Advanced Disc Filing System

PRES

PRES

ADFS

Advanced Disc Filing System
User
Guide

All correspondence should be addressed to:

PRES
6 Ava House
High Street
Chobham
Surrey GU24 8LZ (Tel. 0276 72046)

First published by Acorn Computers Ltd. 1984
This revised edition first published by P.R.E.S. Ltd. 1987

CONTENTS

1. INTRODUCTION

This User Guide describes in detail the use of the Advanced Disc Filing System - ADFS for short. The Guide is broadly in two sections, firstly the facililties ADFS provides and secondly more detailed reference information for the more advanced user.

Chapter 2 describes the ADFS commands in detail. You are advised to read the first four sections to understand the basic concepts of the filing system. The rest of the chapter may then be read in the order in which it is presented, or may be consulted from time to time, when a detailed description of a particular command is required.

Chapter 3 describes how the utility programs of the Welcome disc are used, and how to obtain information on the utilities. All of the utilities are documented on disc and this information may be printed out using the *UTILS program described in chapter 4.

Chapter 4 gives information about using the ADFS facilities from BASIC. The statements used are the same as those available when using tape; however the ADFS provides more versatility in the way in which they can be used. The section on sequential files only need be read if you are going to use BASIC data files (as opposed to program files).

Chapter 5 is the first chapter in the reference section. All chapters in this part of the book are designed to be consulted only when a particular item, eg the syntax of a * command, has to be looked up. Chapter 5 gives a list of the ADFS * commands in alphabetical order.

Chapter 6 describes the ADFS machine code entry points and will be of interest only to programmers using assembly language, or very advanced BASIC programmers. In addition to the six filing system calls, details are given on OSCLI, OSBYTE and OSWORD in connection with filing systems.

Chapter 7 lists the ADFS error messages in alphabetical and numerical order. An explanation of possible causes of each error is given.

Chapter 8 gives technical information about the way in which data is stored on the disc. It is provided mainly for interest's sake, but will also be of use to people trying to fix discs that have become corrupted in some way.

Appendix A contains trouble-shooting information: what to do when either the hardware or software of the Plus 3 does not appear to work.

Appendix B describes the various ways in which the ADFS may be called up.

Appendix C is general information on the Welcome disc utilities, using ADFS in modes Ø-3, how to disable ADFS and the various methods of instalation

2. THE ADFS COMMANDS

One of the most used ADFS commands is *CAT, which may be abbreviated to *. This is just one of the many commands that are provided to help you manage your files on discs. Some of the commands will be familiar to most users as they are also available when using the cassette filing system (CFS). These are: *CAT (*.), *EXEC, *HELP, *LOAD, *SAVE, *RUN (*/) and *SPOOL, and are all discussed with respect to the ADFS in this chapter.

This chapter is organised in two parts. The first four sections give general information which is required in order to understand what the commands do. These sections should be read by everyone using the ADFS for the first time.

The last three sections of the chapter describe the commands themselves. The three sections cover directory oriented commands, normal file commands and commands affecting the whole disc respectively. Between them, the sections cover all of the * commands which the ADFS responds to (and even some non-ADFS commands).

The Index at the back of this User Guide is arranged so that all of the command names (those prefixed with an '*') appear indented. Also the main page reference containing the more detailed description of the command appears first

The sections on the * commands are rather 'dense' in their information content. In order to provide a gentler first introduction to the ADFS command, brief descriptions of each command are given below:

To call the ADFS up from another filing system, say tape, either *ADFS or *FADFS may be used. The latter has the same effect as pressing <BREAK> when the ADFS is the highest-priority filing system, and the former is similar to *FADFS followed by a *MOUNT command.

Before a disc may be used with the ADFS, it must be mounted. This is done automatically when the filing system is entered by <CTRL> A <BREAK> or <SHIFT> <BREAK>, but not by *FADFS or F <BREAK>. Mounting a disc simply means reading some important information from it, so the ADFS knows where to put files etc. The command *MOUNT is used to perform the operation. Similarly, before a disc is removed, it should be dismounted (using *DISMOUNT) so that the ADFS can perform various 'house-keeping' operations on the disc.

Areas of memory may be saved using *SAVE. This is useful for storing machine code programs or screen images on the disc. If a second processor is fitted, memory from this or the IO processor may be saved by specifying the 'high-order' address. Programs and data may be loaded into memory (at a given location, if required) using *LOAD. A machine code program may be executed using *RUN.

Files may be deleted using *DELETE or *REMOVE to delete a single file, or *DESTROY to delete several files in one go. Directories may also be deleted, but only if they are empty (contain no references to other files).

To copy a file on to another part of the disc (or on to a different disc), the *COPY command may be used. This may also copy several files at once. A file's name may be changed using the *RENAME command. This may also be used to move the file into a different directory.

The commands *SPOOL and *EXEC are actually Operating System commands and are described in the Acorn Electron User Guide. The *CLOSE command closes all open files, and has exactly the same effect on CLOSE#0 in BASIC.

Directories play an important role when using the ADFS, so there are several commands to support them. A new directory may be created using *CDIR, and the current directory may be selected using *DIR. *BACK reverts back to the directory before the current one. The library directory (where the ADFS looks for commands) may be set using *LIB.

The contents of a directory may be printed in summary using *CAT, and greater information is given by *INFO and *EX . The commands *LCAT and *LEX provide a shorthand way of obtaining information about the library directory. A directory may be given a long, meaningful name by using the *TITLE command.

File access permissions are controlled by the *ACCESS command. Permissions are 'R' for read, 'W' for write, 'L' for locked and 'E' for execute-only.

Various commands deal with the disc as a whole. *FREE and *MAP print information about the amount of disc space that has been used and how many areas of free space there are, and *COMPACT is used to reclaim areas of free space (which were occupied by deleted files, for example).

The *OPT command is used to control two aspects of the ADFS: what happens when <SHIFT> <BREAK> is pressed when selecting the ADFS (the so-called auto-boot option controlled by *OPT4), and whether messages are printed every time a file is accessed (*OPT1).

Finally *HELP ADFS (actually a MOS command) may be used to print a summary of the ADFS commands' syntax.

Selecting filing systems

One of the most important facilities that the * commands provide is the ability to switch between filing systems. The ADFS will normally be selected automatically when the machine is switched on, or when <BREAK> is pressed. However, it is sometimes desirable to be able to select a filing system without having to reset the whole machine. To this end, several commands are provided to call filing systems:

*ADFS	Start up the ADFS and reset it to previous state
*FADFS	Start up the ADFS 'quietly', without accessing the drive
*TAPE	Start up the cassette filing system at 1200 baud
*ROM	Start up the ROM filing system

All of these commands may be used on an Electron running ADFS

The action of *ADFS is to select the ADFS and restore the CSD and CSL to the same as the last time ADFS was selected. If the ADFS has not been selected since the last <CTRL> <BREAK> or power on, both directories will be 'Unset' and the drive will not be accessed. Otherwise, the drive will be accessed in order to load the previous directory.

The action of the *FADFS is to select the ADFS without accessing the drive. As a consequence CSD and CSL will be 'Unset'.

There are many ways in which the ADFS may be selected. The exact action taken depends on which of <SHIFT>, <CTRL>, A and F are pressed in combination with <BREAK>, and whether the ADFS has been used since the last hard reset. Appendix B describes the effect of all possible combinations. The most useful ways of starting the ADFS are:

- Power on or <CTRL> <BREAK> to ensure that the ADFS is in its initial state.

- <CTRL> A <BREAK> to reset the ADFS completely and set the CSD and library directories correctly. If there is a file on the disc with pathname $.LIB* this will be used as the library, otherwise '$' will be selected.

- <SHIFT> <BREAK> to execute the disc's auto-boot file.

- Just <BREAK> to escape from a crashed program and restore the ADFS to its previous state if possible.

When you first start using your Plus 3 you will probably want to transfer some programs from tape on to disc. To do this you will have to switch between filing systems. For example, to copy the BASIC program ' MyProg ' to disc, this sequence of commands will be required:

```
>*TAPE <RETURN>

>LOAD "MyProg" <RETURN>

>*ADFS <RETURN>

>SAVE "MyProg" <RETURN>
```

Of course, you could save the program using a different name from that used when it was loaded.

Files and directories

Before we get into the detailed descriptions of ADFS * commands, it is necessary to describe some of the important concepts. This section describes general files and directories.

A file is just a sequence of bytes (characters) that happens to be stored on a file medium (in the present case, a disc) instead of in the computer's memory. The advantages of using an external medium to store information are clear even from using cassettes: the data isn't lost when the computer is switched off, and you can store more information in the file than the computer's memory can hold at once.

The length of a file may be from zero bytes, an 'empty' file, to a limit set by the filing system in use. For the ADFS this is about 328,000 (or 656,000 for double-sided discs) bytes, which should be long enough for most purposes. In order to distinguish between files you must name them. We have already seen an example of this, saving a BASIC program with a command such as:

```
SAVE "MyProg"
```

A filename may contain between one and ten characters. Characters that may be used include all of the upper and lower case letters and the digits (though the ADFS treats 'a' and 'A' as the same letter when searching for files). Certain punctuation symbols may also be used in filenames, but because many of these have a special meaning to the ADFS it is best to avoid them for now.

What the file contains, ie the meaning of the bytes that comprise it, is of no interest to the ADFS. When you get a catalogue of a disc using *CAT, you can't tell if a file contains machine code, a BASIC program, pure text, or some other type of data. The ADFS will let you load a text file as BASIC if you want to, but BASIC will object and give a 'Bad program' error. One use of directories is to avoid this sort of confusion by grouping files of similar types together.

You may be familiar with the ability to use a null filename when loading a file under the cassette filing system. For example

```
CHAIN ""
```

will load and run the next program on the tape. Under the ADFS you are not allowed to use null filenames like this, but in some circumstances you can use a * instead:

```
CHAIN "*"
```

will load and run the first file in the current directory. The current directory is listed using the *CAT command, and as the entries are held in alphabetical order it is fairly easy to work out which file will be accessed. This is a use of the ADFS's 'wildcard' facility which is described below.

The term 'directory' has been used a couple of times now. A directory is simply a file which contains information about other files. Usually, the files in a given directory are related in some way, eg all the files might be utility programs, or VIEW word processor files. When a disc is formatted (which needn't concern us here as the Welcome disc comes ready formatted), a single directory is created. This is called the root directory. Since directories are simply files, they are given names. The root directory is called '$'.

