SLOGGER

Slogger Computers
107 Richmond Road
Gillingham
KENT ME7 1LX

S E D F S
Slogger Electron Disk Filing System
User Guide

For the Acorn Electron

CONTENTS

INTRODUCTION

The purpose of this product and its documentation is to provide the user with the means to exploit the Slogger Electron Disk System to the full. Step by step instruction is given with practical examples on all aspects of the interface lading from the most basic functions to advanced techniques. Not only is information given on the operation of SEDFS and its commands but a wider view is explored of SEDFS' place in the world of programming and data processing. File operations are discussed in depth leading to random access.

Talking to SEDFS

The user is able to communicate directly with SEDFS by typing a special form of command which is prefixed by a *. For instance:

```
*CAT
```

This type of command is called a *command.

There are other commands which will also initiate a response from SEDFS which do not have the *. For instance:

```
LOAD
```

Although it is not apparent to the user, these commands do not go directly to SEDFS but are first passed through BASIC.

Syntax of commands

All SEDFS commands consist of a single word preceded by a *. The parameters follow after a space. For example:

```
*STAT <drive>
```

*STAT is the command name followed by a single parameter in the form of a drive number. Sometimes more than one parameter is given, in which case they are separated by a space. For example:

```
*LOAD <fsp> (<address>)
```

Parameters which are enclosed in brackets are optional. Commands and their associated parameters are explained fully in the main body of the text.

Many commands have abbreviated forms which may be used instead of the full length version and these two forms are presented side by side in the text.

CHAPTER ONE

DRIVES AND DISKS

Floppy disks

Disks are made of very thin but very tough plastic and are coated with a magnetic substance called ferric oxide. This is a substance which can be magnetised in such a way that data can be recorded on it as a pattern of magnetic dots. In this sense a disk is like a cassette tape. Information is recorded on it as magnetic dots which represent the 0s and 1s of binary numbers.

Disks come in a variety of sizes, but 5.25" diameter is the one most commonly used with the BBC Micro and Acorn Electron. This is probably the wisest buy for compatibility with most software currently on the market.

The long slot at the bottom of the casing exposes the shiny surface on which data is recorded. Do **NOT** touch this area because continued mishandling in this matter will eventually ruin the disk.

Write protect notch

The slot in one of the sides of the disk case is the write protect notch. In its open state, the disk can be used normally but if you want to protect all the files contained on the disk then stick a cover over the notch. There should be a number of these supplied when you buy your disks and they prevent the disk being written to. This means that data can still be read as normal but cannot be altered in any way. If a write operation is attempted with the disk protect tab in place then the screen message:

```
Disk read only
```

will be displayed. In this state the data is completely safe and there is no way that any file can be changed.

Tracks

The read/write head is the part of the disk drive which places (writes) magnetic dots on the disk surface and is capable of sensing (reading) magnetic dots already recorded on the disk. The head travels across the surface of the disk in small, measurable units called steps. Movement is controlled by a stepper motor disk controller. You may be able to hear this in action due to the sometimes noisy clicking which occurs when the head moves across the disk. At each position the head had access to a track of data, a track being a path or ring on the disk along which data is stored. On receiving a pulse from the floppy disk controller, the stepper motor moves the head backwards or forwards to an adjacent track. The floppy disk controller always knows the exact location of the read/write head by counting the number of pulses it gives out. i.e. one pulse, one track. When the drive is switched on, the track count is initialised by moving the head towards track 0 where a switch is tripped to indicate when the head has reached track 0.

Tracks - 40 or 80

There are three types of drive:

1. 40 track
2. 80 track
3. 40/80 switchable

The difference between them is that on a 40 track drive, the steps are twice as long as those on an 80 track drive and the read/write head itself will be wider. It is impossible for a 40 track drive to read an 80 track disk but the reverse is possible because an 80 track drive can be made to "double step" and thereby simulate the operation of a 40 track drive. This is achieved by performing two steps for one control pulse. Obviously 80 tracks can contain twice as much data as 40 tracks resulting in a much more economical use of disk space.

Sectors

Each track is divided into a number of manageable units called sectors, each containing 256 bytes of data. When inserted into the drive, the centre of the disk is gripped and the disk is spun continuously whilst in operation. Most disks have a reinforced hub to help prolong their lives. As the disk is spinning any part of the track can be read as it goes past the read/write head and the floppy disk controller knows which sector it is reading by making reference to the index hole. Referring back to illustration 1.1, the index hole is a window in the casing near the centre of the disk. Take a disk and, by carefully opening your fingers inside the hub, turn the disk inside the casing. Before long a small hole should appear at the window. When the disk is in the drive a light shining through this hole activates a sensor sending a pulse to the floppy disk controller indicating that the disk is positioned at the beginning of a track (sector 0). From here it is simply a matter of keeping a count of the sectors.

Drive

The drive is the term as used by Acorn to identify a side of the disk. The term is confusing as it can be mistaken to mean the drive unit itself. However, in most instances its meaning will be obvious by the context. Different drive units are capable of accessing different combinations of disk sides, as outlined below:

Drive Unit	Disk Sides Accessible (Drive)
Single Drive Single Sided	0
Single Drive Double Sided	0 and 2
Double Drive Single Sided	0 and 1
Double drive Double Sided	0, 1, 2 and 3

Catalogue

This is a filing system within a filing system which contains information on all data which exists on the disk including the filenames and the length and location of files. It exists at the beginning of track 0 and is used in every read or write operation.

By means of tracks and sectors, any part of the disk can be accessed very quickly and part of the information held in the catalogue gives the file's position on the disk in terms of sectors. In normal use, tracks and sectors are dealt with automatically by SEDFS and the user may not manipulate them directly. However, knowledge of the subject is useful as some SEDFS facilities make use of tracks and sectors to organise the layout of the disk.

Buying disks

Disks usually come in boxes of ten with varying specifications and prices. First make sure you have the right size for your drive. Then look at the top left hand corner of the disk where the certified density and tracks per inch are specified. Disks are all made the same way but are graded according to their final quality. It is possible to format any disk to any specification but the certified standard is guaranteed by the manufacturer and you may find that a disk formatted above the certified standard is unreliable. The range of specifications is as follows:

Double or Single density
Double or Single sided
40 or 80 Track (48tpi or 96tpi)

The track number may be expressed in tpi (tracks per inch). You may be confused as to why 48 applies to 40 tracks and 96 applies to 80 tracks. The reason for this is that the density of tracks is expressed in terms of an inch but during operations the read/write head traverses a little less than an inch. The inch is a convenient unit to use and you could say that if the read/write head did actually cover an inch then either 48 tpi or 96 tpi could be recorded on the disk.

The other term you will see on a disk is "soft sectored". This refers to the fact that sectors are laid down by the formatting procedure (See next section on Formatting) and the index hole identifies the first sector. Thus the sectors are formed by SOFTware. On the other hand, a "hard sectored" disk which is not used by the Acorn Electron has many holes in the revolving surface which, when lined up with the index hole, identify each sector. Thus the sectors are formed by HARDware.

Formatting

Before we can do anything at all with a disk it must be formatted. A new disk is completely blank in the same way that a new cassette tape is. However, there are crucial differences. A new cassette tape can be written to immediately without any preparation and data is written sequentially. That is, data can begin at any point on the tape and is recorded one byte after the next until all the data has been recorded. This continuous block of data is termed a "file". The next file can b recorded straight after the first one, or disastrously, even on top of it if we are not careful. Reading a file requires the same type of process. The tape must be wound manually to the beginning of the file and the computer set to read the required file. This, in simple terms, constitutes a cassette filing system.

A disk filing system is much more complex than this with the ability to go to any point on the disk automatically in a very short space of time (random access). This ability to access any part of the disk is made possible by recording the catalogue and the positions of the tracks and sectors before attempting to record any data onto it. Once this process is complete, data can be written to specific parts of the disk and its existence recorded in the catalogue. When the disk is read, the specified file is searched for in the catalogue, the location of the data is found and the read/write head is moved to the correct track. As the disk begins to turn, the correct sector is located so that operations can begin.

Well, how do we go about laying down the track, sector and catalogue structure onto a new disk? We use for FORMAT program which is contained within SEDFS. This program can be invoked by typing:

```
*FORMAT
```

and pressing <RETURN>. You will be presented with the following screen:

```
            D I S K   F O R M A T T E R




      Drive (0-3)
```

Type "0" and press <RETURN>. Now the following will appear:

```
      0=40, 1-80 tracks
```

Type "0", which will select the format routine for a 40 Track and press <RETURN> and the prompt to initiate the format is given.

```
      Press F to start
```

After the format has been specified, a safety check is made to ensure that you really do want to format. This is because if a disk already containing files is reformatted, all the existing files will be completely and irretrievably obliterated. If you are sure you wish to continue with the format procedure then press "F".

As each track is formatted, its number will be displayed on the screen. If an error occurs, such as it would with a faulty disk, the routine will stop at the offending track and the message "Verify Error" will be displayed. This can occur when a disk has been damaged in some way and the only solution is, unfortunately, to throw away the offending article and use another one.

Disk Errors

While we are discussing the layout (configuration) of disks, there is a way of checking them for faults which could be caused by such misuse as bending, dirt etc. A faulty write operation can also make the disk unreadable. We hope this never occurs but it could be produced by a machine fault, a drive fault or simply pressing <BREAK> during a write operation. The command:

```
*VERIFY <drive>
```

will do the necessary checks. If the drive is not specified, the current drive will be verified. The program prompts for any key to be pressed upon which each sector is individually checked. If any track is faulty the program will stop and "Verify Error" is displayed. Sometimes it is possible to correct an error by using a disk editor, but this requires some technical knowledge. If you do not possess a disk editor or are not sure of how to use one, the easiest solution is to transfer all the useable files to another disk (this will be dealt with later) and re-format the disk. If the disk has been physically damaged, throw it away.

CHAPTER TWO

DISK FILES AND THE CATALOGUE

What is a file?

The dictionary defines a file as a collection of papers held in an orderly manner, and this is very close to the definition we use with the computer. A file in computer terms is a collection of logically related records which is treated as a complete unit and given an individual name. By "logically related" we mean having a common link or purpose as, say, a list of names would have.

Filename

This is a name we give to a file. Although it can contain any type of character, it must not be more than seven characters in length. Filenames are of your own invention and to aid in locating files, it is advisable to keep the names meaningful. For instance, an accounting program could be called "ACCOUNT". This may seem a petty point to mention but it can in practice save a lot of time and frustration in the future when, perhaps several months later, you are trying to locate a program on a disk containing many files (the names of which are not now so familiar!).

Types of file

There are different types of file for different uses. A list of them is given below and deeper explanations are given later where appropriate.

Broadly speaking there are five types of file:

1. The ASCII or text file
2. The BASIC program file
3. The BASIC data file
4. The Binary program file
5. The Binary data file

To illustrate basic file manipulation, we will make a BASIC program file. Type in example 2.1

Example 2.1

```
  5 REM ASCII DISPLAY
 10 REPEAT
 20 INPUT"PRESS A KEY",A$
 30 PRINT ~ASC(A$)
 40 UNTIL FALSE
```

We can record this program from memory onto a disk by simply typing:

```
SAVE"PROG1"
```

and pressing <RETURN>. The program will be saved in the form of a BASIC program file. The word "SAVE" is the instruction to write the program to disk and is followed by the filename in inverted commas. Just to prove that your file is really there, remove the disk from

the disk drive and switch off the computer. Switch on again, return the disk to the drive and type:

```
LOAD"PROG1"
```

This command reads the disk and copies the file into the computer's memory. LIST to make sure everything is alright and then run the program.

Incidentally, it is important to leave the disk out of the drive when switching the computer on or off because some makes of drive may corrupt the disk during the process.

The Catalogue

Now that we have some idea of how to format the disk and save and load programs, we can look a little deeper into how SEDFS organises a maximum of 31 files per disk side. Files are organised on the disk by the catalogue. Each catalogue has a predefined structure and can accommodate up to 31 files.

First format a new disk on drive 0 with 40 tracks. Then perform the command:

```
*CAT (<drive number>) or *. (<drive number>)
```

to display the catalogue. The catalogue occupies sectors 0 and 1 of track 0 and contains information on the filenames, addresses and lengths of files contained on the disk.

