole or even the majority of the disk space becomes unreadable. Rather it is usually the case that one or two sectors cannot be read, the remainder of the disk still being intact. Of course if those unreadable sectors happen to be either of sectors 0 and 1 on track 0 of the disk - i.e. those sectors containing the disk catalogue - then the situation is pretty hopeless. But if this is not the case, it may well be possible to recover a large proportion of the file which has become corrupted.

Program MAYDAY illustrates the use of a simple technique which may be employed to good effect here. Basically it involves nothing more than the transfer of readable sections of a corrupted file to a new file. Any section is specified to the program by means of 'start' and 'end' addresses and, by calling the program a number of times, a new file can be constructed which contains all of the readable sectors of a corrupted file. Moreover, if backup copies of the blighted file exist, MAYDAY can also be used to copy appropriate sectors from the backup copy to fill in the gaps of the new file. (Obviously the feasibility of this latter step will be heavily dependent upon the nature and organisation of the corrupted file - a linked organisation, for instance, would present quite acute problems.) Alternatively, the missing sectors can be 'patched' in by using the techniques outlined in Chapter 7.

(This Case Study program, as with all of the others, has been written under the author-imposed constraint of not using Assembly Language. It should be pointed out, however, that it is possible to

achieve much better success at recovering files by using the facilities provided by the BBC Operating System for assembly-level programming.)

It is assumed that, before the program is used, an investigation will have been carried out to determine the exact state of the corrupted file. A complete picture should be available of which sectors of the file are readable, and which are not. The simplest way of obtaining such a picture is by using a little program of the following form (where <filename> and <start> are replaced as required):

```
10 ON ERROR GOTO 60
20 fi%=OPENIN("<filename>")
30 FOR address%=<start> TO EXT#fi% STEP &100
40 PTR#fi%=address%: byte%=BGET#fi%
50 PRINT"Bytes &";~address%;" to &";~(address%+&FF);" OK"
60 NEXT
70 CLOSE#fi%: END
```

With <start> initially at 0, the program will attempt to read the first byte in each successive sector of the file, printing out a message when it has done this. When an unreadable sector is encountered the program will fail with the 'Disk fault' message, and the start address of the corrupt sector can be noted. The program is now run again , with <start> set to the opening address of the sector beyond the corrupt one. This process continues until the whole file has been examined, and a full record of bad sectors built up.

MAYDAY - SAMPLE DIALOGUE

(By way of an example, assume that a file called DATA occupies four sectors, of which the second is corrupt. The following dialogue would be produced in copying the readable sectors - 1,3 and 4 - to another file NEWDATA.)

```
>CHAIN "MAYDAY"

Name of SOURCE file?DATA

Name of DESTINATION file?NEWDATA

Start from: &0          (i.e. Sector 1 of file DATA)
End at: &FF

Load DATA disk and hit any key X <----(X='Any key')
Reading
....
Load NEWDATA disk and hit any key X
Transferring &0
....
to &FF
Exit from MAYDAY

>RUN                (Run again to copy sectors 3 and 4)

Name of SOURCE file?DATA

Name of DESTINATION file?NEWDATA

Start from: &200    (i.e. Sectors 3,4 of file DATA)
End at: &3FF

Load DATA disk and hit any key X
Reading
........
Load NEWDATA disk and hit any key X
Transferring &200
.........
to &3FF
Exit from MAYDAY

>
```

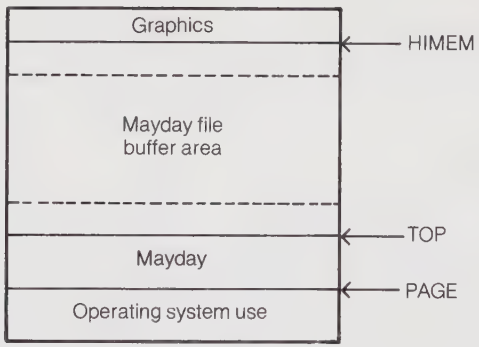
(The copying of the good sectors is now complete. From this point it is necessary to reconstruct the missing sectors, either by using MAYDAY again to transfer these sectors from a backup file, or by 'patching' data bytes into the sectors.)