Directories are useful in several ways. Creating a new directory effectively gives you a 'disc within a disc'. For example, say five members in a group (perhaps in a class at school) were sharing a disc. Directories could be created on the disc for each of the five members of the group (using commands such as *CDIR Jim, *CDIR Jane) and then each person could save his or her programs in the appropriate directory. This would prevent Jim accidentally deleting one of Jane's files, because files in different directories are totally separate, even if they have the same name.

Another use for directories, as mentioned above, is grouping related files together. A disc might contain a directory for holding BASIC programs, another for machine code source programs, and yet another for machine code object programs, or utilities. In fact the Welcome disc does just that: all of the BASIC welcome programs are in a directory called 'WELCOME' and the utility programs are all grouped together in 'LIBRARY'. Using directories in this way makes it much easier to keep track of what programs are on the disc.

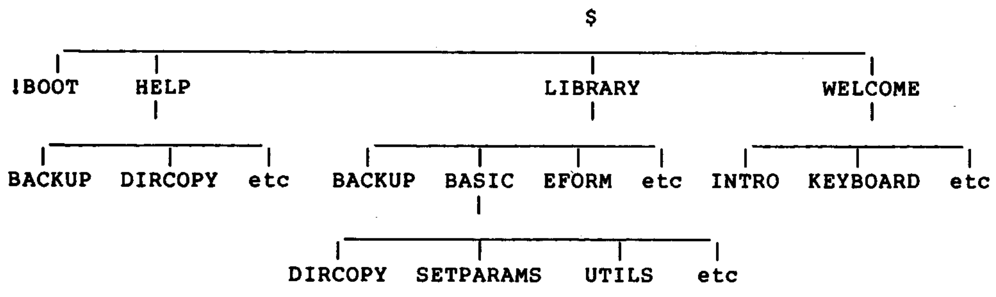
You can distinguish directories in catalogue listings by the letter 'D' after the directory's name. This is the only thing that distinguishes a directory from normal files from the user's point of view. However, certain commands which may be applied to normal files, such as *RUN, may not be used on directories, and the ADFS will produce an error message.

Directory files are always the same length (1280 bytes) and contain information about other files (sometimes known as the directory's 'children'). The information stored in each file includes its name (one to ten characters); its 'access' permissions, eg can the file be deleted? is it a directory?; its length; its disc 'address' (so that the ADFS can locate it easily when trying to LOAD it, for example), and the load and execution addresses. The last two items are used by certain ADFS commands such as *LOAD and *RUN.

File hierarchies and pathnames

We have seen that a directory is in many respects a normal file that contains information about other files (up to 47 of them). This implies that directories may 'contain' other directories, and so hierarchy or tree-structure of files may be built-up. To illustrate this, we will look at the structure of the Welcome disc.

The contents of the 'WELCOME' disc are shown diagrammatically below:



At the top of the 'tree' is the root directory '$'. This is the directory that is always selected when a *MOUNT command is executed. The four files in '$' are the ones listed when *CAT is performed on the Welcome disc. The normal file '!BOOT' has no children.

The three directoies 'HELP', 'LIBRARY' and 'WELCOME' have other files beneath them. The contents of these directories may also be listed using *CAT, but an extended form must be used whereby the name of the directory to be catalogued is given after the command. The directory 'LIBRARY' has another directory within it called 'BASIC'. Examples of *CAT commands which may be used on these directories are:

```
*CAT LIBRARY
*CAT HELP
*CAT WELCOME
*CAT LIBRARY.BASIC
```

So far, files have been referenced by their simple names, eg 'MyProg'. When such a filename is given in a command the ADFS assumes you mean the file of that name which is located in the currently selected directory (CSD). Since the CSD is '$' by default, the ADFS takes 'myprog' to mean the file called 'myprog' in the directory '$'. This is written $.myprog. A filename of this kind is called a pathname as it tells the ADFS the path is should take through the tree structure to access the desired file.

Pathnames may be extended to take in as many directories as necessary. Here are some examples of pathnames for files shown in the diagram above:

$	'$', the root directory
$.!BOOT	The file '!BOOT' in directory '$'
$.WELCOME	The directory file 'WELCOME' in directory '$'
$.LIBRARY.EFORM	The file 'EFORM' in directory 'LIBRARY' in directory '$'

Since the CSD's name is assumed to prefix all filenames that do not start with a '$', the last three examples could be given as '!BOOT', 'WELCOME' and 'LIBRARY.EFORM' respectively (assuming the CSD is '$'). The first example would in theory be '' but since it is hard to detect zero characters in commands the CSD is written '@' instead when its name is required explicitly.

Wildcard characters in filenames

There is a facility in the ADFS for using 'wildcards' in filenames. These are special characters that stand for a single arbitrary character or a sequence of up to ten arbitrary characters. The single wildcard is # and the multiple match character is '*'. Examples of filenames incorporating wildcards are:

chl#	(eg 'chla', 'chlb' etc)
data##	(eg 'data01', 'dataAB' etc)

$.games.*	(any file in '$.games')
let	(any filename with 'let' in it)

How filenames incorporating wildcards are interpreted depends on the command. Most commands will act on the first file that matches the wildcard pattern. The search is done in alphabetical order. The *LOAD command works like this, so:

```
*LOAD b*
```

will load the first file in the current directory that begins with 'b' (or 'B' as upper and lower case are not distinguished when searching for files).

Other commands act on all files which match the filename. Such a command is *INFO, which gives information about files. The command:

```
*INFO book.ch*
```

will print information about all the files in directory 'book' beginning with 'ch'.

The third set of commands doesn't allow wildcards at all. The commands are usually 'dangerous' ones where use of a filename with wildcards might result in disaster, for example:

```
*DELETE *
```

if allowed, would delete all the filenames in the current directory.

Commands dealing with directories

We have now seen what is meant by the terms 'directory' and 'hierarchy'. Because directories are so important, there is a whole group of commands that affects them. The commands perform such operations as obtaining information about the files in a directory (*CAT and *INFO), changing the CSD, creating new directories, and dealing with the library directory.

Changing directories

There are three commands which are used to change between the various directories which the ADFS keeps track of.

*DIR

The *DIR command is used to change the CSD. Suppose you are using the Welcome disc, and want to access the files in the directory called 'WELCOME'. Instead of having to type long filenames such as:

```
LOAD "WELCOME.INTRO"
*CAT WELCOME
SAVE "WELCOME.TEMP"
```

you can select 'WELCOME' as the CSD so that the ADFS takes all files to be in that directory unless you tell it otherwise. The *DIR command is used to change directory, and it is followed by the pathname of the new directory, eg:

```
*DIR WELCOME
```

The command *DIR without a pathname will set the CSD to '$'.

If you have the Welcome disc in the drive, you can try some *DIRs for yourself. There are three directories on the disc, '$', 'LIBRARY' and 'WELCOME' and any of these may be selected using *DIR. To get to the root directory, any of these may be used:

```
*DIR
*DIR $
*DIR &
```

The name '&' is simply an alternative to '$'. Once you are in '$', either of the other directories may be selected thus:

```
*DIR W*
```

or

```
*DIR L*
```

Note the use of the multiple wildcard to find the first directory beginning with 'W' and 'L' respectively. If you are in '$.LIBRARY' (ie you issued the second command above), then to set '$.WELCOME' as the CSD, its full pathname must be specified, ie you must type

```
*DIR $.W*
```

rather than

```
*DIR W*
```

This is because the second form of the command will look for a directory beginning with 'W' in the CSD ('LIBRARY'). Since there is no such directory, a 'Not found' error will be given.

It is quite important when using the ADFS to keep track of the current 'context', that is what the current directory and drive are, and what the library directory is (we come to libraries and drive numbers a little later). Many unexpected errors occur because the user is in a different directory from that which he thought. The quickest way to find a summary of the current context is to perform a *CAT (or *.) command. The first three lines printed give the important information.

Sometimes you might want to move to the directory which is the parent of the current one, without giving its complete pathname. However, the tree structure on the disc can be nested as deep as you like, with directories containing directories containing directories and so on. Suppose the CSD has the pathname $.book.chapter1.section3.diagrams. If you wanted to move up a level, the command:

```
*DIR $.book.chapter1.section3
```

would have to be given. Luckily there is a convenient shorthand way of expressing this:

```
*DIR ^
```

Whenever circumflex '^' appears in a pathname it is taken to mean 'the parent of the directory so far'. The directory so far at the start of a pathname is the CSD, so just '^' means the parent of the CSD. Multiple '^'s can be used to move up more than one level, and '^' may be followed by the rest of the pathname to go back down the tree. For example, to set the CSD to $.book.chapter2.section1 (assuming the CSD is now $.book.chapter1.section3) either of these commands could be used:

```
*DIR ^.^.chapter2.section1
*DIR $.book.chapter2.section1
```

The first example moves up to '$.book' using two consecutive '^'s and the second example selects '$.book' explicitly. Both versions move down to 'section1' in the normal manner.

Using *DIR to select the drive

When a second drive is installed, we need some way of differentiating between files on the first drive and on the second one. This is done by starting pathnames with :<drv> where <drv> is the drive identifier. The first drive is usually referred to as Ø and the second drive as 1. However, various synonyms may be used: the first drive is Ø, 4, A or E and the second drive is 1, 5, B or F.

Examples of pathnames with drive specifications are:

```
*CAT :Ø.$.LIB
LOAD ":1.$.MYPROG"
```

Alternatively, a given drive may be selected permanently by specifying its number in the *DIR command:

```
*DIR :Ø
*DIR :1.$.GAMES
```

In fact, ':<drv>.' really means ':<drv>.$.', so the first example would select the CSD as '$' on drive Ø and second example could have been written:

```
*DIR :1.GAMES
```

*BACK

In addition to remembering the CSD, the ADFS also keeps track of the previously selected directory (PSD), that is the directory that was the CSD when the last *DIR command was issued. To access the PSD, the *BACK command is used. This is the same as typing *DIR <PSD> where <PSD> is the pathname of the previous directory. In addition, *BACK sets the PSD to the CSD, so typing it again will get you back to the directory before the first *BACK. An example will make it clearer. Suppose you wanted to do a lot of loading and saving of programs in both '$.WELCOME' and '$.LIBRARY', swapping between the two directories from time to time. The sequence of commands to do this might be:

```
*DIR $.WELCOME

Commands using WELCOME

*DIR ^.LIBRARY

Commands using LIBRARY

*BACK

Commands using WELCOME

*BACK

Commands using LIBRARY
```

and so on...