After typing *CAT, the screen will look like this:

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

No file
>
```

As expected, we are informed that the disk is empty by the "No file" message at the bottom of the screen. Of the other information, so far we have explained drives but the rest requires some clarification.

(00)

This tells us how many times the disk has been written to. Every time we save a file, this number is incremented by 1.

Drive 0

This is self-explanatory and indicates which side of the disk we are looking at.

```
*DRIVE <drive number> or *DR. <drive number>
```

will change the current drive but as we have only formatted side 0, any attempt to read a different drive will result in the error message:

```
Disk not formatted
```

Directory :0.$

The directory is a method of indexing disk files by prefixing them with a letter or the $ sign. As already stated, filenames can be up to seven letters long. For instance, "PROG1" will do very well and when saved will be assigned to the current directory which in this case is $ as indicated by the screen message. The directory sign is part of the filename so the filename in this instance is actually "$.PROG1". As the file was saved under the current directory, the $ sign is not displayed. If, however, we were to save under a different directory e.g. directory A, by using the command:

```
SAVE "A.PROG1"
```

then the directory will be displayed. This is a convenient method of grouping similar programs under a common heading. By using different directories we can save files with the same name on the same catalogue. Don't try this with files in the same directory unless you want to change a file as the one already on the disk will be over-written by the new file.

The :0 in the screen message refers to the current drive. The default drive and directory are 0 and $ respectively. This brings us conveniently to the full file specification (fsp). This is as follows:

```
:<drive number>.<directory>.<filename>
```

or, for the file "PROG1" on drive 0, directory $:

```
:0.$.PROG1
```

The current directory can be changed with:

```
*DIR <directory>
```

Change the directory to A with:

```
*DIR A
```

and perform another *CAT. Notice that directory $ is now displayed with the filename and directory A is shown at the head of the catalogue. The drive can be changed with:

```
*DIR :<drive>
```

and the complete directory is modified with:

```
*DIR :<drive>.<directory>
```

Library :0.$

This is set to drive 0, directory $, (the default value) and is useful in loading a file without changing drive or directory. For example, to load a BASIC program "PROG1" which comes under the library specification, simply type:

```
*PROG1
```

If this program is in machine code it will be run as well. The library can be changed with:

```
*LIB :<drive>.<directory>
```

To change to drive 1, directory C, type:

```
*LIB :1.C
```

Now any file can be loaded from drive 1, directory C by preceding its filename with a *. It is not quite legal to do this with BASIC files, but it does work.

Option 0 (off)

The final message displayed by *CAT is the Option specification. This refers to start up action which will be discussed in chapter four.

Number of Files and Disk Memory

40 Track

In 40 track format, 100K bytes are available to store files under each catalogue. There is one catalogue for each side of the disk. Each track is divided into ten sectors. There are 256 bytes per sector, giving a total of 2560 bytes per track. This means that a total of 102,400 bytes (characters) can be stored. This is the actual total and results from the fact that 1K is 1024 and not 1000 as you would at first expect. We can see the total bytes capacity and the unused capacity with the command:

```
*STAT (<drive number>) or *ST. (<drive number>)
```

With a newly formatted disk in 40 track single density, this will display:

```
Drive 0    Disk size      15E       100K
           Disk unused    15E       100K
```

Obviously, when a program is saved onto the disk the volume unused is reduced. The amount of volume unused is useful to know when you want to save a large file. Even if you have fewer than 31 files in the catalogue but the volume unused is too small the file will not be saved and the message:

```
Disk full
```

will be displayed.

80 Track

Now reformat the disk (*FORMAT), (or, if you have a double-sided drive, format drive 2), this time selecting 80 tracks if you have an 80 Track drive. When complete, perform the command:

```
*STAT 2
```

This will give the capacity of drive 2 and should display:

```
Drive 2    Disk size      31E       200K
           Disk unused    31E       200K
```

80 tracks have been put onto the same disk area as 40 tracks were with the previous format. The result is double the capacity although there is still only one catalogue with a maximum of 31 files. The advantage with 80 tracks is that we can, however, have much larger files.

Play with these two methods of formatting until you understand them fully. If you are using drives 0 and 2, it is possible to read the

catalogue and capacity from the default drive by including the drive in the command. For example:

```
*CAT. 2 ..... *STAT 2
```

Files can be saved or loaded by using the full filename (e.g. :2.$.PROG1) or to do it the easy way, there is no need to specify the drive if the current drive is the one required. Change the current drive with:

```
*DRIVE <drive number> or *DR. <drive number>
```

So to change to drive 2, use:

```
*DR. 2
```

It is possible to reset to the default drive (0) by <CTRL><BREAK>. This is the easiest way but be careful if you have a program resident in memory because if you wish to keep it, type OLD before typing anything else after <CTRL><BREAK>.

Altering Catalogue Files

The commands we are about to discuss change the catalogue entries and, although not actually working on files themselves, are extremely convenient, enabling us to keep disks in good order.

Renaming Files

Individual files can be renamed with:

```
*RENAME <old fsp> <new fsp> or *REN. <old fsp> <new fsp>
```

Imagine you have saved a file called "HAMSTER" but later changed it to refer to cats. It would then be desirable to rename the file.

```
*RENAME HAMSTER CATS
```

will change the name from "HAMSTER" to "CATS". If the directory is included in the command then we can change directories with or without changing the filename. For example:

```
*RENAME A.HAMSTER B.HAMSTER
```

Naming the drive

The disk itself can also be given a name and this facility is particularly useful where a disk or volume is used for files with a common purpose. The title can be up to 12 characters long and appears at the top of the catalogue after performing *CAT. In double density each volume can be given an individual title. The command is:

```
*TITLE <diskname> or *TI. <diskname>
```

For example, to give the title "SEDFS" to the disk in the current drive, type:

```
*TITLE SEDFS
```

The title must be in inverted commas if a space is to be included in it:

```
*TITLE "SEDFS DISK"
```

To change the title of a disk, use the same command with a new title and the old title will be overwritten.

How to Delete Individual Files

So far we have shown how to load and save files using various formats and how to use names to organise them into meaningful categories within a disk or volume. Having learnt how to put files onto a disk, you now need to be able to take them off. Often when developing programs, you will end up with several versions of a file, all in different stages of development, and to economise on disk space you will need to delete the old files. The simplest and safest command to do this with is:

*DELETE <fsp> or *DE. <fsp>

This will remove the specified file from the catalogue and although the data is still physically present on the disk, SEDFS will not recognise its existence. (It will eventually be obscured when another program is saved over it.) With a disk editor, it is still possible to reclaim the file by reinstating the name and attributes in the catalogue but this is a difficult task and, in this case, prevention is definitely better than the cure. Be warned!

Compacting the Disk

Deleting a file will leave a "hole" on the disk. That is, if the deleted file occupied a space between two other files, there will be an unused area of disk memory. Files should occupy contiguous sectors, that is, they should come one after the other without gaps. A gap could be filled with a file that is small enough to occupy the available space but in many cases you will want to save larger files. In this case, the gap is wasted space. The command:

*COMPACT (<drive number>) or *COM. (<drive number>)

will move all the files after the gap, closing them up so that the space is filled. Sectors will now be occupied contiguously, thereby making all the disk available for use. If the drive is not specified then the current drive will be compacted. After compaction, file information is displayed followed by the number of free sectors in hexadecimal. If you have a program in memory before you compact, make sure that you save it first because *COMPACT utilises user memory to read in files before writing them back to a new disk location. Any program previously in memory will be corrupted during this activity.

CHAPTER THREE

COPYING, DELETING AND PROTECTING FILES

The ambiguous file specification (afsp)

We are building quite a comprehensive battery of SEDFS commands and you should now be fairly fluent in the handling of files. So far we have been restricted to keeping files on the same disk or volume but if we wanted to move them around it would be extremely tedious to load the file, change drive, disk or volume and save it again etc, etc. There are other commands which will do this operation for us more efficiently by moving, deleting or copying more than one file at a time but before we can use all of them to the greatest advantage, we need to know about wildcards.

Wildcards

This somewhat whimsical term does not refer to uncontrollable playing cards but to two characters which can stand for one or more characters in a filename or directory. They are:

```
      # ..... This can stand for the directory or any single
              character in the filename
      * ..... This can stand for a complete filename or any number of
              characters at the end of a filename
```

Armed with this knowledge, let's look at some examples.

```
   A.* .......... Applies to all files on directory A
   #.* .......... Applies to all files on all directories
   A.SLO# ....... Applies to all four lettered filenames beginning
                  with SLO on directory A
   A.#SLO ....... Applies to all four lettered filenames ending with
                  SLO on directory A but beginning with any letter
   A.SLO* ....... Applies to any filenames beginning with SLO on
                  directory A regardless of length
```

File specifications which can use wildcards are called "ambiguous" file specifications. Ambiguous file specifications allow a number of files to be processed without the need to type each individual filename. As a practical example we will copy some existing files.

*COPY

Format both sides of a 40 Track disk and enter some files on drive 0. The format of the *COPY command is:

```
      *COPY <source drive>  <destination disk>  <afsp>
```

So, for example, to copy the file "SLOG" from drive 0 to drive 2 would be:

```
      *COPY 0 2 SLOG
```

If we wanted to copy all the files in drive 0 to drive 2, we would enter:

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

Copying from one disk to another can become troublesome if you only have a single disk drive. To copy onto a different disk with a double disk drive, the following could be used:

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

With a single disk drive it is necessary to specify the destination drive to be the same as the source drive. This way, SEDFS will prompt for the source disk and the destination disk to be inserted into the drive unit as required:

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

This will copy all the files from drive 0 on one disk to drive 0 on another. The first prompt will be to insert the source disk and hit a key. The filename and information will be displayed. The next prompt will be for the destination disk so remove the source disk, insert the destination disk and hit a key. All the file data that was loaded into the computer's memory from the source disk will now be written to the destination disk and then the source disk will be requested again. This happens because memory space is limited and only parts of large files can be transferred in one go, so the prompting for the source disk and the destination disk could be repeated many times until the whole file has been copied. It is important to remember that user memory is overwritten during the copying process with the result that any program you had in memory prior to the operation will be lost. Do not forget to save any programs resident in memory before commencing a *COPY.

If the destination disk does not have enough room for the file then the error:

```
Disk full
Bad program
```

will be displayed. It is therefore wise to perform a *STAT to ascertain the spare disk space and, if necessary, a *COMPACT before copying. If these have no success then delete an existing redundant file or copy to another disk.

Copying Complete Drives

There are other copy commands which operate on whole volumes or complete sides of disks without specifying filenames. These are *BACKUP and *MCOPY.

***BACKUP**

As the name suggests, this is a very useful and easy way to make backup copies for security reasons. Disks are easily damaged so it is always wise to keep copies especially when valuable information is involved.

This command makes an exact copy of the source disk onto the destination disk. This is a potentially disastrous operation as the previous contents of the destination disk are lost in the process. The safety catch *ENABLE must be used before *BACKUP can commence and again be careful because, as with *COPY, user memory is overwritten. The command is simple to use and only the source and destination drives need to be specified. It takes the form:

```
*BACKUP <source drive>  <destination drive>
or
*BAC. <source drive> <destination drive>
```

If you intend to use this command regularly you will soon want to invest in a double disk drive. Although the problem is not so acute with 40 track, with 80 track on a single disk drive, the disks will have to be swapped many times before the whole disk has been copied.

***ENABLE**

This command is usually used directly before "dangerous" commands i.e. DESTROY, BACKUP and MCOPY, these functions having irreversible effects such as deleting filenames! A test is made in the *DESTROY, *BACKUP and *MCOPY commands as to whether the *ENABLE command was made. If it wasn't then a prompt..."Are you sure? Y/N" is made as a safety precaution.

***MCOPY**

There is another command called *MCOPY which performs the same function as *COPY but it is much quicker because it will copy as many files as memory will allow in one go, rather than working each file as an individual block. Like *BACKUP, *MCOPY destroys all existing files on the destination drive and for this reason, *ENABLE must first be used. Unlike *BACKUP (which produces an exact copy of a disk), *MCOPY only copies the files and not the unused disk space. It also compacts the drive or volume and the computer's user memory is overwritten (so again beware of leaving unsaved files in memory).

The command takes the form:

```
*MCOPY <source drive>  <destination drive>
```

The destination drive must not be smaller than the source drive or the message:

```
Drive ... larger than Drive ...
```

will be shown.

Deleting files

The ambiguous file specification is again a powerful tool and as well as being used to copy many files in one operation it can also be used to delete them. *WIPE deletes all files which match the ambiguous file specification. As with *DELETE, their entries are removed from the catalogue. The command is:

```
*WIPE <asfp> or *W. <afsp>
```

The files matching the ambiguous file specification will be listed individually. As each one is displayed you will be prompted for a Y or N. This gives you last minute control over whether to delete or not. On the other hand a *DESTROY, which also uses the ambiguous file specification, produces a complete list of all the matching files with one prompt. If you choose to delete then they all go at once. The form of this command is:

```
*DESTROY <afsp> or *DES. <afsp>
```

File Protection

Many of the commands previously discussed must be used with care as they could accidentally be used to erase a valuable file. To help guard against this, the command *ACCESS is provided to lock files under the ambiguous file specification. Locked files cannot be deleted or overwritten although they are powerless against *FORMAT, *BACKUP or *MCOPY. The command form is:

```
*ACCESS <afsp> (L) or *ACC. <afsp> (L)
```

If the optional L is included after the ambiguous file specification, the file or files will be locked and will appear in the catalogue followed by L. If the L is omitted from the command then previously locked files under the ambiguous filename will be unlocked. So, to lock all files in the current drive, use:

```
*ACCESS #.* L
```

To unlock them use:

```
*ACCESS #.*
```

Locked files cannot be written to in any way but are completely accessible to reading.

CHAPTER FOUR

THE BASIC PROGRAM AND ASCII FILES

The BASIC Program File

We will use example 2.1 from chapter 2 to make a BASIC program file. Type in example 2.1 and save it under the filename "PROG1".

Example 2.1

```
 5 REM ASCII DISPLAY
10 REPEAT
20 INPUT"PRESS A KEY",A$
30 PRINT ~ASC(A$)
40 UNTIL FALSE
```

To see the program as it is on the disk, we use the command:

```
*DUMP <fsp>
```

which dumps the contents of the specified file onto the screen. To dump PROG1, type:

```
*DUMP PROG1
```

You will now be looking at the program as it is recorded on the disk (Example 4.1).

Example 4.1

```
Location] [      File Data      ] [ASCII Representation

     0000 0D 00 05 14 20 F4 20 41 .... . A
     0008 53 43 49 49 20 44 49 53 SCII DIS
     0010 50 4C 41 59 0D 00 0A 06 PLAY....
     0018 20 F5 0D 00 14 16 20 E8  ..... .
     0020 22 50 52 45 53 53 20 41 "PRESS A
     0028 20 4B 45 59 22 2C 41 24  KEY",A$
     0030 0D 00 1E 0D 20 F1 20 7E .... . ~
     0038 97 28 41 24 29 0D 00 28 .(A$)..(
     0040 08 20 FD 20 A3 0D FF ** . . ....
```

The column of four figured numbers on the left are byte numbers in hexadecimal. These start at 0 and run to 7 in the first line. The second line starts at 8. The middle block is the program itself as bytes in hexadecimal and the column on the right is the ASCII representation of the program.

Notice that all the BASIC keywords are missing from the ASCII column. This is because they are stored in a "tokenised" form i.e. they are each coded into a one byte value to produce a more compact file. For example, the first BASIC word REM is stored as &F4. You can see this in byte 5. Note that & is used to denote that a number is hexadecimal unless as in example 4.1 where the numbers are automatically taken to be hexadecimal.

The program starts with &0D. This indicates the start of the line and is the ASCII value for the <RETURN> key. The next two bytes are the line number followed by the line length in one byte. The program data follows. Each line takes this format and you can check the whole program if you wish against the token values and ASCII values given in your computer manual. The program ends with &0D &FF as does every BASIC program.

The ASCII or Text File

Before we can proceed onto an explanation of the ASCII file, it is necessary for you to understand just what is meant by the term "ASCII". It stands for "American Standard Code for Information Interchange". This rather grand title refers to a standard which gives text characters (the alphabet, punctuation, etc) and some control commands (such as carriage return), an individual number. Use example 2.1 to display the ASCII number in hexadecimal of each key as it is depressed. Try upper and lower case using <SHIFT> and <CAPS LOCK>.

An ASCII file can be created by first opening it with the command:

```
*SPOOL <fsp>
```

and closing it with:

```
*SPOOL
```

We will now make an ASCII file with example 2.1. First make sure that example 2.1 has been loaded into memory then open the ASCII file with command:

```
*SPOOL ASKY
```

*SPOOL is the command to open the file and ASKY is the filename. Notice that inverted commas are not necessary. After this command, everything that is displayed on the screen will be written to the file so to get our program onto the screen, list it with:

```
LIST or L.
```

When the list is complete, type *SPOOL again but without the filename to close the file. Having made our ASCII file, we can look at it with:

```
*DUMP ASKY
```

Although this really is the same program as before, it appears longer because every character is an ASCII one, therefore BASIC keywords are written in full rather than in token form.

Notice that L. the list command is present at the beginning of the file because it was typed in after *SPOOL.

To view the file in its original form, two commands are available. They are:

```
*TYPE <fsp> and *LIST <fsp>
```

*TYPE <fsp> displays the file as it looked when copied to disk. *LIST <fsp> displays the file with line numbers.

Merging files

A number of operations can be performed with an ASCII file. One is to spool a BASIC program and then enter a word-processing program. Using the facilities of the word-processor, the ASCII version of the BASIC program can be merged into a document. Most word-processors will be able to edit the program and turn it back into a BASIC program for normal use. In the same way it should be possible to spool portions of text so that they can be fed into a different word-processor. This facility is useful when preparing text for recipients who possess different equipment.

An ASCII file of a BASIC program can be turned back into a BASIC program with the command *EXEC <fsp>

This reads the named ASCII file straight into the keyboard buffer with the effect that the computer treats the incoming data as though it had been typed in from the keyboard.

The *SPOOL/*EXEC sequence is very useful for merging two BASIC programs. It is often the case that useful procedures are stored on a separate disk of their own to be used as necessary. For example, you might have some graphics utilities - one procedure for drawing circles, another for squares, etc. Utilities like these could be used over and over again in different programs. All you need to do is incorporate them into any program is convert them to ASCII files using *SPOOL and then *EXEC them into the program under development.

Although this is not a difficult operation there are certain considerations which must be taken into account. First make sure that the line numbers in the main program are different from those in the utility program because when a file is merged with one already in memory, the incoming line numbers are added to the existing line numbers. This means that any line numbers which are the same will be replaced with the incoming line numbers. By making the incoming line numbers higher than the existing line numbers, the ASCII file will be added to the end of the existing program, or by using this technique with care the incoming file can be placed at any point within the existing program. A good way to think of this process is to imagine that you are actually typing the spooled file in by hand because that is just how the computer treats it.

The following example illustrates the procedure for merging two BASIC programs. Type in the example 4.3 and save it under the filename "P-1".

Example 4.3

```
10 MODE 4
20 PRINTTAB(10,10)"DRAW A SQUARE"
30 PROCsquare
40 END
```

Now type NEW to clear memory and type in example 4.4. This is the procedure, called in line 30 of the first program, to draw a square.

Example 4.4

```
50 DEFPROCsquare
```

```
60 MOVE 400,400:DRAW 500,400
70 DRAW 500,500:DRAW 400,500
80 DRAW 400,400
90 ENDPROC
```

Instead of saving this procedure as a BASIC file, use *SPOOL to create an ASCII file called "P-2". You should now have the files "P-1" and "P-2" on disk. To merge them, LOAD "P-1" as normal and LIST to make sure the program is correct. Now type:

```
*EXEC P-2
```

You can now see the two programs merged into one. Run the new complete program and the square will be drawn on the screen. The worrying message "Syntax error" which always appears after a merge can be ignored.

Creating Lists

*SPOOL can be used to create lists of DATA, consider 4.5 below:

Example 4.5

```
10 *SPOOL RANDOM
20 FOR N=1 TO 20
30 DIE1=RND(6)
40 DIE2=RND*6)
50 PRINTTAB(0);"DIE-1 ";DIE1,TAB(20);"DIE-2 ";DIE2
60 NEXT
70 *SPOOL
```

Example 4.5 simulates the auction of throwing two dice and saves the results of 20 throws in an ASCII file called RANDOM. Notice the use of TAB. Careful use of dotted or broken lines can create attractive tables which could be merged into a document using a word-processor. For example, adding the following line separates each throw by a dotted line:

```
55 PRINTTAB(0);STRING$(39,"-")
```

To inspect the table once completed use *TYPE or *LIST. For long tables which are larger than the screen and scroll past too quickly, use paged mode by holding down <CTRL> and pressing "N". Only small portions of text will be scrolled onto the screen until <SHIFT> is pressed to access the next portion of data. Paged mode can be cancelled by holding down <CTRL> and pressing "O".

EXECutive Files

ASCII or text files can be used in another way to control the operation of other programs. In this guise they are used as executive files.

An ASCII file can be produced directly from the keyboard without having to go through the *SPOOL procedure. The command:

```
*BUILD <fsp> or *BU. <fsp>
```

is used for this purpose and files created in this way can be *EXECed in the same way that a *SPOOLed file can. The file may contain any set of commands which can be produced from the keyboard (such as

"*commands") to control the operating system or ROM software (DFS included), printer control codes, etc. The most common use of this type of file is the !BOOT file. You may have noticed that commercial disk software is often loaded and run by holding down <SHIFT> while pressing <BREAK>. This facility is extremely useful in that it can be used not only for loading programs but also for initialising the computer for a specific purpose. For example, for programming the red user-defined keys, carrying out *FX commands. All these initialisation and load commands are stored in a special ASCII file created with *BUILD and called !BOOT.

Here is how to make a !BOOT file. First type:

```
*BUILD !BOOT
```

The computer now knows that it is to make an ASCII file directly from the keyboard called !BOOT. A line number will appear and you must type in the command you want to place in the !BOOT file. For example, suppose you have a program that you wanted to load called "ANDREW", you would simply alter the line number:

```
1 LOAD "ANDREW"
```

To run the program as well, you would of course use CHAIN rather than LOAD. Press <RETURN> and line 2 will appear. You can now type in another command. For instance, why not set up a red function key. A useful command for one of these is to save a particular program. When developing software it is wise to save your work periodically in case you have an accident such as pressing <BREAK> or corrupting memory with some faulty machine code or even if there is a power cut, you will at least have a recent copy. It can become tedious to keep typing SAVE "filename" and putting the command onto a function key greatly facilitates the operation. So, on line 2, type:

```
2 *KEY 0 SAVE "filename"
```

On pressing <RETURN>, line 3 will appear and you can keep adding commands in this way until you are satisfied that all your needs are catered for. To save the file, press <ESCAPE> in response to a line number. The !BOOT file is now on the disk and can be brought into action with *EXEC just like any other ASCII file but to make <SHIFT><BREAK> effective, the correct start-up option must be placed in the catalogue. We do this with:

```
*OPT 4 <number> or *O.4 <number>
```

This command sets the start-up action (i.e. on <SHIFT><BREAK>) which will operate on the !BOOT file contained on the disk in the current drive. In double density, it is possible to set a separate start up action for each volume. Perform a *CAT and look at the entry above the library information. You will see "Option 0 (off)". This tells us that the start-up action is for that drive. At present there is no start-up action. Type:

```
*OPT 4 1
```

and perform another *CAT. The option will now be number 1 and in the brackets will be (LOAD). The start-up action on the !BOOT file will be *LOAD. Option 2 as selected with:

```
*OPT 4 2
```

will cause a *RUN of the !BOOT file. Options 1 and 2 are only useful when the !BOOT file is in machine code.

```
*OPT 4 3
```

will select option 3 and the start-up action on the !BOOT file will be *EXEC.