MAYDAY - METHOD USED

The method used is unavoidably determined by the nature of the problem. Program MAYDAY has to cope with files of any content, and thus it must read the source file on a byte-by-byte basis. Moreover,the destination file is hardly likely to be wanted on the same disk as the source file, which, after all, is corrupted! Copying a file section must therefore be carried out by a buffering process of reading the file section into RAM, and then sending it from RAM to the destination file. (This assumes that a single-disk drive is being used; the restriction would not apply with a dual-disk system.)

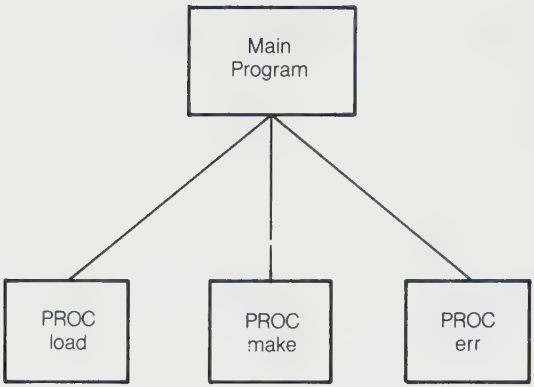
This need to implement a file buffer poses two problems which have to be overcome. The first is, whereabouts in RAM should the buffer be positioned? and the second, what if the buffer isn't big enough for the file section to be copied?

As to the first problem, a look at the BBC Micro store map will help:



When MAYDAY is in store, the area PAGE to TOP is occupied by the program itself. The locations from HIMEM to the physical top of RAM are used for the graphics display of whatever mode is in effect. In theory, therefore, the area from TOP to HIMEM is available for use as our file buffer. But in practice this is not the case. Any variables which are used by MAYDAY are placed immediately above TOP. Any PROC statements cause the system stack to be lengthened, and this area resides immediately below HIMEM! Nevertheless, the general principle of using this area is sound, so long as adequate tolerances are used. MAYDAY sets its file buffer area, therefore, as being that section of RAM from TOP+1000 to HIMEM-100.

As to the second problem, large disk files can occupy rather more space than is available even in this area of RAM. Also if MAYDAY is run in one of Modes 0-6, the buffer area is reduced as HIMEM is moved to account for the additional RAM needed for graphics. A necessary feature of the program is for it to keep track of how much of the specified file section it has transferred at a time, and if necessary perform more than one iteration in order to carry out the complete copying operation.


Main
Program
PROC
load
PROC
make
PROC
err

MAYDAY - FULL PROGRAM LISTING

```
 10 REM****************
 20 REM* CASE STUDY 7 *
 30 REM****************
 40 ON ERROR PROCerr: END

 50 INPUT'"Name of SOURCE file",fi$
 60 INPUT'"Name of DESTINATION file",fu$: PROCmake(fu$)
 70 INPUT'"Start from: &" x$: start%=EVAL("&"+x$)
 80 INPUT"End at: &" x$: end%=EVAL("&"+x$)

 90 REPEAT
100   PROCload(fi$): fi%=OPENIN(fi$)
110   IF end%>EXT#fi% THEN end%=EXT#fi%
120   PTR#fi%=start%
130   sad%=TOP+1000: count%=0: PRINT'"Reading"

140   REPEAT: IF count% MOD 64=0 PRINT".";
150     byte%=BGET#fi%: sad%?count%=byte%: count%=count%+1
160     UNTIL EOF#fi% OR start%+count%>end%
                       OR sad%+count%=HIMEM-100
170   CLOSE#fi%

180   PROCload(fu$)
190   fu%=OPENUP(fu$): REM Use OPENIN with BASIC I
200   PTR#fu%=start%: PRINT'"Transferring &";~start%
210   FOR out%=0 TO count%-1: IF out% MOD 64=0 PRINT".";
220     byte%=sad%?out%: BPUT#fu%,byte%
230     NEXT: CLOSE#fu%
240   start%=start%+count%
250   UNTIL start%>end%: PRINT" to &";~end%;"'"Exit from MAYDAY"'
260 END

270 DEFPROCload(s$): LOCAL key%
280 PRINT'"Load ";s$;" disk and hit any key";
290 key%=GET: PRINT
300 ENDPROC

310 DEFPROCmake(s$)
320 fu%=OPENIN(s$)
330 IF fu%=0 THEN fu%=OPENOUT(s$)
340 CLOSE#fu%
350 ENDPROC

310 DEFPROCerr
320 PRINT'"Error ";ERR;" in Line ";ERL': REPORT: PRINT
330 CLOSE#0
340 ENDPROC
```