*LIB

The third directory that the ADFS keeps track of is the library directory (CSL). This is only used by certain specialised commands connected with running machine code programs.

There are three ways of executing a machine code program on the disc:

```
*RUN <file>
*/ <file>
*<file>
```

Examples are *RUN roms, */dump and *print. Note that the third form may only be used if the command's name is not the same as one of the built-in * commands, which are listed in chapter 5. All three variations will load the file specified and execute it. First, the ADFS prefixes the pathname given in the command with the CSD's path and searches for a file with that composite pathname. If it can't find one, it uses the CSL pathname and tries again. If the file still can't be found, an error ('Bad command') is given. To specify the directory to be searched after the CSD, the *LIB command is used:

```
*LIB $.assem.utils
```

will set the CSL to $.assem.utils. A quick way of setting the library directory to be the same as the CSD is:

```
*LIB @
```

which uses the special-character form of CSD.

When the ADFS is entered using *FADFS or an equivalent, CSD and CSL are both 'unset'. Typing *MOUNT sets the CSD to '$', but CSL remains 'unset' and must be set by *LIB. When the ADFS is entered using <CTRL> A <BREAK> (see the table in appendix B), CSD is set to '$' and CSL is either set to '$' or the first directory in '$' whose name begins with the characters 'LIB', if one exists.

The *LIB command is described in detail under *RUN.

Obtaining file information

There are several useful commands for obtaining information from the current directory, the library, or an arbitrary named directory.

*CAT

We have already used the *CAT command to list the contents of the CSD, which has been '$' so far. If you follow the command with the pathname of a directory, that directory's contents will be listed. Examples are:

*CAT $	List the root directory
*CAT &	'&' is a pseudonym for '$'
*CAT $.LIBRARY	List the directory '$.LIBRARY'
*CAT @	An alternative to *CAT
*CAT WELCOME	List WELCOME, assuming CSD is '$'

We will now look in detail at the information printed by *CAT. Suppose the following was obtained:

```
Misc disc           (13)
Drive:Ø             Option ØØ  (Off )
Dir. $              Lib. $

GAMES      DLR(1Ø)  OP         WR  (13)
TEMP       WR (11)  UTILS      DLR (Ø9)
VIEW       DLR(Ø8)
```

The first three lines give general information about the directory and ADFS, and subsequent lines give information about the individual files within the directory that was catalogued.

The first line gives the title and 'master sequence number' (MSN) of the directory. The directory title is a string of up to 19 characters that gives an indication of what the directory contains. You can set the title of the current directory using *TITLE, though by default it is set to the filename of the directory. In this example, the title is 'Misc disc'.

The master sequence number starts at zero when the directory is created. Whenever a file is saved into the directory, the MSN is increased by one. This number is stored in the file's own sequence number entry, enabling the age of a file to be guessed by comparing its sequence number to the
MSN. Sequence numbers go back to ØØ after reaching 99.

Line two of the listing gives information about the current drive. The drive number is usually Ø or 1 as described above. The 'Option' tells the ADFS what to do with a file called '$.!BOOT' if <SHIFT> is pressed at the same time as <BREAK>. Doing this causes the ADFS to 'auto-boot', so that the file '!BOOT' may be loaded, executed or treated as keyboard input. See *OPT4 for more details on auto-booting. By default, no attempt is made to use '!BOOT' when <SHIFT> <BREAK> is pressed, hence the term 'Off' in line two.

Line three of a directory listing gives the names of the currently selected directory and library directory.

The remainder of the directory listing gives the names of the files in the directory, their access permissions and their sequence numbers. Access permissions are described under the *ACCESS command at the end of this section and sequence numbers were described above.

*LCAT

A separate command is provided to do a *CAT on whatever directory is currently the library. Typing

```
*LCAT
```

will give the same effect as

```
*CAT $.LIBRARY
```

assuming that '$.LIBRARY' is the CSL directory.

*INFO

The ADFS holds more information about each file than just its name, sequence number and access permissions. The *INFO command prints this extra information. The command is followed by a pathname which may contain wildcards. If it does, information for all of the files matching the pathname is printed. Type:

```
*INFO $.*
```

If the Welcome disc is in the drive, the output might look something like this:

```
!BOOT          LWR(10)  00000000 FFFFFFFF 0000001A 00003A
HELP           DLR(05)  000045
LIBRARY        DLR(12)  000007
WELCOME        DLR(11)  00002C
```

The first part of each line is the same as that produced by *CAT: the file's name, access permissions and sequence number. The rest of the line has one of two formats, depending on whether the file is a directory or not.

Non-directory files have four items of information on the line. These are: load address, execution address, length and disc address respectively. All of the addresses are in hexadecimal. The load address tells the disc filing system where in the Electron's memory to put the file in response to a *LOAD command. The execution address is where the program will be executed from if it is *RUN. The length is simply the number of bytes that the file occupies. To obtain this in decimal, it may be printed in BASIC preceded by a '&'. For example, the length of '!BOOT' may be found using the BASIC statement:

```
PRINT &1A
```

The disc address of the file tells the ADFS where to look for it, for example after the user has typed a *LOAD command. It is the sector number of the first sector of the file. The disc address is not usually of interest to users.

The information printed for directories only consists of the disc address, as the other attributes have no meaning. (It is illegal to load or execute a directory and its length is always 1280 bytes.)

It is possible to obtain information about only one file by giving the filename after the *INFO command:

```
*INFO !BOOT
```

You can also print information about some of the contents of a directory by following the command with a suitable wildcard pathname. For example, to print information about all files beginning with 'B' in the directory '$.WELCOME' when the CSD is '$.LIBRARY', either of these might be used:

```
*INFO $.WELCOME.B*
*INFO ^.WELCOME.B*
```

It is possible to protect machine code files by making them 'execute only'. To do this, a command such as

```
*ACCESS myprog E
```

is used. Once this is done, 'myprog' can only be *RUN, not loaded or opened (it may however be deleted). If a file is protected in this way, *INFO will only print its *CAT information.

*EX and *LEX

A particularly common form of *INFO is *INFO * which prints information about all the files in the CSD. Another command is provided to obtain this effect in fewer keystrokes:

```
*EX
```

In fact, *EX prints information about all files in any directory, which defaults to CSD if absent. Thus

```
*EX book
```

prints information about all files in the directory 'book', and is equivalent to *INFO book.*. A similar command is provided to act on the CSL instead of a specified directory. This is *LEX so on the Welcome disc, these two commands have the same effects:

```
*LEX
```

```
*INFO $.LIBRARY.*
```

Making new directories

*CDIR

Before any files can be saved in a directory, it must be created. To do this, the command *CDIR is used. It is followed by the pathname of the new directory, eg:

```
*CDIR $.GAMES.ADVENTURES
```

will create a directory called 'ADVENTURES' in the directory '$.GAMES'. If '$.GAMES' was the CSD, the command could be shortened to:

```
*CDIR ADVENTURES
```

You can experiment with *CDIR on the WELCOME disc. Get into the WELCOME directory using:

```
*DIR $.WELCOME
```

If you catalogue the directory, you will see that all of its files are 'normal' ones; none of them have the letter 'D' next to their names. We will now create a directory called 'MyDir'. Since the CSD is '$.WELCOME', the new directory's full pathname will be '$.WELCOME.MyDir'. Type the command:

```
*CDIR MyDir
```

The disc will be accessed, and after a moment the prompt will re-appear. List the directory again using *CAT (wherever *CAT is mentioned you may, of course, use the abbreviation *.). This time an extra entry will be listed. It will have the form:

```
MyDir     DLR(56)
```

The sequence number will probably be different but the name and attributes should be the same. Once a directory has been created, it may be selected. Type the command:

```
*DIR MyDir
```

If you then catalogue it, the three 'general' lines will be printed but no filenames. This is because a directory is empty until a file is saved in it. You can now start to save and load files in 'MyDir' without fear of accidentally overwriting one of the important files in 'WELCOME' or 'LIBRARY'.

*TITLE

A directory may be given a long, meaningful name describing the sort of files it contains. If you followed the example of creating 'MyDir' above, you might have noticed that the directory's title is also 'MyDir'. When a directory is created its title is set to the same string as its filename. This may be changed to any string of up to 19 characters using the *TITLE command. For example, to change 'MyDir's title to 'Work directory', use the command

```
*TITLE Work directory
```

Other examples are:

```
*TITLE Chapter One
*TITLE My.Utils.Disc
*TITLE
*TITLE A very long title indeed
```

The third example sets the title to nothing and the fourth one will in fact make the title 'A very long title i' as this is 19 characters long. By naming the root directory you can effectively give a name to the whole disc:

```
*DIR
*TITLE Using BBC BASIC
```

File access permissions

*CAT, *INFO and their associated commands print letters describing files' access permissions. These letters appear between the filename and the sequence number. There may be up to three letters for each file. The letters are also called 'flags' because their presence flags the ability to perform certain operations on the file. Possible letters are:

D - Directory The presence of this letter means that the file is a directory. It is set when the directory is created and can't be changed.

E - Execute If this flag is set, the (non-directory) file may not be loaded or read in any way. The only commands that may operate on an execute-only file are *RUN, */, *DELETE, *REMOVE, *DESTROY and *ACCESS. The last may only be used to change the 'L' flag (see below): once the 'E' flag has been set it may not be removed. The *INFO of an execute-only file prints only its name, permissions and sequence number.

L - Locked Locked files cannot be deleted using *DELETE, *DESTROY or *REMOVE, neither may *RENAME or *SAVE (or SAVE) be executed on a locked file. It is also illegal to *SPOOL to a locked file. An attempt to do so will result in the message 'Locked' being given. Directories are locked when they are created, normal files are not. The command *ACCESS may be used to lock and unlock a file.

R - Read If a file has this access character it may be read. This allows operations such as *LOAD, *COPY, *EXEC and OPENIN (see chapter 4) to be performed on the file. The 'R' access can only be removed from non-directory files; a directory is always readable (by OPENIN for example). The command *ACCESS may be used to alter the state of the 'R' flag.