```
*OPT 4 0
```

will return the disk to the off state where there is no start-up action. If options 1, 2 or 3 are present and the start-up action is initiated with a !BOOT file not being present then the message:

```
File not found
```

will be displayed.

CHAPTER FIVE

THE BASIC DATA AND BINARY DATA FILE

The BASIC Data File

This is used for storing sequences of BASIC variables (integer, real and string variables). To read or write a BASIC data file, it must first be opened by allocating to it a channel and buffer.

Channel a route along which data is passed to and from the disk. Channels are identified by the numbers 17 to 21 in the disk filing system whereas other filing systems use different numbers.

Buffer An intermediate storage area in memory where data is held prior to reading or writing the file. This allows a block of data to be accumulated before writing and during reading which can then be processed in one operation thereby reducing the number of disk access operations.

The specific buffer area allocated for a given file is determined by the channel number. Five separate buffer areas are available allowing a maximum of five opened files at any one time. In normal conditions the user need not be concerned with the channel numbers and buffers as SEDFS automatically takes care of these operations.

To open a file and allocate a channel number, the command OPENOUT is used. The full syntax is:

```
<num-var> = OPENOUT <filename>
```

The file is allocated a channel number which is returned in a numeric variable. For example:

```
X = OPENOUT "FILE1"
```

X contains the channel number allocated to the file "FILE1". In all operations the variable X is used as opposed to the filename itself. OPENOUT is only used to create new files and any file of the same name which already exists will be overwritten.

Example 5.2 will read the whole file:

```
10 Y=OPENIN "COUNT"
20 FOR N=1 TO 10
30 INPUT# Y,M
40 PRINT M
50 NEXT
60 CLOSE# Y
```

Line 30 instructs the computer to input a character from the file and place it in the variable M. Line 40 PRINTs the variable M on the screen so that we can see it. Do not confuse the ordinary PRINT command with the file command PRINT#. Think of it in this way:

```
PRINT  sends characters to the screen
PRINT# sends characters to the opened file on the specified
       channel.
```

In the same way, do not confuse INPUT with INPUT#:

INPUT reads characters from the keyboard
INPUT# reads characters from the opened file on the specified channel.

The FOR NEXT loop in lines 20 and 50 causes all ten characters to be read. If the length of the file is not known the program can be made to stop automatically by detecting the end of the file with:
EOF# <channel>.

Let us convert the previous example to do this. Instead of a FOR NEXT loop we will use a REPEAT UNTIL loop with EOF# as the terminating condition. The program therefore becomes:

Example 5.2a

```
10 Y=OPENIN "COUNT"
20 REPEAT
30 INPUT# Y,M
40 PRINT M
50 UNTIL EOF# Y
60 CLOSE# Y
```

The program will stop on reaching the end of the file, a facility which is essential when the number of items in the file is not known. OPENOUT and OPENIN are useful for performing complete operations on whole files. The procedure is:

1. Create a file using OPENOUT
2. When the file needs updating, read into memory the whole file using OPENIN. Arrays can be used to store the data
3. Perform alterations to the file whilst in memory
4. Write the new file to disk with OPENOUT using the same filename. The previous file will be deleted

Random Access

The previous use of OPENIN and OPENOUT is fine if files are not going to exceed user memory. Some word-processors for the Acorn Electron can only work on files which are limited to user memory and so only a few pages of A4 format can be held in each file. Others are capable of working on larger files and read into memory only a small part of the file. On saving, only the part of the file being used is written to disk. This ability to write or read specific parts of the file is made possible by using a technique called random access. Working with random access files means that file size is only limited by disk capacity.

With two more commands, we can use random access. They are OPENUP and PTR#.

OPENUP is similar in syntax to OPENIN and OPENOUT and can open a file for reading and writing:
<num-var> = OPENUP <filename>

PTR# stands for "Pointer" and the value of the pointer determines which byte in the file is to be read or written to next. For example, if its value is 10, the pointer will be at byte 10:

PTR# <num-var> = <numeric>

The numeric variable corresponds to the variable containing the channel number as assigned with OPENUP. This is necessary to make sure that the correct file pointer is accessed, especially when there may be more than one file open at a time. The numeric controls the position of the pointer in bytes.

To be able to use this command we need to know how long each entry is so that we can determine where to place the pointer. A string variable occupies the number of letters in it plus 2. For example:

ANDREW

would be 8 bytes long. If, for the sake of explanation, all entries in our list are four letters long then each entry would be 6 bytes long. So, in pointer terms each entry can be found at the entry number times 6. i.e.:

Entry No.	Pointer value	Entry
0	0	BATH
1	6	SOAP
2	12	TAPS
3	18	PLUG

Using the previous list program, create a file with a number of entries all four letters long. For example:

```
AAAA
BBBB
CCCC
DDDD
```

Having created the file, type NEW to clear memory and then type in the following program:

Example 5.4

```
 10 X=OPENUP "FILE"
 20 REPEAT
 30 INPUT"Record to Read";A
 40 IF A=1000 THEN 80
 50 PTR# X=A*6
 60 INPUT# X,A$
 70 PRINT A$
 80 UNTIL A=1000
 90 CLOSE# X
```

The command OPENUP in line 10 opens the file "FILE" for reading or writing. Line 30 prompts for the required entry. Line 5 multiples the entry number by 6 to obtain the correct byte number which is used by PTR# in line 50 to find the correct entry on the disk. Line 60 reads the entry and line 70 prints it on the screen.

The same program can easily be adapted to change an entry. PTR# is used in the same way but instead of reading the entry, we write to it:

Example 5.5

```
10 X=OPENUP "FILE"
20 REPEAT
30 INPUT"Record to Change";A
40 IF A=1000 THEN 80
50 PTR# X=A*6
60 INPUT"New Record";B$
70 PRINT# X,B$
80 UNTIL A=1000
90 CLOSE# X
```

The only differences from the previous example are that line 60 prompts for the new entry and line 70 writes it to the file.

Having changed the file "FILE" by these two random access programs, enter the file using the original example and read the names. You will see that the file has been changed and is still quite useable.

We have made all out entries 6 bytes long so that the pointer management is simplified. Obviously it is impractical to make all the names four letters long but to keep track of the pointer it is necessary to make entries all the same length. This can be done by determining the maximum length of every name. Let us use 10 for this value. If any names are not 10 letters long, which most will not be, then we can pad the string out with spaces so that the entry on the disk will always equal 10. The pointer will now be incremented/decremented in steps of 12. i.e. 10 + 2 = 12.

Random access with variables of different lengths

Probably the most difficult aspect of using random access files is the correct placing of the pointer. The previous example uses entries all with 10 characters and this makes programming fairly easy, but using different kinds of variables demands that individual entries be of different lengths. When variables are sent to the disk file using PRINT#, they are laid down in a special format.

Integer variable 5 bytes

&40 followed by the number as twos compliment in four bytes, low byte first. For example, 10 would take the following form in hexadecimal:

```
40 00 00 00 0A
```

Real variable 6 bytes

&FF followed by the mantissa in four bytes and the exponent in one byte. 10 would take this form:

```
FF 00 00 00 20 84
```

String variables 2 bytes plus number of characters (bytes) in string.

&00 followed by the number of characters in one byte followed by the characters in reverse order in ASCII form. For example, SEDFS would look like this:

```
00 05 53 46 44 45 53
```

Much of this information is specialised and especially with real variables can be a little involved mathematically. It is not absolutely necessary to completely understand how the figures are

arrived at as long as the length of each variable is clear. It can also be very helpful to be able to recognise the different variables by their first bytes so that *DUMP can be of use when de-bugging a program. It can be very revealing to inspect a file as it appears on the disk to find out exactly what a program is doing.

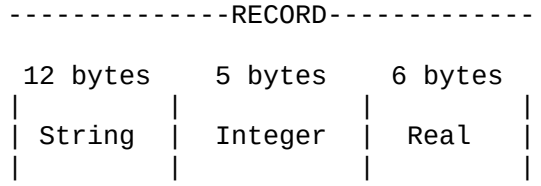
When creating a file it is essential that it is laid down in an organised manner so that when reading it the pointer can be manipulated correctly. If the computer was expecting an integer (INPUT#Y,A%) and the pointer was at a string variable, the error message:

```
Type mismatch
```

would be generated. The same error would occur if the pointer was at a point other than the beginning of the variable.

Fields

A record is a unit within a file and so far we have explained the use of records containing only one item of data. It is possible to divide each record into a number of smaller units called fields. For example, the name list programs given earlier could be altered to handle not just names but also addresses and ages. The record could contain different types of variables:



The example shows a record with three fields consisting of all three types of BASIC variables. The string variable contains 10 characters. The whole record contains 12+5+6=23 bytes.

The program to write this format with 10 records to the file would be:

Example 5.7

```
10 X=OPENOUT "RECORD"
20 FOR next=0 TO 9 STEP 23 (next is the record number)
30 INPUT "STRING",A$
40 INPUT "INTVAR1",A%
50 INPUT "REALVA2",B
60 PTR# X=Next             (next increments the pointer to the
                            start of each record on the disk)
70 PRINT# X,A$
80 PTR# X=next+12          (next+12 increments the pointer to
                            the next byte after the string)
90 PRINT# X,A%
```

```
100 PTR #X=next+17          (next+17 increments the pointer to
                             the next byte after the integer)
110 PRINT# X,B
120 NEXT next
130 CLOSE# X
```

The program to read would be:

Example 5.8

```
 10 X=OPENIN "RECORDS"
 20 FOR next=0 TO 9 STEP 23
 30 PTR# X=next:INPUT#X,A$
 40 PTR# X=next+12:INPUT#X,A%
 50 PTR# X=next+17:INPUT#X,B
 60 PRINT A$,A%,B
 70 NEXT
 80 CLOSE#X
```

In this way files containing records with fields of different lengths can be built up.

Hopefully this section on BASIC data files will have whetted your appetite to delve deeper into the world of data processing. Why not try constructing a file of variable length records? The difficult with this type of file is that the pointer requires to be incremented or decremented by variable amounts. The exact amount could be held at the beginning of each record and calculated automatically at the time of writing the record.

The Binary Data File

If we wished to escape from the intrinsic structure of BASIC and form files to our own specification, the commands:

<num-var> = BGET# <channel> and BPUT# <channel>, <numeric>

will read and write individual bytes. The file must first be opened with OPENOUT in the normal way. The following program will store a list of numbers on a file. Don't enter numbers larger than 255 because this is the limit of one byte. Numbers which exceed this value will be repeatedly subtracted by 256 until they are less than 256.

Example 5.9

```
 10 X=OPENOUT "NUMBERS"
 20 FOR next=0 TO 9
 30 INPUT "A%"
 40 BPUT# X,A%
 50 NEXT next
 60 CLOSE# X
```

To read a single byte from an opened file BGET# is used. To read the file use the following program:

Example 5.10