MAYDAY - ANNOTATED PROGRAM LISTING

List of Global Variables used:

STRINGS

fi$	Name of source file
fu$	Name of destination file
x$	Temporary string for start/end addresses

INTEGERS

byte%	Holds byte after input/before output
count%	Counter for reading of source file bytes
end%	End address for source file copy
fi%	Channel number of source file
fu%	Channel number of destination file
out%	Counter for writing destination file bytes
sad%	Store buffer address
start%	Start address for source file copy

```
   10 REM****************
   20 REM* CASE STUDY 7 *
   30 REM****************
   40 ON ERROR PROCerr: END
```

(The names of the source and destination files are input, the latter being created if it does not already exist. Then the program inputs the hexadecimal start and end addresses of the section which is to be transferred from the source file to the destination file.)

```
   50 INPUT'"Name of SOURCE file",fi$
   60 INPUT'"Name of DESTINATION file",fu$: PROCmake(fu$)
   70 INPUT'"Start from: &" x$: start%=EVAL("&"+x$)
   80 INPUT"End at: &" x$: end%=EVAL("&"+x$)
```

(The source file is opened for input, and the value of end% adjusted if it is larger than the known extent for the file. PTR# is initialised to the start address for input. Variable sad% is set to a value which is decently above the area used by the program and its data. All bytes which are input will be written to a position in store which is relative to this start address.)

```
   90 REPEAT
  100   PROCload(fi$): fi%=OPENIN(fi$)
  110   IF end%>EXT#fi% THEN end%=EXT#fi%
  120   PTR#fi%=start%
  130   sad%=TOP+1000: count%=0: PRINT'"Reading"
```

(Bytes are input from the source file to RAM. They are placed at a position given by the start address, sad%, plus an offset, count%, using the '?' indirection operator. There are three possible ways in which

this loop can terminate: either the end of the source file is reached, or the end of the specified file section is reached, or the program runs out of available RAM. Whatever the case, the file is closed.)

```
140   REPEAT: IF count% MOD 64=0 PRINT"."
150     byte%=BGET#fi%: sad%?count%=byte%: count%=count%+1
160     UNTIL EOF#fi% OR start%+count%>end%
                        OR sad%+count%=HIMEM-100
170   CLOSE#fi%
```

(With a file section now held in RAM at address sad% onwards, the destination file is opened for updating. The sequence of events is the converse of what has just happened. Bytes are output from an address given by sad% plus an offset, out%, until the whole of the stored section has been copied across. The destination file is then closed.)

```
180   PROCload(fu$)
190   fu%=OPENUP(fu$): REM Use OPENIN with BASIC I
200   PTR#fu%=start%: PRINT'"Transferring &";~start%
210   FOR out%=0 TO count%-1: IF out% MOD 64=0 PRINT".";
220     byte%=sad%?out%: BPUT#fu%,byte%
230     NEXT: CLOSE#fu%
```

(It is now necessary to determine exactly what was the terminating condition for the source file input. The concern is to discover whether insufficient RAM was available to transfer the whole of the specified file section. If so, then the input/output cycle has to be repeated for the next file section. In the event, the test is very simple: start% is added to the number of bytes that have been copied, and if this is not larger than the specified end address, end%, then the loop is repeated with a new value for start%.)

```
240   start%=start%+count%
250   UNTIL start%>end%: PRINT" to &";~end%;"'"Exit from MAYDAY"'
260 END
```

(Procedure 'load' displays a message which aims to prompt the user into checking that the correct disk is loaded. After so checking, the user responds by pressing any key which happens to take his fancy.)

```
270 DEFPROCload(s$): LOCAL key%
280 PRINT'"Load ";s$;" disk and hit any key";
290 key%=GET: PRINT
300 ENDPROC
```

```
(Procedure 'make' creates the destination file if this file does not
already exist. Note that this must be capable of being extended, since
it will be created with a zero extent!)

  310 DEFPROCmake(s$)
  320 fu%=OPENIN(s$)
  330 IF fu%=0 THEN fu%=OPENOUT(s$)
  340 CLOSE#fu%
  350 ENDPROC

(Procedure 'err' .. Oh, you must know by now!)

  310 DEFPROCerr
  320 PRINT'"Error ";ERR;" in Line ";ERL': REPORT: PRINT
  330 CLOSE#0
  340 ENDPROC
```

MAYDAY - ROOMS FOR IMPROVEMENT

Only one suggestion as far as MAYDAY is concerned, and this is more of a development than an 'improvement'.