W - Write This flag determines whether a file may be opened for output. This affects the command *SPOOL, the utility *BUILD and

BASIC's OPENOUT and OPENUP. If the 'W' flag isn't set, the file may not be opened for output. See chapter 4 on using the ADFS from BASIC for details of opening files. Directories may not have a 'W' flag. The command *ACCESS may be used to alter the state of the 'W' flag.

The default state of the flags for directories is 'DLR' and for non-directories is 'WR'.

<u>*ACCESS</u>

Most of the flags described above may be changed using the command *ACCESS. It has two strings following it: a wildcard pathname and the new access flag. All files that match the pathname will have their flags changed as specified. Examples of the command are:

*ACCESS game RL	- Lock 'game' and enable it to be read
*ACCESS * E	- Make all of the non-directory files in CSD execute-only
*ACCESS chl	- Remove all flags except 'D' and 'E' from 'chl'
ACCESS lib. WR	- Make all files in 'lib' readable and writable

Typical uses of *ACCESS are:

- Setting the 'L' flag to prevent accidental deletion.
- Removing the 'L' flag to allow deletion of a directory.
- Setting the 'E' flag to 'protect' a machine-code program from being copied.
- Removing the 'W' flag to prevent writing to the file.

Non-directory commands

We now move on to more general-purpose commands, ie those which act on files that aren't necessarily directories. Indeed, some of the commands described below are not legal on directories.

<u>Saving, loading and executing files</u>

Files may be regarded as copies of the computer's memory that lie on the disc. They may contain BASIC programs, text, machine code programs, images of the screen memory or simply data. Languages such as BASIC provide built-in commands to save and load programs, but when other sections of the computer's memory have to be saved or loaded the filing system * commands are used.

<u>*SAVE</u>

The command to save a section of memory is called *SAVE and has several forms. The two simplest versions just take the start address and end address (or length) of the area to be saved, and the name of the file into which the memory must be saved. Examples are:

```
*SAVE RAM 0 8000
*SAVE temp 1228 + 321
```

The first command saves all of the computer's memory between addresses &0000 and &7FFF. Addresses given to filing system commands are always taken to be in hexadecimal. The first number is the address of the first byte to be saved; the second number is the address of the byte after the last one to be saved. It is a common practice in Acorn products to state upper limits in 'byte after' form (eg HIMEM in BASIC). This tends to make lengthy calculations easier.

The second command saves the memory between &1228 and &1548 inclusive in the file called 'temp'. This form of the command (where the start address is followed by '+') has the length of the memory segment instead of the end address as the second number.

In BASIC, the OSCLI statement may be used to incorporate BASIC variables into the *SAVE command. For example, suppose some machine code is assembled at address 'code'. To save the code automatically after assembly, the program might contain the lines:

```
2200 REM Save the object file
2210 OSCLI "SAVE OBJ "+STR$~code+" "+STR$~P%
```

This saves you having to go through a sequence like:

```
>PRINT ~code,~P%
    1254      1745
>*SAVE OBJ 1254 1745
```

to save the object file.

Having issued a *SAVE command, it is informative to look at the file's various 'attributes'. For example, type in these two commands from BASIC:

```
>*SAVE afile 4321+1234
>*INFO afile
AFILE      WR (69)  00004321  00004321  00001234  0015E4

Name       Access   Load      Execute   Length    Disc address
```

The entries labelled 'Load', 'Execute' and 'Length' are the ones of interest. 'Load' is the address in memory at which the file will be loaded using the *LOAD command. Notice that this is the same as the start address of the block of saved memory, so the file will be 'put back' where it was taken from. The next figure is the execution address. This is the address that will be called if the file is executed using *RUN or one of its equivalents. By default it is the same as the load address, but may be set differently if required.

The next number (00001234) is the length of the file. This is the same as the number given in the *SAVE command. If the first form of the command had been used (where the end address is specified instead of the length), the length would be calculated automatically. The last figure is the disc address, and doesn't concern us here.

Sometimes, we want a file's execution address to be different from the start address. This might occur, for example, where a machine code program starts with data rather than code. The execution address should be set to the 'entry point' address after the data. As an example in BASIC, suppose a program was assembled at the address held in 'code' and the start of the program proper was at the label 'entry'. To save the program, the required command is:

```
OSCLI "SAVE OBJ "+STR$~code+" "+STR$~P%+" "+STR$~entry
```

The difference between this and the previous OSCLI example is the addition of the extra STR$ part. The third address is the execution address: where the program must be executed after a *RUN. The value overrides the default value, which is the same as the load address.

The final embellishment to the *SAVE command is the ability to specify the reload address. That is, if you want the load address (as printed by *INFO) to be different from the start address of the file it may be specified after the execution address on a *SAVE command. Examples of *SAVEs with all possible values present are:

```
*SAVE image 3000+1000 8023 8000
*SAVE $.lib.type  2434 2523 C1D C00
```

The first line saves the region of memory between addresses &3000 and &3FFF under the name 'image' and sets the load address to &8000 and the execution address to &8023. The second example saves the program between addresses &2434 and &2522 under the name '$.lib.type' with a load address of &C00 and an execution address of &C1D. Note that the reload address may only be given if the execution address is present.

*LOAD

The opposite action of saving a file is, of course, loading it. The *LOAD command has fewer variations than *SAVE. Examples are:

```
*LOAD image
*LOAD type 2300
```

The first example loads the file called 'image'. The address at which the file will be loaded is its load address as printed by *INFO. The second example loads the file 'type' at address &2300. The file's own load address is overridden.

*RUN, */ and *<file>

Executing a machine code program can be done in three ways. The forms are:

```
*RUN <file>

*/<file>

*<file>
```

where <file> is a pathname which may contain wildcards (which are interpreted in 'first found' mode). *RUN and */ are exactly equivalent. They look in the CSD and CSL for a file of the name specified in the command and load and execute it (using its load and execution addresses respectively) if found. If the file doesn't exist, a 'Bad command' error is given. If the name after the * in the third form does not correspond to a built-in command, the ADFS treats it as if prefixed by *RUN, ie tries to execute the file of the same name.

If the filename to be executed begins with a special character, '$', '&' or ':', the CSD and CSL are not searched for the file. Instead the actual file specified is sought. Suppose that the CSD is '$.UTILS' and CSL is '$.LIB'. The actions of various commands are:

*RUN fred

Looks for $.UTILS.fred, then $.LIB.fred

*RUN file.type

Looks for $.UTILS.file.type, then $.LIB.file.type

*RUN $.cset

Looks for $.cset

*RUN @.help

Looks for $.UTILS.help, then $.LIB.help

*RUN ^.print

Looks for $.print,then $.print again

*RUN :1.stars

Looks for :1.$.stars

The Welcome disc contains several useful programs in the directory '$.LIBRARY' that may be executed using *RUN, or */ or simply *command. These are discussed in detail in chapter 3.

If a file has an execution address of -1 (&FFFFFFFF), it will not be loaded and executed in response to *RUN. Instead, it will be *EXECed. This process is described in detail later, but basically it involves reading the contents of the file as if they had been typed at the keyboard, enabling commands to be put into file for later execution.

Deleting, renaming and copying files

If a file is no longer required, it should be deleted. This reduces the number of 'Disc full' and 'Directory full' errors given. Sometimes, a file is still required but under a different name (to remove a clash of command names, for example). Yet another possibility is the requirement for a copy of a file or files in another directory. This section describes the commands that perform these operations.

*DELETE and *REMOVE

The command *DELETE removes a single named file from a directory. The filename should not contain wildcards. Examples are:

```
*DELETE temp

*DELETE ^.^.filel
```

If the file that you are trying to delete does not already exist an error will be generated ('Not found'). Sometimes, especially from within programs, it is desirable to ensure that a file does not exist, but without producing an error if the file has been deleted already. The command *REMOVE acts as *DELETE but does not complain if the file does not exist. Again, the filename or pathname after the command should not contain wildcards. Examples are:

```
*REMOVE dataFile

*REMOVE $.spoolOut
```

You may not delete a file that is 'locked'. To lock a file, the command

```
*ACCESS file L
```

should be issued. A locked file may not be *REMOVEd, *DELETEd, *DESTROYed or *RENAMEd. You can tell if a file is locked by the letter L appearing next to its sequence number when it is listed by *CAT or *INFO. To unlock a file (and enable the above operations on it), the command:

```
*ACCESS file
```

may be used.

*DESTROY

It is possible to delete a group of files with one command: *DESTROY. This takes a filename that allows wildcards. Any file found that matches the pathname will be deleted. Before carrying out this potentially catastrophic operation, the ADFS prints the *INFO data for the files that will be affected and prompts you with:

```
Destroy ? _
```

In order to carry out the command you must type YES <RETURN>. Any other sequence of characters (except lower case yes) will cause the command to be aborted. To delete all chapters of a book, for example, this command might be used:

```
*DESTROY Chapter*
```

Note that wildcards only act in 'multiple' mode on normal files, not directories. Thus only wildcards in the last part of a pathname will match more than one file. So:

```
*INFO *.*
```

will not print information about all files in all directories in the current directory, but information about all files in the first directory found in CSD. Similarly:

```
*DESTROY #.ch*
```

will (potentially) delete all files beginning with 'ch' in the first one-character directory in CSD.

When trying to delete a directory, you must first ensure that several conditions are met. First, the directory to be deleted must be empty, ie it may not contain any files, including other directories. An attempt to delete a directory containing files will yield a 'Dir not empty' error. A directory must also be unlocked before it can be deleted (directories created by *CDIR are automatically locked), and must not be the CSD or CSL. This set of rules prevents directories being deleted in such a way as to leave the structure of the disc in a corrupt state.

*RENAME

To give a file a new name, perhaps in a different directory, the *RENAME command is used. For example:

```
*RENAME edit $.utils.edit
```

would move the file in the CSD called 'edit' into the directory '$.utils'. The final name does not have to be the same:

```
*RENAME invaders games.galaxians
```

This changes '@.invaders' to '@.games.galaxians'. You don't have to change the directory of the file:

```
*RENAME CH1 CH2
```

will keep the file in the same directory but make its name 'CH2' instead of 'CH1'. In all cases, the first file of a *RENAME command 'disappears'. Neither of the files may contain wildcards.