```
10 X=OPENUP "NUMBERS"
20 REPEAT
30 A%=BGET# X
40 PRINT A%
50 UNTIL EOF# X
```

Of course, random access techniques can be used and are greatly simplified by not having to worry about different variable lengths. The resulting file will also be much more compact than a BASIC file.

CHAPTER SIX

THE BINARY PROGRAM FILE AND MEMORY USAGE

This type of file is used to store machine code programs. A machine code program is, like all other programs, a collection of numbers with specific meanings but unlike others exists in a form which needs no interpretation for the computer to operate on it. BASIC consists of a number of keywords which are based on the English language. They have an almost obvious meaning and this enables the programmer to memorise and these keywords with relative ease. Although BASIC is a convenient and flexible language for us to use, it has to be interpreted for the computer to be able to work on it. Each instruction has to be converted to machine code every time that the program is run, and the main drawback of this is that quite a lot of time is taken up during the process.

Machine code on the other hand is processed by the computer without intervention from any other source. It is therefore very fast and allows access to machine functions that BASIC is not equipped to handle. Due to the unfriendliness of raw machine code, programs are written in assembly language which uses mnemonics to give meaning to instructions. The program as it stands in assembler language is called the source code. This is then assembled into machine code at a specific memory location into its final form called the object code. Acorn Electron users are fortunate to possess a built in assembler for this purpose. It is, however, not our intention to explain assembly language, and newcomers to the subject should read one of the many available text books on the market.

A machine code program (that is, object code) can be loaded and saved to and from any part of the computer's memory, whether it be user RAM or elsewhere. Before we go into the details of moving machine code around we will take a look at the computer's memory.

Memory space on the Acorn Electron is arranged into 256 sections called Pages, each page holding 256 bytes. A memory location is therefore expressed in four hexadecimal characters. The first two represent the Page itself, the second two the location within the Page. Hence the complete memory contains 65536 bytes (&FFFF bytes) or 64K. Not all of this is available to the BASIC programmer and on a system with SEDFS, user RAM is from &E00 to &7FFF. User RAM is also used for the screen memory which starts at different locations for different modes. The start of screen memory is held in a pseudo variable called HIMEM. The start of user RAM is held in a pseudo variable called PAGE. Memory above &7FFF is normally off limits and is set aside for the paged ROMs (BASIC and SEDFS operate from here) and the Operating System.

Memory below &E00 is used for many purposes and much of it is available for use by the machine code programmer. The following list of memory locations gives guidelines as to where machine code can be stored below user RAM.

Page 0 &00 - &FF
This is used by the Operating System and a small part from &70 to &8F is open to the machine code programmer. However, there is not enough space to store whole programs.

Page 1 &100 - &1FF
This is used by the 6502 stack.

Page 2 &200 - &2FF
This is used by the Operating System.

Page 3 &300 - &3FF
This is used by the VDU, cassette system and keyboard buffer.

Pages 4 to 7 inclusive, &400 - &7FF
This is used by the currently active language which operates from ROM.

Page 8 &800 - &8FF
This is used by the sound facilities as workspace, buffers, envelopes 1 to 4 storage and also as a printer buffer. If your program does not use sound or the printer then this area can be used.

Page 9 &900 - &9FF
This is used in a number of ways and, like page 8, is mainly a buffer area. It contains the speech buffer, cassette and RS423 output buffers. Envelopes 5 to 16 from the sound facilities are stored here.

This space can be used for machine code programs if you do not use envelopes numbered above 4, use the serial interface or use the cassette system for file handling. Ordinary loading and saving will not affect this buffer but, in any case, now that you have the SEDFS system, the cassette unit will be rarely used.

Page 10 &A00 - &AFF
This is another buffer area for the RS423 and cassette systems but this time for input. This area is usually free.

Page 11 &B00 - &BFF
The soft key buffer resides here. The soft keys are the user-definable function keys which are programmed with *KEY so steer clear of this area if you are using them.

Page 12 &C00 - &CFF
ASCII characters 224 - 255 as defined by VDU23 are stored here. This space is available should you not use these characters.

Page 13 &D00 - &DFF
This area is allocated for the NMI routine, paged ROM extended vectors and the paged ROM workspace table.

Page 14 &E00
This is the start of the user RAM. The value &E00 is held by default in the pseudo variable PAGE but can be altered should the need arise by resetting PAGE. Machine code can be placed in the area of user RAM (&E00 - &7FFF) but this is mainly used for BASIC programs.

The Sideways ROMs

The abbreviation ROM means Read Only Memory and the sideways ROMs are special chips which are placed into the printed circuit board of the computer and are used to hold specialised software. SEDFS and BASIC are held in sideways ROMs as are many other specialised programs such as wordprocessors, databases and graphics utilities.

Sideways ROMs occupy memory from &8000 to &BFFF. As you may already know, the operating system can handle up to 16 sideways ROMs. Although it is not possible to have all of them in memory at the same time, they can be "paged" in or activated by using the appropriate command. For example, to enter Slogger's STARMON, type *STARMON. Even though STARMON is sitting in its ROM socket, it will not be "paged" in until called by its *command. Some ROMs such as SEDFS will be "paged" in automatically as needed depending on the type of entry conditions for that particular ROM. This ingenious system is responsible for the Acorn Electron's ability to access a large number of specialist programs and yet still retain user RAM intact.

There are one or two commands which are directly concerned with sideways ROMs.

From our short look at memory usage, it is clear that the best places to put machine code programs are in the buffer areas as long as the buffer will not be used in the execution of the program. For instance, you would not place a program to play music in the sound workspace in page 8 as the program would be destroyed as soon as it was run.

Having found out where to put our programs, we will now look at how to put them there. The first and most obvious solution is to assemble new code at the correct address but often you will want to be able to load in the already assembled program without reference to the source code. To be able to do this we need to save the object code to disk. BASIC programs are straight forward to save and load because the language itself takes care of where in memory the program is placed by referencing PAGE. When writing machine code, these facilities are not available and programs have to be thought of as blocks of memory which can be copied with reference to the start location and length. We transfer blocks of memory from the computer to the disk by means of the *SAVE command. This is similar to SAVE but allows the program to be recorded with information on length, start address, execution address and load address.

The full command is:

   *SAVE <fsp> <start addr> <end addr+1> (<load addr>) (<exec addr>)
or
   *SAVE <fsp> <start address> + <length> (<load addr>) (<exec addr>)

Note: Parameters in brackets are optional.

Let us suppose we wish to save a program called "ROSE" which is 10 bytes in length starting at address &A00. The command using the first version of *SAVE would be:

```
      *SAVE ROSE A00 A0A
```

Alternatively, instead of working out the address, we could specify the start address + the length:

```
      *SAVE ROSE A00 +A
```

The last two parameters are optional and, if not specified, will be set to the same value as the start address. Notice that the ampersand sign (&) is not necessary because all figures are automatically taken as hexadecimal.

Having saved the file to disk we can inspect this information with:

```
      *INFO <afsp>
```

This being an ambiguous file specification, a range of files can be examined in one operation.

Type:

```
*INFO ROSE
```

The result should be something like this:

```
$ROSE       000A000  000A00  00000A  05B

  1    2       3        4       5     6
```

1. Directory/Filename
2. Locked status
3. Load address
4. Execution address
5. Length of file in bytes
6. Start sector

The locked status in this case is blank. If the file was locked then L would appear in this space. For example:

```
$ROSE   L   000A000  000A00  00000A  05B
```

The start sector is the actual place where the file starts on the disk. In SEDFS, with 10 sectors per track this would be track 10, sector 1.

To get the file back into memory, we use:

```
*LOAD <fsp> (<address>)
```

The address is optional and if not specified the file will be loaded at the load address as shown by *INFO which of course was determined at the time it was saved. If the address is specified then the file will be loaded at this address. i.e.:

```
*LOAD ROSE 900
```

will load the file ROSE at location &900.

This system of loading and saving files is extremely flexible and can be used to manipulate blocks of memory from and to any part of the complete 64K of memory.

Saving Screen Memory

One useful technique is to save screen memory so that at a later date, a graphics image can be displayed without running the original program to set it up again. The procedure is to save a block of memory from HIMEM to &7FFF.

Find HIMEM with:

```
PRINT ~HIMEM
```

The ~ (tild) will give the answer in hexadecimal. The command is:

```
*SAVE SCREEN (HIMEM) 8000
```

Variables cannot normally be used in *commands and so the value in hexadecimal is inserted rather than the variable HIMEM. Notice that the end address is &8000 rather then &7FFF because the last location+1 must be specified. To use the screen at a later date, just type:

```
*LOAD SCREEN
```

There is one more command for loading a machine code program which will run it as well. In this respect it is similar to the BASIC keyword CHAIN. It is:

```
*RUN <fsp> (<parameters>)
```

The program will be loaded into memory at the load address and will start to run at the execution address. This is useful because a machine code program, unlike a BASIC program, need not start running at the beginning. Parameters after the filename can be accessed by using OSWORD with 5 in the accumulator. This command cannot be used within a BASIC program.

SEDFS and Memory Usage

Overlays

Even though *SAVE and *LOAD are mostly used with machine code programs, they are able to work on BASIC programs which need to be loaded in at specific addresses. Overlays are almost essential where memory space is short as in the Acorn Electron with a screen mode which uses a lot of memory. Mode 2 takes up 20K of user RAM leaving only about 8K left for the user!

The solution to the problem is to load into memory only those parts of the program that you are using at any given time. As an example, suppose that you had a menu drive program with two parts. The menu is the link between the two working parts and must stay in memory all the time. Depending upon which option is selected from the menu, either part one or part two is loaded on to the end of the menu. The part which stays in memory all the time is called the zero overlay and those that are loaded in later are called primary overlays. It is possible to load onto any of the primary overlays to produce a secondary overlay and so on. Before we go further, let us look at a memory map of user RAM.

```
The program starts at PAGE.
The end of the program is called TOP.
The storage area for the program variables is between HIMEM and
LOMEM.
In the default state, LOMEM and TOP are the same.
```

If the primary overlay is loaded directly onto the end of the zero overlay in this case then the variables will be lost. Before operations start the variables need to be placed in a safe area by raising the value of LOMEM to an area of sufficient size just below HIMEM. Make sure you have enough room for the variables by estimating the memory requirement of the variables. To do this add up the number of bytes taken by each variable.

```
Integer variables are 5 bytes long.
Real     variables are 6 bytes long.
String   variables are 2 bytes + (number of characters) long.
```

Subtract the total from HIMEM and you have the value for LOMEM. Add a few more bytes for good measure. If there is not enough room, the "No room" or "No such variable" messages will most likely tell you so on running the program. Change the value of LOMEM with:

LOMEM = <new value>

LOMEM will not now be the same as the top, indeed, there will be a space between TOP and LOMEM where the primary overlay can be placed.

Now to find where to place the primary overlay so that is joins correctly onto the zero overlay. First find TOP with PRINT~TOP. (The tild sign prints the answer in hexadecimal.) This will be the end of the program and will contain the end of program marker. This takes the form of two bytes containing &0D &FF. For another program to be loaded onto the end of the zero overlay, these two bytes must be destroyed so that BASIC will continue to read past this point. The trick is to load the primary overlay in at TOP-2. We must *LOAD <fsp> at TOP-2. It is essential that the program numbers in the primary overlay are higher than those in the zero overlay.

The next three short programs serve to illustrate the principles of overlaying. The first is the zero overlay. Type it in and save it under the filename "ZERO". Notice than line 10 sets LOMEM to TOP+90. This reserves plenty of space for the primary overlays to go into. If this line was omitted the primary overlays would go on top of the variables causing a "No such variable" error. Lines 100 and 140 load the primary overlays at address &1A0E. When you have typed in the program, it is best to check this figure with:

PRINT ~TOP-2

This is because if there is any difference in your program such as an extra space, this figure will be different. If it is wrong then on *LOAD you will get a "Bad Program" error.

Type NEW to clear memory and type in example 6.2 (the first overlay) and save under the filename "P1".

Type NEW and type in example 6.3 and save it under the filename "P2".

Now load in example 6.1 and run. On selecting 1 or 2 from the menu, either P1 or P2 will be loaded in. Prove it to yourself by typing "1" or "2" and <ESCAPE>. LIST and you will see the overlay.

Example 6.1

"ZERO"

```
 10 LOMEM=TOP+90
 20 REPEAT
 30 PRINT"To Load Primary Overlay 1 Press 1"
 40 PRINT"To Load Primary Overlay 2 Press 2"
 50 G$=GET$:IF G$<"1"ORG$>"2" THEN 50
 60 IF G$="1" THEN PROCload1
 70 IF G$="2" THEN PROCload2
 80 UNTIL FALSE
 90 DEFPROCload1
100 *LOAD P1 1A0E
110 PROC1
120 ENDPROC
130 DEFPROCload2
140 *LOAD P2 1A0E
150 PROC2
160 ENDPROC
```

Example 6.2

```
"P1"
      200 DEFPROC1
      210 PRINT"This is Primary Overlay 1"
      220 PRINT"Press space to continue"
      230 G$=GET$:IFG$<>" " THEN 230
      240 ENDPROC
```

Example 6.3

```
"P2"
      200 DEFPROC2
      210 PRINT"This is Primary Overlay 2"
      220 PRINT"Press space to continue"
      230 G$=GET$:IFG$<>" " THEN 230
      240 ENDPROC
```

These simple techniques work perfectly well but with one drawback. If we make any change to the zero overlay the value of TOP will change accordingly and we have to calculate TOP-2 and manually alter the load address of the primary overlays. When a program is in the development stages it is extremely inconvenient and it would be easier if we could use it in its variable form in the *LOAD command. Unfortunately in the present state this is not possible but with the help of OSCLI we can indeed use variables in *commands. *Commands are Operating System commands which are automatically passed to the command line interpreter. Using OSCLI (Operating System Command Line Interpreter) is just another way of performing a *command which takes the instructions in the form of a string.

To use this system with *LOAD, each part of the command is put into a string and concatenated (joined together) within brackets.

*LOAD with the following space can be expressed as "LOAD ". The filename is placed in a string variable, i.e. overlay$. The space between the filename and the load address is treated separately e.g. " ". The load address (TOP-2) is converted to a string using STR$ i.e. STR$~(TOP-2) ..... the tild ensures that the result is hexadecimal.

So after concatenation the complete line is:

```
      OSCLI("LOAD "+overlay$+" "+STR$~(TOP-2))
```

Using this method overlays will take care of themselves and always be loaded at the correct address thus making the programmer's task simpler and less tedious. The new version of the zero overlay is given below:

Example 6.4

```
       10 LOMEM=TOP+90
       20 REPEAT
       30 PRINT"To Load Primary Overlay 1 Press 1"
       40 PRINT"To Load Primary Overlay 2 Press 2"
       50 G$=GET$:IF G$<"1"ORG$>"2" THEN 50
       60 IF G$="1" THEN PROCload1
       70 IF G$="2" THEN PROCload2
       80 PROCload
```

```
 90 UNTIL FALSE
100 DEFPROCload
110 OSCLI("LOAD "+overlay$+" "+STR$~(TOP-2))
120 IF G$="1" THEN PROC1 ELSE PROC2
130 ENDPROC
```

Note: The use of OSCLI is not restricted to *LOAD but can be used on all *commands.

CHAPTER SEVEN

MORE SEDFS COMMANDS

***OPT**

*OPT has been extended to encompass some of the original SEDFS commands and takes the form:

```
*OPT <value> <value>
```

***OPT 1, <value>**

```
*OPT   1,0 turns off the filing system messages
*OPT   1,1 turns on the filing system messages
*OPT   1,2 turns on the extended messages
```

***OPT 7, <value>**

*OPT 7 allows the drive to operate at full speed. As access time accounts for the greater proportion of total operation time, this simple command will greatly speed up any program which relies heavily on disk operations.