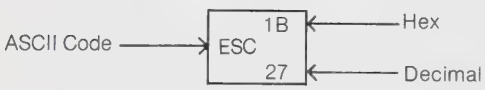
The program works very slowly because it is written in BASIC, and would certainly benefit from being translated into machine code. However, taken as it stands, the program is basically a file-to-file transfer utility. It could be developed, therefore, in this sort of direction. For instance, the facility to append one file on to the end of another should be perfectly possible. By allowing the user to specify start and end addresses for both source and destination files separately a utility for moving file blocks about could also be achieved.

Of course, since the information passes through RAM on its way from one file to another, it could be caused to undergo some sort of metamorphosis. If the source file was a BASIC program for instance, a piece of intermediate processing could remove REM statements before sending the data on to its destination.

The limit, as they say, is your own imagination!

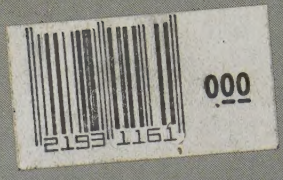
APPENDIX A:
ASCII Codes

0 NUL 0	10 DLE 16	20 SP 32	30 0 48	40 @ 64	50 P 80	60 ` 96	70 p 112
1 SOH 1	11 DC1 17	21 ! 33	31 1 49	41 A 65	51 Q 81	61 a 97	71 q 113
2 STX 2	12 DC2 18	22 " 34	32 2 50	42 B 66	52 R 82	62 b 98	72 r 114
3 ETX 3	13 DC3 19	23 # 35	33 3 51	43 C 67	53 S 83	63 c 99	73 s 115
4 EOT 4	14 DC4 20	24 $ 36	34 4 52	44 D 68	54 T 84	64 d 100	74 t 116
5 ENQ 5	15 NAK 21	25 % 37	35 5 53	45 E 69	55 U 85	65 e 101	75 u 117
6 ACK 6	16 SYN 22	26 & 38	36 6 54	46 F 70	56 V 86	66 f 102	76 v 118
7 BEL 7	17 ETB 23	27 ' 39	37 7 55	47 G 71	57 W 87	67 g 103	77 w 119
8 BS 8	18 CAN 24	28 (40	38 8 56	48 H 72	58 X 88	68 h 104	78 x 120
9 HT 9	19 EM 25	29) 41	39 9 57	49 I 73	59 Y 89	69 i 105	79 y 121
A LF 10	1A SUB 26	2A * 42	3A : 58	4A J 74	5A Z 90	6A j 106	7A z 122
B VT 11	1B ESC 27	2B + 43	3B ; 59	4B K 75	5B [91	6B k 107	7B { 123
C FF 12	1C FS 28	2C , 44	3C < 60	4C L 76	5C \ 92	6C l 108	7C | 124
D CR 13	1D GS 29	2D - 45	3D = 61	4D M 77	5D] 93	6D m 109	7D } 125
E SO 14	1E RS 30	2E . 46	3E > 62	4E N 78	5E ^ 94	6E n 110	7E ~ 126
F SI 15	1F US 31	2F / 47	3F ? 63	4F O 79	5F _ 95	6F o 111	7F DEL 127



INDEX

PRENTICE
HALL
INTERNATIONAL
PERSONAL
COMPUTER
BOOK



A full, clear, step-by-step introduction to disk systems and data handling with the BBC Microcomputer.

The book introduces the Disk Filing System Commands in a logical order, explaining in detail how each is used. It develops sound 'housekeeping' techniques essential for all disk users. It presents in depth the techniques of writing BASIC programs which use serial or random access disk files. A comprehensive section on disk error messages is included.

Seven fully documented case studies provide practical illustrations of these techniques.

Suitable for all BBC Micros with Acorn or Acorn-compatible disk operating systems – covers BASIC I and BASIC II.

No assembly language is used.

A disk containing the seven case-study programs is available separately. The disk includes a full 65000-byte database, containing details of articles on the BBC Microcomputer.

ISBN 0-13-215930-9

£7.95

ISBN 0-13-215930-9
9 780132 159302