*COPY

The *COPY command makes a copy of a file (or several files) in a different directory and retains the original(s). The command is followed by two names: the pathname (optionally containing wildcards) of the file(s) to be copied, then the non-wildcard name of the directory into which the files are to be copied. A typical *COPY command is:

```
*COPY $.* $.safe
```

This will copy all non-directory files in the root directory into the directory called '$.safe'. There will thus be two versions of all of these files on the disc. Another example is:

```
*COPY LIB.# @
```

which copies all files with one-character names from '@.LIB' to the CSD. *COPY can also transfer files to another disc, if there is more than one fitted. For example, to copy all of the files in root on drive 0 to root on drive 1:

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

You can't use *COPY to copy files on to another disc on a single-drive system as it does not give you a chance to swap discs. The utility 'DIRCOPY' may be used for this purpose. If you want to copy the whole of a disc on to another, maintaining exactly the same directory structure, use the *BACKUP utility mentioned in chapter 3. Warning: The *COPY command uses the workspace from OSHWM to HIMEM when copying files. Since this area is also used by languages (such as BASIC) and work processors (such as VIEW), it is important that any program or text file be saved before *COPY is used. This is not necessary if a second processor is attached.

The *SPOOL and *EXEC commands

These commands enable you to send all screen output to a file, and to treat a file's contents as keyboard input. Strictly speaking these are operating system commands (as opposed to ADFS commands) so you may have come across them already when using tape.

*SPOOL

After the command

```
*SPOOL file
```

has been executed (where 'file' is any non-wildcard pathname), all characters that are printed on to the screen (including invisible control characters) will also be sent to the named file. This will continue until the command

```
*SPOOL
```

without a filename is executed. One application of *SPOOL is to keep a permanent record of what has been shown on the screen. The *SPOOL file may later be edited using a word processor such as VIEW, or displayed directly on to the screen using one of the utilities described in chapter 3.

As an example, suppose we wish to include a multiplication table from 1 to 12 in a document being prepared using VIEW. The first step is to write a simple BASIC program that prints the table onto the screen in the desired format. Since the text printed will later be read into VIEW in exactly the same way as it was printed, the program should print the table a line at a time starting from the top. One way of doing this is shown in the following program:

```
1000 REM  Program to print a 1..12 times table
1010 width=5
1020 PRINT TAB(30)"Times table"'
1030 @%=width
1040 PRINT "Times"TAB(2*width);
1050 FOR i%=1 TO 12
1060   PRINT i%;
1070 NEXT
1080 PRINT'
1090 FOR j%=1 TO 12
1100   PRINT j%":"TAB(2*width);
1110   FOR i%=1 TO 12
1120     PRINT i%*j%;
1130   NEXT
1140   PRINT
1150 NEXT
1160 PRINT
```

If you run this program in MODE 3 or MODE 0, you will see that it produces a fairly simple-looking table. The next step is to get the table into a form that may be read by VIEW. This is accomplished by adding just two lines to the program. Add the lines:

```
10 *SPOOL timesText
1170 *SPOOL
```

The first line starts spooling, sometimes called 'opening the spool file' so that the text printed on the screen will also go into 'timesText'. The second line stops spooling, or 'closes the spool file' so that output goes only to the screen and not the file. Before running the program again, save it with a command such as:

```
SAVE "TimesTable"
```

Run the program. This time the disc will be activated and the screen will go blank from time to time (because the disc system is being used in one of the 'large' screen modes). When the program has finished, type the command:

```
*INFO timesText
```

This will reveal the presence of a new file which is &3D9 bytes long. This file consists of exactly the same characters that were printed on the screen by the program. If you have been using the Welcome disc, you can see the contents of the file by typing:

```
*LIB $.LIBRARY
*TYPE timesText
```

This uses one of the utility programs in the directory '$.LIBRARY', described in chapter 3.

If you have a VIEW cartridge, the following sequence of commands will let you read 'timesText' into the word processor:

```
*WORD
```

```
=>READ timesText
```

Pressing <ESCAPE> as usual to see the text will reveal that it was read in properly. Note that you cannot LOAD spooled files in VIEW as it objects to the presence of line-feed characters that are generated whenever BASIC prints a new line.

Another use of *SPOOL, merging BASIC programs, is described in chapter 4 on using the ADFS with BASIC.

*EXEC

In some ways this may be regarded as the opposite to *SPOOL. After the command

```
*EXEC file
```

has been executed, the Electron will stop using the keyboard as its source of input. Instead it will obtain characters from the file named after the command. This continues until the file is exhausted, whereupon input reverts back to the keyboard.

A typical application of *EXEC is to execute a list of commands that would otherwise be tiresome to type over and over again. An example is programming the Electron's function keys. Many users who use these keys like to set them to certain fixed strings at the start of a session with the computer. If all ten keys are used, this could be a time-consuming task. The short-cut is to put the commands to define the keys into a text file, and just *EXEC this whenever the keys have to be defined (eg at power-up or after a <CTRL> <BREAK>).

The first step is to create the file with the commands in. This can be done using any word processor that marks the end of lines using a carriage-return character (VIEW does this), or using the *BUILD utility which is described in chapter 3. For now we will use BASIC to create a command file. Type in:

```
NEW
10 *SPOOL myKeys
20 PRINT "*KEY0BASIC0"
30 PRINT "*KEY1OLD0LIST0"
40 PRINT "*KEY2AUTO1000,100"
50 *SPOOL
RUN
```

When the program is run it will create a textfile called 'myKeys' which contains the commands printed by the PRINT statements. To subsequently execute the commands as if they had been typed in at the keyboard, simply type:

```
*EXEC myKeys
```

The commands will appear on the screen exactly as if you had typed them, albeit much faster. To check that the commands were obeyed, try pressing FUNC 0, FUNC 1 and FUNC 2.

The commands that appear in a *EXEC file may be of any type, for example BASIC commands, as above, VIEW commands, or even other * commands. The language you are using cannot distinguish between what is typed and what is taken from a *EXEC file, so the general rule is 'if it can be done from the keyboard, it can be done from an exec file'.

Closing all open files

Sometimes the error 'Already open' is encountered. This occurs when an attempt is made to delete, overwrite or open for output a file which is already open. From BASIC it is a simple matter to delete all open files by typing CLOSE#0.

*CLOSE and *BYE

In languages other than BASIC, eg VIEW, there is no built-in CLOSE command, so if an 'Already open' error is preventing you from saving some text you would have to save the text under a different name, call BASIC, close all files, re-enter VIEW, reload the file and finally save it under the correct name.

To avoid this long sequence of commands, the ADFS command *CLOSE is provided. This closes all files, as CLOSE#0 in BASIC, but has the advantage that it may be typed anywhere that * commands are allowed, eg from VIEW.

*BYE has the same effect as *CLOSE but is quicker to type. The main difference is that when the ADFS is used with Winchester discs on the BBC Microcomputer, *BYE automatically moves the disc 'head' to a safe area on the disc. Because Winchesters are not used with the Electron, the commands are effectively the same.

Commands affecting the whole disc

We move now on to more general commands which affect the whole of the filing system, or the current drive, rather than just files or directories.

Obtaining help about the ADFS

The operating system provides a command *HELP. On a basic Electron with no expansion this will just print the version number of the operating system. However, as components are added to the system, more information is printed out. For example with the Plus 1 added, *HELP tells you that the Electron has printer and analogue to digital conversion capabilities.

*HELP ADFS

Typing *HELP with the ADFS installed tells you that the ADFS is available, with text of the form:

```
Advanced DFS 1.10
  ADFS
```

This says that the ADFS has a version number of 1.10 and that a 'subheading' ADFS may be used. Thus typing:

```
*HELP ADFS        or
*H.. for short
```

will give a summary of the ADFS commands' syntax. Note that only the purely ADFS commands are listed, not the operating system ones such as *CAT and *SPOOL.

Auto-boot files

In the discussion of the output produced by *CAT earlier we mentioned the 'Option' line, and the file '!BOOT'. This section explains what the 'option' facility does.

If the ADFS is entered in *ADFS mode (eg by pressing A <BREAK>), the disc drive will start up and the information about the previous CSD is loaded. (If there is no previous CSD, the '$' will be used). Also at this stage, the CSL is set to the first directory whose name begins with '$.LIB'. When this process has finished, control returns to the current language as usual.

If, however, the user holds down <SHIFT> before pressing <BREAK>, and keeps it pressed for a couple of seconds after releasing <BREAK>, the ADFS will attempt what is known as an 'auto-boot'. First the filing system will find out what the disc's boot option is. It can have one of four possible values:

00	Off, do nothing
01	*LOAD the file called !BOOT or 'Not found' error
02	*RUN the file called !BOOT or 'Bad command' error
03	*EXEC the file called !BOOT or 'File not found' error

When a disc is formatted (using the *EFORM utility) its boot option is set to 'Off'. However, by setting it to one of the other three values, the ADFS will look for a file called '$.!BOOT' and perform one of the actions given above on it. If no such file is on the disc, the error message indicated above is printed and you have to press <BREAK> or A <BREAK> without <SHIFT> to enter the ADFS normally.

*OPT4

The command to set the boot option is *OPT4 followed by the option number (separated by , or a space). Examples are:

```
*OPT4,1
*OPT4 3
*OPT4
```

The last example is the same as *OPT4,0. An obvious use of the '!BOOT' file is to make it a command file that is *EXECed. The Welcome disc, for example has its boot option set to 03 (EXEC) and the '!BOOT' file contains the simple text:

```
CHAIN "!B"
```

where '!B' is the name of another (BASIC) file in the root directory. The *OPT4 command always acts on the current drive, so to set the boot option on drive 1, two commands are needed:

```
*DIR :1
*OPT4,2
```

It is also possible to make the '!BOOT' file be accessed when <BREAK> is pressed without <SHIFT>. To do this, the start-up options must be set: see the disc documentation on the 'SETPARAMS' utility for details.

Enabling ADFS messages

A facility exists whereby the ADFS will print *INFO information whenever files are accessed.

*OPT1

This command controls whether ADFS messages are printed or not. By default, no messages are given. The command:-
*OPT1,1
switches messages on and

*OPT1,Ø   or
*OPT1
switches them off again. The messages are useful for checking that the correct files are being accessed during the execution of a program. The command *OPTØ has the same effect as *OPT1,Ø.