```
*OPT 7,0              sets STEP RATE to 15 msec
*OPT 7,1              sets STEP RATE to 10 msec
*OPT 7,2              sets STEP RATE to  6 msec
*OPT 7,3              sets STEP RATE to  3 msec
```

***OPT 8, <value>**

```
*OPT   8,0   allows the filing system to read 40 Track disks in
             40 Track drives and 80 Track disks in 80 Track
             drives. (This is the default.)
*OPT   8,1   makes an 80 Track drive double step so that it can
             only read 40 Track disks.
*OPT 8,255   causes automatic Track detection to be activated,
             enabling 40 and 80 Track drives to be read in an
             80 Track Drive.
```

***FX255,value** ..... Read write start-up options (value in decimal)

SEDFS uses two bits (4 and 5) of this as an alternative to setting up the step rate. These are defined as follows:

```
Bits 0 to 2 ... Screen mode
Bit  3 ........ Reverse action of <SHIFT><BREAK>
Bits 4 to 5 ... Step rate
Bits 6 to 7 ... Future use

Example:   *FX255,63       Set 15 msec step rate (default value)
           *FX255,47       Set 10 msec step rate
           *FX255,31       Set  6 msec step rate
           *FX255,15       Set  3 msec step rate
```

Note that for the *FX255 command to take effect, the <BREAK> key must be pressed (<CTRL><BREAK> will restore these values to default)

***DATE**

The date and time are kept within the disk interface even when the computer is switched off and can be accessed through the DFS utility command *DATE. Unlike the standard Cumana system, the date and time are also altered using *DATE rather than the SET_TIME program. So, to set the correct date and time, type *DATE followed by the date and time as in the following example:

```
*DATE Mo 24:11:86 15:30
      |      |      |
     DAY   DATE   TIME
```

Then, by typing *DATE, the SEDFS will return the correct day and date and time as just entered.

***FDCSTAT**

If a disk error occurs during the operation of SEDFS then it is possible to determine the type of fault using the *FDCSTAT command. This command returns the error of the last disk operation from the disk interface and was non-zero for the error condition to arise. The section on error messages at the end of the manual gives full details of the errors reported by *FDCSTAT.

***RUNPROT**

This command is associated with the Slogger T2SEDFS Tape 2 Disk ROM. The T2SEDFS requires a page of 256 bytes of RAM from the Electron during the loading of a game from disk. This is not the case with *RUNPROT as it utilises the memory of the Disk Interface itself.

To run a game simply type:

```
*RUNPROT T.<filename>     (*RUNP. T.<filename>)
```

where <filename> is the name of the game.

***HELP (<keyword>) or *H. (<keyword>)**

A plain *HELP (without a keyword) provides a list of all the sideways ROMs, each of which may or may not produce a list of keywords by which more help can be obtained. After *HELP, if present, SEDFS will respond with the following:

```
      Slogger DFS <version No.>
        DFS                     |  Keywords
        UTILS                   |
```

Using the keywords, we can call up list of the commands open to us. *H.DFS gives the following list of SEDFS commands:

```
      Slogger EDFS

        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         <src drv>  <dest drv>
        LIB          <old fsp>  <new fsp>
        MCOPY        <src drv>  <dest drv>
        RENAME       <old fsp>  <new fsp>
        RUNPROT      <fsp>
        STAT         (<drv>)
        TITLE        <title>
        WIPE         <afsp>
```

*H.UTILS gives the following list of SEDFS utilities.

```
      Slogger EDFS

        BUILD        <fsp>
        DATE
        DISK
        DUMP         <fsp>
        FORMAT       <argument>
        FDCSTAT
        LIST         <fsp>
        TYPE         <fsp>
        VERIFY       (<drv>)
```

***MAP <volspec> or *MA. <volsp>**

Related to *STAT, this command shows the occupied and free space on the specified drive in terms of sectors. If the drive is not specified then the current drive is mapped. A list of filenames and locked status is produced with the start sector and length in sectors of each particular file.

Let us suppose that two files "BOOGIE" and "MINUET" are saved on disk. "BOOGIE" is &1E sectors in length and "MINUET" is &11 sectors is length.

*MAP will produce:

```
      $.MINUET      002 011
      $.BOOGIE      013 01E
        Free space  031 2EF

      Free sectors 2EF
```

The first column of figures indicates the start sector. Notice that "MINUET" begins at sector 2 because sectors 0 and 1 are occupied by the catalogue. The second column of figures indicates the length of the file. To gain familiarity with these figures, try to verify them by calculating the next sector after "MINUET". Starting at sector 2 + length (&11) = &13. This is the start sector for "BOOGIE" as can be seen above. &13 + &1E = &31 which is the next sector after "BOOGIE". This is shown above by "Free space" and also shows that there are &2EF sectors empty. Finally, *MAP shows the number of free sectors.

```
      $.BOOGIE      000 01E
      $.MINUET      01E 011
        Free space  02F 0F1

      Free sectors 0F1
```

CHAPTER EIGHT

SEDFS DISK STRUCTURE

SEDFS Disk Structure :-

Sectors 0,1	Catalogue of files
Sector 2 onwards	The files themselves

Catalogue structure

A catalogue occupies two sectors and takes the form:

First sector

Bytes	
&00 - &07	First eight bytes of the file TITLE
&08 - &0E	First filename
&0F	Directory of first filename (bit 7 = 1 if locked)
&10 - &17	Second filename
&18 - &1F	Third filename
	... continuing for 31 files

Second sector

Bytes		
&00 - &03	Last four bytes of disk TITLE	
&04	Cycle number	
&05	Number of filenames in the catalogue (x 8)	
&06	Bits 0,1	Number of sectors referenced by the catalogue (2 high order bits of 10 bit number).
	Bits 4,5	Start up option (BOOT)
&07	Number of sectors referenced by the catalogue (18 low order bits of 10 bit number).	

File address and lengths

&08	Load address		8 low order bits
&09	Load address		8 middle order bits
&0A	Execution address		8 low order bits
&0B	Execution address		8 middle order bits
&0C	File length		8 low order bits
&0D	File length		8 middle order bits
&0E	bits 0,1	File start sector	2 high order bits
	bits 2,3	Load address	2 high order bits
	bits 4,5	File length	2 high order bits
	bits 6,7	Execution address	2 high order bits
&0F	File start sector		8 low order bits

File addresses and lengths are repeated for each of the 31 files possible.

SEDFS I/O Port Definitions for WD 1793

	Read		Write
&FC90	Status register		Command register
&FC91	Track register		Track register
&FC92	Sector register		Sector register
&FC93	Data register		Data register
&FC94	Control port	Bit 7	
		Bit 6	
		Bit 5	Head load
		Bit 4	Motor on
		Bit 3	Density...1=FM , 0=MFM
		Bit 2	Drive Select 1
		Bit 1	Drive Select 0
		Bit 0	Side

&FC90 Status port

The result of an FDC operation is returned in the status register. Certain bits are extracted from this result which may be displayed by the command *FDCSTAT in the form of a hexadecimal number. The format of the byte is:

Bit	
7	Drive not ready (NOT displayed by FDCSTAT)
6	Write protect only on a write operation
5	RECORD type = 1 if read deleted data
4	RECORD NOT FOUND (RNF) - sector completely lost
3	CRC error - bad sector
2	n/a
1	Data request (NOT displayed by FDCSTAT)
0	Command busy (NOT displayed by FDCSTAT)

RNF

The error is not recoverable and the whole track would need formatting. For most people the easiest solution is to copy as much as possible to another disk and reformat the whole disk.

CRC (Cycle Redundancy Check)

If the disk is written to, this error will most likely be corrected. A Disk Editor will correct any errors if possible.

CHAPTER NINE

OPERATING SYSTEM ROUTINES

All file operations in BASIC use Operating System routines. When writing in assembly language, the friendliness of BASIC is not available and the Operating System routines must be accessed directly. This can be a little tedious at times due to the amount of data which has to be provided for the routines to work on but some facilities are opened up which are not accessible from BASIC. Below is a list of BASIC routines against the equivalent Operating System routine.

LOAD SAVE *LOAD *SAVE	OSFILE	
OPENIN OPENOUT OPENUP CLOSE#	OSFIND	
BGET# BPUT#	OSBGET OSBPUT	File must first be opened using OSFIND.
	OSGBPB	Reads or writes groups of bytes.
PTR# } EXT# }	OSARGS	Also does much more than BASIC.

OSFILE

Call address &FFDD Indirect address &212

This routine is used for the following operations on whole files:

1. Loading into memory
2. Saving from memory
3. Writing attributes ..... Load Address
   Execution Address
   Length
   Locked Status

The routine is accessed using the Call address. This can be done from BASIC using the command CALL &FFDD or from Assembly with the command JSR &FFDD. Before we can call the routine, a block of memory is set aside into which all the data required by OSFILE is entered. This section of memory is called a "Parameter Block". The parameter block must contain 18 bytes and the file information should be laid out in the following manner:

Byte	
	Address of filename terminated by &0D
00	
01	
	Load address of file
02	
03	
04	
05	
	Execution address of file
06	
07	
08	
09	
	For SAVE - Start address of memory block For LOAD - Length of file to read
0A	
0B	
0C	
0D	
	For SAVE - Byte after the last address of memory block For LOAD - File alterations - Locked Status
0E	
0F	
10	
11	

There are a number of ways to set up the parameter block and possibly the easiest to understand is from BASIC using the indirection operators. First reserve a section of memory using:

```
DIM BLOCK 17
```

This reserves a block of 18 bytes (0 to 17) starting at the address contained in the variable BLOCK. You can test this with the following lines:

```
10 DIM BLOCK 17
20 PRINT ~BLOCK
```

If page is set to &E00 as it should be on SEDFS then the program will give &29 as the address of BLOCK. It is situated at the end of the program. We write values to BLOCK by specifying the address with BLOCK + (relevant number of bytes). For example, the address of the filename must go into BLOCK+0, the load address of the file must go into BLOCK+2 etc. Now let us look at the parameter block in more detail.

Bytes 00 - 01

The user must write the filename to a specify memory location and write this location to the parameter block. The filename must be terminated with &0D. Take the filename "STEVE" and place it in location &2000. Using indirection operators we could do this in one of two ways.

1. Using the query indirection operator to write single bytes. The following program works but remember to add &0D to the end of the string.

Example 10.1

```
10 BASE=&2000
20 BASE?0=ASC("S")
30 BASE?1=ASC("T")
40 BASE?2=ASC("E")
50 BASE?3=ASC("V")
60 BASE?4=ASC("E")
70 BASE?5=&0D
```

This method is adequate but rather long-winded. An easier solution is to use the string indirection operator.

2. The string indirection operator writes the whole word with &0D automatically terminating it in one go:

```
20 BASE=&2000
30 &BASE="STEVE"
```

So now be have the filename in memory at address &2000. To insert the address &2000 into the parameter block is not an easy matter.

The low byte of the address must be entered first. This seems a little backward but is standard 6502 practise. As two digits in a hexadecimal number = 1 byte and the low byte is on the right then &2000 will be stored as:

```
&00   Low  byte
&20   High byte
```

So our program to set up the parameter block should look like this:

Example 10.2

```
10 DIM BLOCK 17
20 BASE=&2000
30 $BASE="STEVE"
40 BLOCK?0=&00
50 BLOCK?1=&20
```

Bytes 02 - 05

The four next sections of the parameter block are each four bytes long so they can be written using the pling indirection operator. The load address of the file is the location where it will be placed on loading back into the memory. For BASIC programs this will be PAGE. In our case this will normally be &E00. Add the following to the program:

```
60 BLOCK!2=&E00
```

Bytes 06 - 09

In a BASIC program the execution address will be the same as the load address:

```
70 BLOCK!6=&E00
```

Bytes 0A - 0D

This has two functions depending on whether the operation is to save or load. On saving, these bytes must contain the start address of the memory block we are saving. We will save the program we are currently writing. i.e. it will save itself therefore if the start address is PAGE then the instruction will be:

```
80 BLOCK!&A=&E00
```

Bytes 0E - 11

On saving these bytes contain the first free address after the section of memory to be saved. For a BASIC program this is given by TOP. For example:

```
PRINT~TOP
```

As the value of TOP will not be known until the program is complete, the variable itself has to be entered rather than its value:

```
 90 BLOCK!&E=TOP
100 BLOCK?&10=0
110 BLOCK?&11=0
```

If a file is to be saved, only the first two bytes of this four byte group are used so the last two are loaded with 0 to ensure that they are clear. Hence the need for lines 100 and 110.

The other function for these bytes is to contain the Locked Status. If changing the locked status, all the bytes are occupied but only the first needs to be changed by the programmer. In this byte, if bits 1 and 3 or both 1 and 3 contain 1 then the file is locked. They must both contain 0 for the file to be unlocked. So, to lock the file you could use:

```
BLOCK?&11=2
```

To unlock the file, use:

```
BLOCK?&11=0
```

We now have the completed parameter block and we have to inform the operating system of the address of the parameter block and what action to take.

The X and Y registers are used to pass the address of the parameter block to the routine. The X register should contain the low byte of the parameter block and the Y register should contain the high byte. The high and low bytes can be separated by using BASIC's MOD and DIV functions and the values can be passed to the registers in the variables X% and Y%. Add to the program:

```
120 X%=BLOCK MOD256
130 Y%=BLOCK DIV256
```

The A register contains the type of action to be taken and can be set from BASIC by using the variable A%. To save a file, A must = 0 so add:

```
      140 A%=0
```

Now that the information is set up, all we need to do is CALL OSFILE. This is accessed through location &FFDD so:

```
      150 CALL &FFDD
```

The completed program should look like this:

Example 10.3

```
       10 DIM BLOCK 17
       20 BASE=&2000
       30 $BASE="STEVE"
       40 BLOCK?0=&00
       50 BLOCK?1=&20
       60 BLOCK!2=&E00
       70 BLOCK!6=&E00
       80 BLOCK!&A=&E00
       90 BLOCK!&E=TOP
      100 BLOCK?&10=00
      110 BLOCK?&11=00
      120 X%=BLOCK MOD256
      130 Y%=BLOCK DIV256
      140 A%=0
      150 CALL &FFDD
```

On running it will save itself.

The type of action taken is decided by the contents of the A register thus:

A=0	Save a block of memory as a file. File information is provided in the parameter block.
A=1	Write the load and execution address in the parameter block to the catalogue entry of the existing file.
A=2	Write the load address only in the parameter block to the catalogue entry of the existing file.
A=3	Write the execution address only in the parameter block to the catalogue entry of the existing file.
A=4	Write the locked status to the existing file.
A=5	Reads the file's catalogue information into the parameter block. On completion, the A register will contain 1 if the file was found. If the file was not found it will contain 0.
A=6	Delete the named file.
A=&FF	Load the named file. If the first byte in the execution address in the parameter block is <>0, the file will be loaded to the load address in the catalogue. If this byte = 0 then the load address in the parameter block is used.

With all this information you should be able to load, save and read and write file information. It may seem a little long-winded but in machine code it is very fast. Also files, and indeed any block of data, can be loaded anywhere in memory.

The next set of Operating System file routines depend on a file being opened before they can be used. To open and close files, OSFIND is called.

OSFIND

Call address &FFCE

This routine opens and closes files. Up to five files may be open at any one time. When a file is opened, it is assigned a channel number by the Operating System so that it can be identified for further operations. In the disk filing system the channel numbers are 17 to 21 for files 1 to 5 respectively.

As with OSFILE, the A register determines the action to be taken:

A=0	Close the file whose channel number is contained in Y. Close all files if Y=0.
A<>0	Open the file whose name is pointed to by the address in X and Y. A=&40 open the file for input (reading). A=&80 open the file for output (writing). A=&C0 open the file for input and output (random access).

As with OPENOUT when A = &80, opening a file will create a new file and if a file of the same name already exists on the current drive and directory it will be overwritten.

On exit from the routine, X and Y are unchanged. A is unchanged on closing but on opening will contain the channel number. If A = 0 after attempting to open then the file count not be opened.

Example 10.4 To open (create) the file "AILEEN"

```
10 BASE=&2000
20 $BASE="AILEEN"
30 X%=&00
40 Y%=&20
50 A%=&80
60 CALL &FFCE
70 CHANNEL%=A%
```

Example 10.5 To open the file "AILEEN" for input

```
10 BASE=&2000
20 $BASE="AILEEN"
30 X%=&00
40 Y%=&20
50 A%=&40
60 CALL &FFCE
70 CHANNEL%=A%
```

Example 10.6 To open the file "AILEEN" for input and output

```
10 BASE=&2000
20 $BASE="AILEEN"
30 X%=&00
40 Y%=&20
50 A%=&C0
60 CALL &FFCE
70 CHANNEL%=A%
```

To close the file "AILEEN" the channel number contained in the variable CHANNEL% in the previous examples must be transferred to the Y register.

Example 10.7 To close any file

```
10 Y%=CHANNEL%
20 A%=0
30 CALL &FFCE
```

or. to close all files:

Example 10.8

```
10 Y%=0
20 A%=0
30 CALL &FFCE
```

OSARGS

Call address &FFDA

This routine reads an open file's attributes. The most frequently used attributes is most likely to be the sequential pointer.

On entry, X points to a four byte parameter block in zero page. The most obvious place to locate this block is from &70 - &8F which is set aside for use by the user. On read operations this block is set up by OSARGS but on the only write operation (write the sequential pointer) the user must supply the information.

Y contains the channel number as provided by OSFIND or:

Y contains 0
A specifies the type of operation

If Y=0 and A=0 the current filing system is returned in A. On exit, A will contain a number from 0 to 6 depending on the filing system you are using. As you have SEDFS, A will contain 4 but for completeness the list is:

0	No filing system
1	Cassette filing system at 1200 baud
2	Cassette filing system at 3000 baud
3	ROM filing system
4	Disk filing system
5	Econet filing system
6	Telesoftware system

If Y=0 and A=1 the address of the rest of the command line is placed in the zero page parameter block pointed to by X. By examining the address the parameters passed with *RUN can be obtained.

If Y=0 and A=&FF all files will be updated to the disk so that the current contents of the file buffer will be saved.

If Y contains the channel number and A=0, read the sequential pointer. If Y contains the channel number A=1 write the sequential pointer.

If Y contains the channel number and A=2 read the length of the file. If Y contains the channel number and A=&FF the file whose channel number is in Y will be updated to the disk.

The state of the registers after an OSARGS call will be:

X and Y	Unaltered
A	When entry is with A=0 and Y=0, the current filing system is returned else A is unaltered.
CN,V,Z	Undefined. D=0

OSBGET

Call address &FFD7

This read a byte from an open file. On entry Y must contain the channel number as provided by OSFIND. The byte which is read is determined by the position of the sequential pointer.

On exit:

X and Y	Unchanged
A	Contains the byte read
C	Set if the end of the file has been reached. If so then the byte in A is invalid.
N,V,Z	Undefined

OSBPUT

Call address &FFD4

This writes a single byte to an open file. On entry, Y must contain the channel number as provided by OSFIND. A is loaded with the byte to be written. The sequential pointer determines the place in the file where the byte in A is to be written.

On exit:

X, Y and A	Unaltered
C,N,V,Z	Undefined

OSGBPB

Call address &FFD1

This reads or writes groups of bytes to or from an open file and has the facility to transfer filing system information. A parameter block must be set which is pointed to by X and Y. The parameter block format is:

Bytes	
00	Channel number
01	Pointer to data in the I/O or tube processor
02	
03	
04	
05	Number of bytes to transfer
06	
07	
08	
09	Value of sequential pointer used on transfer
0A	
0B	
0C	

A defines the type of action to be taken.

A=1 Write bytes to disk using the new sequential pointer.
A=2 Write bytes to disk using the old sequential pointer.
A=3 Read bytes from disk using the new sequential pointer.
A=4 Read bytes from disk using the old sequential pointer.
A=5 Read the disk title and auto boot option.

The returned data takes the form:

Length of the title in one byte.
Title itself in ASCII values.
Start up option in one byte.

A=6 Read the current directory and drive.

The returned data takes the form:

Length of drive number identity in one byte.
Drive number in ASCII characters.
Length of directory name in one byte.
Directory name in ASCII characters.

A=7 Read the current library and drive. (The format of the returned information is the same as for A=6).

A=8 Read the file names from the current drive and directory. On each read of a filename, the control block is modified so that the next filename can be read. The channel number byte contains the cycle number. The sequential pointer is adjusted to point to the next filename.

On entry:

05	Number of filenames to transfer.
06	
07	
08	

OSFSC

(No direct call address)

Indirected through &21E. This is used from various filing system control functions. On entry:

A=0 Performs a *OPT command with X and Y containing the two parameters.

A=1 Check for EOF. On entry X must contain the channel number of the open file being checked and on exit X=&FF if EOF has been reached else X=0.

A=2 The same as for A=4.

A=3 For unrecognised commands. The filing system will attempt to RUN any unrecognised command and if not able to do so quickly a "Bad Command" message will be issued.

A=4 *RUN the file pointed to by X and Y. The current drive and library will be searched first and if the file is not found then the current library will be searched. If the file is found, it will be loaded according to its LOAD address and run from its EXECUTION address.

A=5 This is used on *XCAT to produce the catalogue. The remainder of the command line is pointed to by X and Y for any parameters.

A=6 Used when changing filing systems.

A=7 This returns the range of channel numbers used by SEDFS. On exit X contains the lowest number and Y contains the highest number. The range for SEDFS is 17 to 21.

A=8 This is used by SEDFS to ensure that *ENABLE precedes dangerous commands.

On exit:

All registers undefined.

On the first call:

09	Sequential pointer should = 0
0A	
0B	
0C	

The returned data takes the form:

Length of filename 1
Filename 1
Length of filename 2
Filename 2

On exit:

X,Y,A	Unaltered
N,V,Z	Undefined
C	Is set if the transfer would not be completed (if there are no more filenames or the end of the file had been reached). The number of bytes or names that failed to be transferred are written to the parameter block in bytes 05 - 08

OSWORD

To gain greater BBC compatibility, SEDFS supports three OSWORD calls which are defined as follows:

OSWORD &7D	Read number of times disk has been written to
OSWORD &7E	Read number of sectors in disk catalogue
OSWORD &7F	Perform emulation of the BBC 8271 FDC direct disk access commands

OSWORD &7D

Call address &FFF1

This reads the catalogue of the current drive to ascertain the number of times the disk has been written to. The result is placed in a 1 byte address as pointed to by the X and Y registers.

On entry:

A = &7D
X = LSB of result address
Y = MSB of result address

On exit:

A, X, Y	Unchanged

OSWORD &7E

Call address &FFF1

This reads the catalogue of the current drive to ascertain the number of sectors available to the catalogue. The result is placed in a 4 byte block as pointed to by the X and Y registers.

On entry:

```
A = &7E
X = LSB of result address
Y = MSB of result address
```

On exit the result block takes the form:

```
Byte

0 = 0
1 = LSB of sector count
2 = MSB of sector count
```

OSWORD &7F

Call address &FFF1

This is used for direct disk access. Using OSWORD with A=&7F and X and Y pointing to a parameter block, the normal SEDFS can be bypassed.

On entry:

```
A = &7F
X = LSB of parameter block address
Y = MSB of parameter block address
```

On exit:

```
A, X, Y  Unchanged
```

The parameter block is slightly different depending on which command is being used. The command is specified in byte 6 of the parameter block.

The commands supported by SEDFS are:

INITIALISE	&75
SEEK	&69
READ DRIVE Status	&6C
READ/WRITE SPECIAL REGISTERS:	
READ &3D	
WRITE &3A	
READ I.D.	&5B
READ data	&53
WRITE data	&4B
VERIFY	&5F
READ data & deleted data	&57
WRITE deleted data	&4F
FORMAT track	&63

Initialise

Initialises the step rate of the floppy disk controller. The parameter block takes the form:

```
Bytes
 00      Drive number
 01 - 04 Not used
 05      Number of parameters (4)
 06      Command = &75
 07      Parameter #1 = &0D
 08      Parameter #2 Step rate (See below)
 09      Parameter #3 not used
 0A      Parameter #4 not used
```

The step rate is determined by the 2 l.s. bits:

```
 11  =  15 m.s.
 10  =  10 m.s.
 01  =   6 m.s.
 00  =   3 m.s.
```

Seek

To seek a specific track:

```
Bytes
 00      Drive number
 01 - 04 Not used
 05      Number of parameters (1)
 06      Command = &69
 07      Parameter # 1 Track address 0 - 255
```

Read drive status

To check whether or not the disk is write protected.

The parameter block is:

```
Bytes
 00      Drive number
 01 - 04 Not used
 05      Number of parameters (0)
 06      Command = &6C
 07      Result
```

The result byte takes the form:

```
 Bit
  7           0
  6           RDY1        always 1
  5           0
  4           0
  3           WR. PROT = if write protected
  2           RDY 0   always 1
  1           0
  0           0
```

Read/Write Special Registers

The parameter block is:

Bytes	
00	Drive number
01 - 04	Not used
05	Number of parameters (1 if READ...2 if WRITE)
06	Command for READ = &7D Command for WRITE = &7A
07	Register address
08	Data if write - Result if read
09	Result if write

Register 0

40 track disk in 80 track drive flag:

Bit	
7	Used by SEDFS
6	0 = normal operation 1 = perform double track stepping
5	0
4	0
3	0
2	0
1	0
0	0

Register 12

Track address

This register contains the track number which the FDC thinks it is at. This would be used for reading/writing non-standard sectors. i.e. 128 byte sectors.

READ I.D.

This command is used to read the I.D.s of sectors on a track. The ten sectors on a track starting at the Index hole may be numbered 4,5,6,7,8,9,0,1,2,3. They could be numbered 0,2,4,6,8,1,3,5,7,9 which would be called "interleaving" of sectors. Information also returned from the READ I.D. is the size of the various sectors. Formatting a track with sectors of varying sizes is a common method of "protecting" disks from being copied easily.

The parameter block is:

Bytes	
00	Drive number
01 - 04	Address for I.D. bytes C,H,R,N
05	Number of parameters
06	Command &5B
07	Track number
08	Zero
09	Number of I.D. fields to read
10	Result

A result of zero means that a sector I.D. was captured. A result of &10 means that no I.D. could be found, probably due to an unformatted disk.

The first "sector header" information (C,H,R,N) will be written to the address given in bytes 1 - 4. The bytes are:

C

When the Read/Write head of the drive is stepped to a particular track (say track 2) then normally the cylinder number of a sector in that cylinder will be 2. **However**, if the value of C returned from the READ I.D. was 1 then we must be looking at a 40 track disk in an 80 track drive because it requires two steps of an 80 track drive to cover each cylinder of a 40 track disk. Conversely, if the value returned in C which seeking track 2 and performing a READ I.D. was cylinder 4 then we must be looking at an 80 track disk in a 40 track drive.

The physical track and the cylinder number do not necessarily take on the same value. For instance, the physical number may be 1 but the cylinder number could be any number up to 255. By this means a disk may be "protected".

H

The head parameter. This is determined on format and should correspond to the side number. i.e. 0 for side (or drive) 0. However, this value can take on a value of up to 255 thereby providing a means of protecting the disk to prevent unlawful copying.

R

The record number of the sector found. Normally this will be in the range 0 to 9. For the purposes of disk protection, however, this could take on a value between 0 and 255.

N

This is a value up to 3 indicating the size of the sector. See READ data.

Sector sizes in bytes:

```
N = 0       128
N = 1       256 ..... The normal value
N = 2       512
N = 3       1024
```

Read data

To read up to the maximum number of sectors on a track. The parameter block is:

```
Bytes
 00      Drive number
 01 - 04 Address to put read data
 05      Number of parameters
 06      Command &53
 07      Track number
 08      Sector to start reading from
 09      Sector length / Number of sectors from start
 10      Result
```

Byte 9 takes the form:

```
Bit
 7 )          000 =   12 bytes
 6 ) length   001 =  256 bytes
 5 )          010 =  512 bytes
              011 = 1024 bytes
 4 )
 3 )
 2 ) number of sectors (0-9)
 1 )
 0 ) This value must be <= (Max sector - start sector)
```

For instance, a value of &21 in byte 9 will read one 256 byte sector and a value of &2A will read 10 sectors.

Example 10.9 to read a sector from Track T%, sector S%

```
 10DIM OSWDBK 16                  Allocate 16 bytes for
                                  OSWORD block
 20T%=0                           Track zero
 30S%=0                           Read from sector zero
 40?(OSWDBK+0)=0                  Drive 0
 50!(OSWDBK+1)=&2000              Read into memory
 55?(OSWDBK+5)=3                  at &2000
 60?(OSWDBK+6)=&53                Read command
 70?(OSWDBK+7)=T%                 Set track to T%
 80?(OSWDBK+8)=S%                 Set start sector to S%
 90 ?(OSWDBK+9)=&21               Read 1 x 256 byte sector
100A%=&7F
110X%=OSWDBK MOD 256              Point X% and Y% to
                                  OSWORD block
120Y%=OSWDBK DIV 256
130CALL &FFF1                     Perform actual OSWORD
                                  operation
 140IF ?(OSWDBK+10)<>0 THEN       Check result for good
PRINT"Error"                      read
```

Write data

The parameter block takes the form:

```
Bytes
 00      Drive number
 01 - 04 Address from which to write data
 05      Number of parameters
 06      Command &4B
```

```
 07      Track number
 08      Start sector
 09      Sector length / number of sectors
 10      Result
```

Verify data

The parameter block takes the form:

```
Bytes
 00      Drive number
 01 - 04 Not used
 05      Number of parameters
 06      Command &5F
 07      Track number
 08      Start sector
 09      Sector data/ length etc.
 10      Result
```

Read data and deleted data

To read sectors of data and deleted data from disk. The parameter block takes the form:

```
Bytes
 00      Drive number
 01 - 04 Address to put data
 05      Number of parameters
 06      Command &57
 07      Track number
 08      Start sector number
 09      Sector data
 10      Result
```

Write deleted data

To write sectors with deleted data address mark. The parameter block takes the form:

```
Bytes
 00      Drive number
 01 - 04 Address from which to write data
 05      Number of parameters
 06      Command &4F
 07      Track number
 08      Start sector number
 09      Sector data
 10      Data
```

Format track

The parameter block takes the form:

```
Bytes
 00      Drive number
 01 - 04 Address of format data
 05      Number of parameters
 06      Command &63
```

```
07      Track to format
08 - 11 Not used
12      Result
```

Non-standard formats can be used and are supported by the READ and WRITE commands.

Format Track

The 8271 Format command is supported by SEDFS so that commercial software using this facility is compatible. The following is an example of a Format routine using OSWORD &7F.

Example 10.10

```
 10?&2000=0                          Drive 0
 20!&2001=&3000                      data starts at &3000
 60?&2005=5
 70?&2006=&63                        format command
 80?&2007=0
 90?&2008=&0
100?&2009=&21
110
120FORI%=0TO9*4 STEP 4               create sector list
130I%?&3001=0
140I%?&3002=I%/4
150I%?&3003=1
160NEXT
170
180X%=0                             Point X% and Y% to
                                    OSWORD
190Y%=&20                           control block at
                                    &2000
200A%=&7F                           OSWORD 7F
210
220
230FORT%=0TO79                      80 tracks
240FORI%=0TO9*4STEP4:I%?&3000=T%:   put 'C' in sector list
NEXT
250PRINT"Track ";T%
260?&2007=T%
270CALL&FFF1                        do actual format
280NEXT                             repeat for 80 tracks
290
300?&2006=&4B                       write sector command
310?&2007=&0                        track 0
320?&2008=&0                        sector 0
330?&2009=&22                       SSSNNNNN S=1 and N=2
                                    sectors
340FORI=0TO511:I?&3000=0:NEXT
350?&3106=T%*10 DIV 256             Catalogue size m.s.
360?&3107=T%*10 MOD 256             Catalogue size l.s.
370CALL&FFF1                        Write catalogue to disk
```

CHAPTER TEN

ERROR REPORTING

SEDFS error messages are in addition to those already present on the Acorn Electron which are fully explained in the Acorn Electron User Guide.

If commands are entered in command mode then any errors are reported on the screen. If SEDFS is controlled via a program then errors are handled in the normal Acorn manner by reporting them on the screen with a message or through any user-written error handling routine.

The following list gives the error message which would be displayed if in command mode or REPORT was used in the error handling routine and the error number which would be displayed if ERR was used in the error handling routine.

Number		Message	
Hex	Decimal		
BD	189	Not Enabled	Issued when a *ENABLE was not entered before a command which required *ENABLE.
BE	190	Catalogue Full	Issued when the catalogue contains 31 files and an attempt was made to save another file to the catalogue.
BF	191	Can't Extend	Imagine that a file A has been saved on a disk. Saving a different file (B) after this will result in file B occupying the next free space on the disk after file A. Any attempt to increase the length of file A results in the error message.
CO	192	Too many files open	A maximum of five files may be open at any one time and any attempt to open a sixth will result in this error.
C1	193	File read only	This error is produced by attempting to write to a file that has been opened for read only.
C2	194	File open	Any attempt to re-open a file that is already open will result in this error. This most often occurs when a program has been terminated abruptly (e.g. <ESCAPE>) without *closing* an opened file. To get back into

			the file it will need to be closed first.
C3	195	File locked	Displayed when an attempt is made to delete or write to a locked file.
C4	196	File exists	This error is caused by an attempt to rename a file to a name which already exists on the specified directory.
C6	198	Disk full	When an attempt is made to OPENOUT or save a file to a disk which has insufficient space.
C7	199	Disk fault	Failure to read disk due to corrupted data, the wrong density or damaged disk, etc.
C8	200	Disk changed	If you change the disk when one or more files are open then this error will be reported.
C9	201	Disk read only	Produced by an attempt to write to a disk which has been write protected.
CA	202	Bad sum	Not implemented.
CB	203	Bad option	Only *OPT options 1 and 4 are valid.
CC	204	Bad filename	You have tried to use a filename which is too long or contains illegal characters.
CD	205	Bad drive	Drive numbers must be in the range 0 to 3.
D6	214	File not found	An attempt to read a file which does not exist will produce this error.
DC	220	Syntax error	The command was recognised but the form was wrong.
DE	221	Channel	If reference is made to a channel with an incorrect number then this will work.
DF	222	EOF	End Of File. If an attempt is made to read beyond, the point the error is generated.
FE	254	Bad Command	If SEDFS fails to recognised a command as valid or as a file in the library then the error is generated.

Disk faults

00	0	No errors	
08	8	CRC error	Data was previously written badly and is now corrupted. The sector may be recovered using a Disk Editor.
10	16	RNF (Record Not Found)	The sector requested **cannot** be found due to a corrupted disk, wrong track, etc. This sector cannot be recovered if corrupted and the whole track must be reformatted.
18	24	Combination of error 8 and 10	
40	64	Disk Read Only	At attempt has been made to write to a write-protected disk.

Index