Obtaining disc storage information

Two commands are available that give information about the currently selected disc.

*FREE

This tells you how much space on the disc is in use, and how much is left. The output of the command typically looks like this:

```
ØØØ15C Sectors =      89,Ø88 Bytes Free
ØØØ3A4 Sectors =     238,592 Bytes Used
```

The first figure is in hexadecimal, the second in decimal. The total number of sectors and bytes is constant for a given size disc. For a single-sided 8Ø track disc it is: &5ØØ (128Ø) sectors and 327,68Ø bytes. A sector is simply a small area of the disc that the ADFS finds it convenient to deal with. There are 256 bytes in each sector. Every time a directory is created, five sectors of the disc are used. A normal file occupies ((len+255) DIV 256) sectors, where 'len' is the length of the file, as given by *INFO.

*MAP

The other command that obtains information about the whole disc is *MAP. Typing this will yield a table similar to this:

```
Address   :   Length
ØØØØ13    :   ØØØØØ4
ØØØ48B    :   ØØØØ75
```

The list may be shorter or longer than the example shown. The entries are the areas of free space on the disc and their lengths. In this example, there are two such areas. Both the address (which is the disc sector address of the first free sector) and the length (which is the number of sectors in the free area) are given in hexadecimal.

The ADFS maintains a list of free space (which is created when files are deleted) so that when a new file is created it can re-use the free space. There may be up to 80 entries in the free space list (or 'map'). If the list starts to become full, the disc is becoming fragmented; that is, there are many fairly small areas of free space on the disc. Sometimes, when trying to save a long file you will get a 'Compaction required' error. This means that there is enough total room on the disc to hold the file, but the space is scattered all over the disc. To solve the problem, the disc must be compacted. This operation goes through the disc, shifting used and free areas of the disc around so that the free space list has fewer entries, but of larger size.

*COMPACT

After much use, the organisation of files on a disc may become fragmented. For example, there may be gaps on the disc where files have been deleted. These gaps may together occupy a lot of disc space, but individually they may not be large enough to hold a particular file. Compaction resolves the problem.

There is a command, *COMPACT, to compact the disc; however, the fastest and safest way is to use the utility FASTCOMPAC on the utilities disc. It can be called up from the menu (*UTILS), or directly by entering *FASTCOMPAC. Note that any programs or data in the Electron's memory will be overwritten, since FASTCOMPAC uses all available memory to speed up the process. Therefore you should save any valuable program or information (perhaps on another disc) before using FASTCOMPAC.

If you want to compact without overwriting your program, *COMPACT can be used directly. By default, *COMPACT will use the screen memory for its working area. Enter:

```
*COMPACT
```

Compaction is more efficient in MODE 0, 1, 2 or 3, because more screen memory is available.

You can also specify exactly which areas of memory COMPACT should use for its workspace. The command is followed by the start address of the area to be used and the length of the area. Both numbers are in pages rather than bytes, and should be given in hexadecimal. You should only use this form of the command if you understand the memory map of the Electron.

For example:

```
*COMPACT 20 10
```

will use the area between addresses &2000 inclusive and (&2000 +&1000) = &3000 exclusive. To use the memory between PAGE and the screen (HIMEM in BASIC), the following OSCLI statement may be used:

```
OSCLI "COMPACT "+STR$~(PAGE/&100)+" "+STR$~((HIMEM-PAGE)/&100)
```

Alternatively, both the program and screen memory may be used:

```
OSCLI "COMPACT "+STR$~(PAGE/&100)+" "+STR$~((&8000-PAGE)/&100)
```

Note that more than one *COMPACT may be required before enough free space is collected to avoid 'Compaction required' errors.

Changing discs

*MOUNT and *DISMOUNT

Having finished using a particular disc, you may want to access information on another one. To prevent information being lost during the swap, a couple of commands have to be issued before removing the old disc and after inserting the new one. Assuming that a single drive is in use, here is the sequence required to change discs:

```
*DISMOUNT
(Take out old disc and insert new one)
*MOUNT
```

The *DISMOUNT command (which may optionally be followed by a drive number) closes all sequential files and makes CSD and CSL 'unset'. Sequential files are those which are dealt with a byte at a time, as used by *SPOOL and *EXEC, and BASIC's OPENIN, OPENUP and OPENOUT functions. They are discussed with respect to BASIC in chapter 4. By closing all sequential files, *DISMOUNT ensures that the information on the disc is up to date. After a disc has been *DISMOUNTed, there is no directory or library selected, so most commands will yield a 'No directory' error.

If there are no sequential files open, *DISMOUNT may be omitted.

The *MOUNT command, which may also be followed by a drive number, causes the system to acknowledge the existence of the new disc. It loads the disc's free space map (so that it knows where to put new files) and sets CSD to '$'. CSL is unchanged (so will be 'unset' if *MOUNT follows a *DISMOUNT) and must be assigned explicitly by a *LIB command. *MOUNT is very similar in effect to *DIR :0, the difference being that the latter leaves the CSL as it was and the former 'unsets' it.

The *DISMOUNT and *MOUNT commands may also be used to switch between drives in a multi-drive system. To change from drive 0 to drive 1, for example:

```
*DISMOUNT 0
*MOUNT 1
```

Alternatively the single command *DIR :1 could be used.

3. THE ADFS UTILITY PROGRAMS

The Welcome disc contains several programs for use with the ADFS. These perform a wide range of tasks, including formatting discs, making backups and examining the contents of files. Among them are:

*EFORM	Formats a disc so that it may be used with the ADFS
*VERIFY	Checks that a disc's contents are not corrupt
*BACKUP	Copies the entire contents of one disc to another
DIRCOPY	Copies selected files from one disc to another
*LIST	Displays a text file with line numbers
*TYPE	Displays a text file without line numbers
*BUILD	Creates a text file from the keyboard
*DUMP	Displays a file in hex and ASCII
SETPARAMS	Sets up various ADFS options
UTILS	Produces the menu of utility programs

Commands marked with a '*' are written in machine code, the others are BASIC programs. Before using a utility, the current library should be set to '$.LIBRARY'. This may be done manually using the command

```
*LIB $.LIBRARY
```

or automatically by starting the ADFS with CTRL A BREAK.

Running utilities individually

All the utilities on the Welcome disc can be run using * commands. Where the utility is a machine code program, it is executed immediately and the command name may be followed by parameters, for example:

```
*TYPE myFile
*EFORM M Ø
```

When the utility is in BASIC, the * command calls a *EXEC file which enters BASIC and chains the program. The program itself is in the directory '$.UTIL'. For example, the command *SETPARAMS will *EXEC a file called '$.LIBRARY.SETPARAMS' which contains the following:

```
*BASIC
CHAIN "$.LIBRARY.BASIC.SETPARAMS"
```

Utilities written in BASIC do not have parameters following the * command; they are all 'prompt driven'.

The utilities menu

Typing the command

```
*UTILS
```

will cause a list of the current utility programs to be displayed. There may be up to 47 utilities on a disc, and their names are displayed in up to three columns of 16 lines. Each name has a digit, letter or other character next to it. To run a given program, type the character corresponding to the name. For example, if the first few entries on the list were

```
0 * BACKUP
1 * BUILD
2   DIRCOPY
3 * DUMP
4 * EFORM
5 * LIST
```

then typing 3 would cause the *DUMP utility to be executed. The * indicates a machine code program. If the program is in BASIC, it will be chained, and therefore you will be left in BASIC when it terminates. If the program is a machine code one, you will be given a chance to type the parameters. The command name is given as a prompt. For example, if you type 1 for BUILD, a prompt

```
*BUILD _
```

would appear at the bottom of the screen. Type the parameters, followed by <RETURN>. The command will then be called using the parameters supplied.

Machine code utilities always return to the calling program, so after running one of these you will see the menu again.

To escape from the menu, press ESCAPE.

Obtaining documentation on utilities

All of the Welcome disc utilities have instructions on the disc. To read these, call the menu program using *UTILS. Press the space bar. This will call up the help menu. This is very similar to the utils menu, but because utilities added by users might not have any documentation the list may be shorter. To see the information for a given command, press the key corresponding to its name.

This will cause the appropriate file to be printed in paged mode. Press SHIFT to continue when output stops. At the end of the instructions, you will be asked to Press SPACE to continue.

To recall the utils menu, press the space bar again.

Adding your own utilities

More experienced users might want to add their own utilities to the disc. If these are added correctly, they will automatically be included in the menus printed by *UTILS. The exact method depends on whether the program is in BASIC or machine code.

Adding BASIC utilities

If you have a BASIC program 'SCRNDUMP' which you would like to include in the library, follow the instructions:

1. Save the BASIC program as '$.LIBRARY.BASIC.SCRNDUMP'.
2. *BUILD $.LIBRARY.SCRNDUMP. This should contain the two lines:

```
*BASIC
CHAIN "$.LIBRARY.BASIC.SCRNDUMP"
<ESCAPE>
```

It is very important that there are no characters after the <RETURN> of the second line. If there are, the *EXEC file will not be closed properly before the BASIC program is called, and the extra characters will be read by any INPUT statements in the program.

3. You should document the utility. If you have VIEW, the file should be prepared using MODE 6. Because VIEW uses control codes to right justify the text, these have to be removed, as follows:

- Edit the file using VIEW.
- Save it as 't' (just in case).
- Type *SPOOL temp.
- Type SCREEN.
- Type *SPOOL.
- Type NEW followed by READ temp.
- Remove the last line of the text which will say *SPOOL.
- Save the file under '$.HELP.SCRNDUMP'.
- Delete 't' and 'temp'.

Alternatively, short files may be prepared using, for example, *BUILD

```
$.HELP.SCRNDUMP.
<ESCAPE>
```

Adding machine code utilities

The technique for adding machine code programs to the library is very similar to that described above. To add a utility called 'MCDUMP', for example:

1. Save the object code with a command of the form:

```
*SAVE $.LIBRARY.MCDUMP <start> <end> <exec> <reload>
```

2. Prepare the documentation file as described under 3 above.

4. USING THE ADFS FROM BASIC

Electron BASIC provides commands, statements and functions which access the filing system directly. Because of the uniform specification of Acorn filing systems, the same program can normally work with ADFS, Econet and tape. The only differences in operation are those forced by the restrictions of the medium used, eg cassette files are 'serial' and cannot support random access, and by the fact that PAGE is higher. This means that many of you programs that work from tape will not work when the ADFS is fitted.

Page under the ADFS

There is one major difference between using BASIC under the tape filing system and under the ADFS: the default value of the pseudo-variable PAGE is much higher under the ADFS. The reason is that discs need much more 'workspace' to operate than tape; for example, the ADFS always keeps the current directory in RAM to avoid having to access the disc for simple operations such as *CAT and *INFO.

If you print the value of PAGE using a statement such as

```
PRINT ~PAGE
```

you will see that its value has increased from &E00 to &1D00 (or &1F00 if you are using Econet too).

Since PAGE is where BASIC programs start, there are 3840 fewer bytes available to BASIC under the ADFS than the cassette filing system. This disadvantage is offset to a degree by the speed of discs: large programs that can no longer be loaded under the ADFS can be split into several smaller sections which are CHAINed in as necessary. A method of 'copying down' programs so that they reside at the old value of PAGE is given later in this chapter.

Methods of accessing the ADFS

Interaction with the filing system from BASIC can take several forms: commands such as LOAD and SAVE deal with the current BASIC program directly, 'star' commands like *LOAD, *SPOOL and *CDIR are passed to the ADFS, and finally BASIC statements such as OPENIN and CLOSE deal with filing system sequential files. In this section only the first and last uses of the filing system are relevant. Some of the more common ADFS commands such as *LOAD are mentioned briefly, but for detailed information you are referred to chapter 2.

Whole-file operations

This section covers the operations that act on whole files, whether they are BASIC programs or machine code files.

The SAVE command

The BASIC command SAVE is similar to *SAVE but acts on the current program instead of a general block of memory. An example is:

```
>SAVE "stats"
```

which will save the current program under the name 'stats'. The start address is set to PAGE; the end address (ie the location after the last byte to be saved) is set to TOP. The *INFO of a BASIC program called 'progl' might look like this:

```
progl      WR (34)    FFFF1D00    FFFF8023    00000777    000023
```

This implies that 'progl' was saved with PAGE set to &1D00. The execution address (&FFFF8023) is meaningless for BASIC programs, and the length &777 is the same as TOP-PAGE in hex when the program is in memory.

The ADFS does not distinguish between types of files, apart from stating that directories have a certain format. This implies that BASIC files can be treated as any other: they may be opened for reading or writing (see 'Sequential files' later), or *LOADed to an arbitrary address. This latter property is very useful, as illustrated under the section: 'Merging BASIC programs'.

The LOAD command

LOAD in BASIC is similar to *LOAD, but makes the file load at PAGE, regardless of its own load address. An example is:

```
>LOAD "stats"
```

which will load the program called 'stats' at PAGE. After the program has loaded, BASIC checks that it is valid (as it does when END is executed), and if it finds an error, 'Bad program' is reported. Note that it is often possible to RUN a 'bad' program, even if it can't be LISTed.

BASIC does not check the length of a program before loading it (as this cannot be done on the cassette filing system) so a bad program may be caused by its being too large to fit in memory, and thus corrupted upon loading. An example is when loading a program that was written using a 'small' screen mode, say MODE 6, when a 'greedy' mode, say MODE 3, was in use.

LOAD is like NEW in that it removes all the current program's variables, apart from the system integer variables, A%-Z% and @%. If the LOAD was unsuccessful (ie the program could not be found), the old program and its variables remain intact.

The CHAIN statement

The CHAIN statement (which may be used from within a program) acts exactly as LOAD followed by RUN. If the program is 'bad', however, BASIC will not attempt to run it. Both LOAD and CHAIN allow 'wildcards' in the filename, for example:

```
>CHAIN "BM*"
```

will load and run the first program in the current directory that begins with the letters 'BM'. As entries are always stored alphabetically in the directory, it is easy to predict which file will be loaded.

SAVE, LOAD and CHAIN all allow general string expressions as their arguments. This is convenient for ensuring that a program is always stored using the same name. If the program contains a line such as:

```
10000 DEF FNNM="sortl"
```

the command:

```
>SA.FNNM
```

saves the file in a conveniently small number of keystrokes.

Downloading a BASIC program

Sometimes a program that worked OK using tape will give a 'No room' error message after being RUN under the ADFS. If the program does not use the disc at all after it has loaded, it is possible to copy the program down to the old PAGE &E00 and run it there. Because the program will now lie in disc workspace, any attempt to use the ADFS will cause the program to be corrupted.

For a program to download itself, the first few lines should be:

```
1000 REM Down-loader program header
1010 IF PAGE<=&E00 THEN 1060
1020 VDU 21
1030 *KEY 0 *TAPE|MFOR I%=0 TO TOP-PAGE STEP 4:
I%!&E00=I%!PAGE:NEXT|MPAGE=&E00|MOLD|MRUN|F|M
1040 *FX138,0,128
1050 END
1060 REM The rest of the program
```

Line 1010 checks to see that the program isn't at page &E00 already. If it is, a jump is made to the main program at line 1060.

Line 1020 turns off the VDU drivers so that the commands used to copy the program down don't appear on the screen.

Line 1030 programs key 0. When executed, the commands select the cassette filing system, moves the program down to page &E00, sets PAGE to &E00, issues an OLD to reset BASIC's pointers, then RUNs the program. The |F is a VDU 6 to re-enable the VDU drivers just before the program is run.

Line 1040 enters the internal code for function key 0 into the keyboard buffer. This will cause the string programmed in the previous line to be executed.

Line 1050 stops the program, re-entering BASIC's command mode so that the function key string may be executed.

The *TAPE command ensures that no ADFS commands are issued when the program is run. This would cause the ADFS to report an error such as

'Bad FS map'. Worse, the program itself might be corrupted by the ADFS attempting to access its workspace (which is where the program now resides).

After a program has copied itself down, pressing <BREAK> to re-enter the ADFS will fail. This is because the ADFS realises that its workspace has been corrupted and prevents itself from being selected. It does this by a process known as frugalising, which effectively makes the ADFS disappear from the machine. To re-establish the ADFS, the whole of the computer's memory has to be cleared out. To do this, type

```
*FX200,2
```

and press <CTRL> <BREAK>. The ADFS will select itself exactly as if the machine had just been turned on. Sometimes the error 'Bad FS map' will be given after exiting from a downloaded program. This may also be cured by the *FX200,2 technique.

Merging BASIC programs

As one of the strengths of Electron BASIC is the way it enables procedure libraries to be builtup, it is obviously desirable to merge programs together, that is joining a 'library' routine to a main program. There are two ways of doing this. The first method involves *LOADing the second file directly at the end of the first one. This is a pure 'append', as the second program is literally attached to the end of the first one. The second method is a 'merge', as the second program is entered as if typed at the keyboard with the first program already loaded.

Suppose it is required to append a file called 'shellSort' to the current program. The following sequence of commands will do it:

```
>PRINT ~TOP-2
     2393
>*LOAD shellSort 2393
>END
```

The PRINT obtains the address of the end of the last line of the current program. This is where the new file has to be placed. The *LOAD gets the procedure file and puts it at the end of the current program (ie at the address printed by the previous statement). The END ensures that BASIC updates its version of TOP to the end of the merged program. LIST will also do this.

It is possible to define a softkey string to perform the above actions:

```
>*KEY0 INPUT'"File name: "f$:OSCLI"LOAD "+f$+" "+STR$~(TOP-2):END|M
```

This also works when there is no program in the machine already, so can be used to load as well as append.

One small problem with the merge technique described above is that if the appended file has line numbers that are lower than the original one, you get programs looking like this:

```
2100 REPEAT
2110 UNTIL INKEY-99
800 DEF PROCshellSort(first,last)
810 REM....
```

If there are no GOTOs or other line-referencing statements in the program, the simple solution is RENUMBER it. If there are GOTOs, RENUMBER will probably get confused and the best solution is to make sure that your library files have very high line numbers (remembering that the upper limit is 32767).

An alternative technique, which is a true merge rather than the 'append' of the first method, is to use *SPOOL and *EXEC. The method is as follows: first load the library file, called 'quickSort' for example. Then type:

```
>*SPOOL qsText
>LIST
>*SPOOL
```

The first line opens a *SPOOL file; any characters sent to the screen will also be put into the text file called 'qsText'. The LIST acts as usual, the listing being sent to 'qsText' in addition to the screen. The *SPOOL closes 'qsText' and makes the output go only to the screen as usual. There is now a file called 'qsText' on the disc which is a text version of the quicksort program.

Next, load the 'main' program with which the sort routine must be merged, eg LOAD "dBase". To merge in the sort file, type:

```
>*EXEC qsText
```

The screen will now display the contents of qsText as they are read in. *EXEC takes the contents of the file specified and reads it in exactly as though it were typed at the keyboard. Thus BASIC thinks it is getting lines from the user, and inserts them into the BASIC program as usual. At the end of qsText, *EXEC will stop and input will revert to the keyboard.

The *EXECed file will appear to be double spaced because of the line feeds put in when the program was LISTed. There will also be a couple of error messages caused by the *SPOOL commands at the start and end of the file. These do not affect the operation of the merge command at all. If you type LIST after the merge operation you will see the new lines. Note that if the main program has line numbers the same as the 'library' routine, the latter will take precedence over the former.

Machine code files

When a BASIC program needs to call a machine code routine, it is desirable to have only the object file in memory, rather than the source program. The usual way of doing this is to assemble the file using the 'assemble at P% or O%' assembly option (see chapter on assembly language in the Electron User Guide), and save the object program with the appropriate execution and reload addresses. When it has to be called, the object file is loaded at the appropriate place.

Suppose there is a machine code routine which is about two pages (512 bytes) long. A convenient place to put it is at the top of RAM, below the screen memory. Assuming MODE 4 is in use, HIMEM (and hence the screen memory) will be at &5800. A convenient place to put the program would be at &5600. HIMEM should be moved down to accommodate the code. The body of the source program will look like this:

```
1000 DIM code &200 : object=&5600
1010 FOR pass=4 TO 6 STEP 2
1020 P%=object : O%=code
1030 [ OPT pass
1040  \The source program
1050  .entry
1055  \The rest of the source
1060 ]
1070 NEXT pass
1080 OSCLI"SAVE objProg "+STR$~code+" "+STR$~O%+" "+STR$~entry+" "+
STR$~object
```

This uses the long form of the *SAVE command:

```
*SAVE <name> <start addr> <end addr> <execution addr> <reload addr>
```

To use the code in another program, this sequence of instructions would be used:

```
300 HIMEM=&5600
310 *LOAD objProg 5600
320 CALL &5600,param1%
330 REM and so on
```

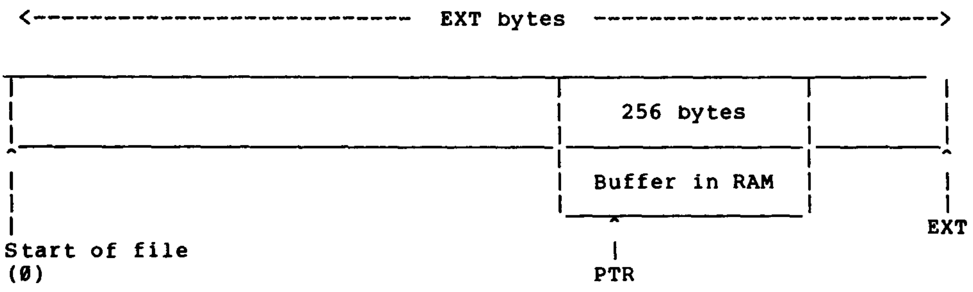
The load address doesn't have to be specified as it was set to &5600 by the assembly program, but it is safer to make it explicit. If the object had to be called only once, and with no parameters, the two lines:

```
300 HIMEM=&5600
310 *objProg
```

would suffice. When the program returns, HIMEM may be reset to &5800.

Sequential files

Data used in BASIC programs is obviously lost when the machine is turned off. In order to provide a permanent record of variables, the filing systems have 'sequential files'. A sequential file may be regarded as an array of bytes, similar to the arrays dimensioned by the special byte form of the DIM statement. The difference is that files lie on a medium such as disc or tape, and arrays are in the computer's main memory. Only a section of an open sequential file (the buffer) is loaded in at once. This provides rapid access to, usually, 256 bytes of the file. An open sequential file may be pictured as follows:



PTR is the sequential pointer which marks the 'current location' in the file. It is set to 0 on an opening file, so read and write operations start at the beginning of the file. PTR is discussed in detail later. EXT means 'extent' - the length of the file. There is a BASIC function to return this value for an open file. It is also possible to alter the length of the file using machine code.

BASIC supports sequential files with various statements and functions. Before a file may be used, it must be 'opened'. Once this is done, the user may read characters from the file, write characters to the file, or update existing parts of the file. There are three functions in BASIC used to open files:

OPENIN	- Open a file for input only. The file must exist already.
OPENUP	- Open a file for input and output. The file must exist already.
OPENOUT	- Open a file for output. The file may be created by OPENOUT.

All three take a filename as an argument and return a 'channel number'. This is used in all subsequent dealings with the file. The value of the channel number depends on how many files are open already, but is always in the range 1-255. If an open function returns a value of 0, the file could not be opened because, for example, an OPENIN file does not exist already. Here are some examples of files being opened:

```
c%=OPENIN("data1")
outChan%=OPENOUT("$.dat.newData")
file=OPENUPdata$ : IF file=0 THEN PRINT "Can't find ";data$:END
```

The result of an open function is always assigned to a variable for later use. Note that OPENOUT overwrites any file of the name specified in its argument so it should be used with care.

If a new file is created by OPENOUT, 65536 (64K) bytes will be reserved for it on the disc. If a file of the same name exists already, the space allocated is the same as the length of the file.

Single-byte file operations

Once a file has been opened, it is possible to access single bytes in it using the BPUT statement and the BGET function. For example, to print the contents of a text file, the following program will suffice:

```
100 REPEAT
```

```
110   INPUT "File name: "file$
120   file=OPENIN(file$)
130 UNTIL file<>0
140 REPEAT
150   char=BGET#file
160   IF char=&0D THEN PRINT ELSE IF char<&20 THEN PRINT "."; ELSE
PRINT CHR$(char);
170 UNTIL FALSE
```

The program emulates the *TYPE utility mentioned in chapter 3. The first repeat loop asks for a filename from the user, until the name given actually exists. The second loop repeatedly gets characters from the file and prints them out. Control characters are printed as "." except for carriage return (&0D) which does a newline. Other characters are printed as themselves.

The BGET function acts in a similar way to the GET function. It returns a character code between 0 and 255, but instead of using the keyboard, it uses the file whose channel number is given in its argument. In common with the other filing system keywords which take a channel number, BGET is always followed by a hash, #.

There is a problem with the program above: when the end of the file is reached, ie all of the characters in it have been read and printed, an attempt to read more characters will generate the error 'EOF' (end of file). To overcome this, a function EOF is provided. This will return TRUE if the last character of the file has been read, and FALSE otherwise. Thus to make the program above terminate correctly, line 170 should be:

```
               170 UNTIL EOF#file
```

In addition, after an open file is no longer required, it should be closed. This frees the area of memory set aside for its buffer, so that another file may be opened. (The ADFS only allows 10 files to be opened at once). The last line of the program above should read:

```
               180 CLOSE#file
```

Closing files after use is especially important if they have been updated (written to) or opened for output, as it ensures that the buffer is copied on to the disc so that the file is kept up to date.

The BPUT statement writes a single byte to a file. The file must have been opened for update or output. For example, the program below emulates the *BUILD utility mentioned in chapter 3:

```
1000 REPEAT
1010   INPUT "File name: "file$
1020   file=OPENOUTfile$
1030 UNTIL file<>0
1040 ON ERROR CLOSE#file: PRINT''"Terminated": END
1050 line=1
1060 REPEAT
1070   PRINT RIGHT$("000"+STR$(line),4)" ";
1080   INPUT LINE ""in$
1090   in$=in$+CHR$(&0D)
1100   FOR i%=1 TO LEN(in$)
1110     BPUT#file,ASC(MID$(in$,i%))
```

```
1120   NEXT i%
1130   line=line+1
1140 UNTIL FALSE
```

The last four lines open the file for output. As mentioned above, this will delete any file of the same name already present.

Line 1040 sets the ON ERROR action for when the user presses <ESCAPE> to exit the program.

Line 1050 initialises the line number printed at the start of each input line. The second REPEAT loop prints the line number, gets a line of text, adds a carriage return to the end, and writes the lines one character at a time to the file. This continues until <ESCAPE> is pressed.

It is clear that BPUT takes two arguments: the channel number followed by the code of the character to be written to the file.

BPUT, in common with the other sequential file statements and functions, has two errors associated with it. 'Missing #' means that the # character after the keyword was omitted, and 'Channel on channel <nn>' means a channel number has been specified that does not correspond to an open file.

Writing and reading BASIC variables

BGET and BPUT are useful when processing textfiles, but not so good for dealing with the quantities available in BASIC, ie variables and constants. The PRINT and INPUT commands have special forms for use with files. Using them, it is possible to store values in files and retrieve them later on. Variables of any type may be written to and read from files. The example program below takes a list of ten names and ages from the user and places them in a file called 'ages':

```
1000 chan=OPENOUT("ages")
1010 FOR i%=1 TO 10
1020   PRINT "Name, age number ";i%": ";
1030   INPUT ""name$,age%
1040   PRINT#chan,name$,age%
1050 NEXT i%
1060 CLOSE#chan
```

It is clear that the special form of PRINT simply involves adding '#<channel>,' after the statement. When this is done, expressions that would normally be printed on the screen are sent to the file (BASIC actually uses BPUT to do this). This special form of PRINT may only be followed by a list of expressions, not print formatters such as TAB etc.

After ten names and ages have been saved, the program above closes the file as usual. The program to get the information back from the file is as might be expected:

```
2000 chan=OPENIN("ages")
2010 FOR i%=1 TO 10
2020   INPUT#chan,name$,age%
```

```
2030   PRINT "Name: "name$;TAB(20)"Age: ";age%
2040 NEXT i%
2050 CLOSE#chan
```

Again, putting '#<channel>,' after the INPUT statement causes the variables to be read from the file specified rather than the keyboard. BASIC uses BGET internally to do this. The usual INPUT prompts and print formatters are not allowed in the special file form. Although the examples above use the same variables in the PRINT and INPUT statements, there is no requirement for this. Indeed, as the PRINTed information may use arbitrary expressions it would be impossible. BASIC will automatically convert reals to integers and vice versa, the relevant type conversions being performed automatically. It is, however, illegal to read back a string into a numeric variable, or vice versa, so:

```
PRINT#file,"A string followed by ",anumber%
```

may not be subsequently read back in with:

```
INPUT#file,anumber%,astring$
```

This would fail on two counts and yield a 'Type mismatch' error.

Variable formats in files

When BASIC puts data into a file using PRINT#, it does not simply write the characters that would appear on the screen normally. (This may be done by preceding a normal PRINT statement by a *SPOOL command, though.) Instead, a compact internal format is used. Integers, reals and strings are written as a type byte, followed by the data proper. The three formats are: Integer. Type byte of &40 (64) followed by the four bytes of the integer, most significant byte first. Real. Type byte of &FF (255) followed by the four bytes of the mantissa (least significant byte first) followed by the exponent. String. Type byte of &00 followed by the length of the string (one byte) followed by the characters of the string in reverse order.

The lengths of the items are therefore 5 for integers, 6 for reals and LEN(a$)+2 for a string a$. This information is important when direct access files are used.

The program below goes through a file that has been created using PRINT# called 'printFile' and prints the types of the data it finds and the values.

```
1000 file=OPENIN"printFile"
1010 REPEAT
1020 type=BGET#file : PTR#file=PTR#file-1
1030 IF type=0 THEN PROCstr ELSE IF type=&40 THEN PROCint ELSE PROCreal
1040 UNTIL EOF#file
1050 CLOSE#file
1060 END
1070 DEF PROCstr
1080  INPUT#file,str$
1090  PRINT "String"TAB(20)str$
1100 ENDPROC
1110 DEF PROCint
1120  INPUT#file,int%
1130  PRINT "Integer"TAB(20);int%
```

```
1140 ENDPROC
1150 DEF PROCreal
1160  INPUT#file,real
1170  PRINT "Real"TAB(20);real
1180 ENDPROC
```

Line 1020 first reads the type byte of the next entry from the file, then 'goes back' a byte in the file so that the same byte will be read again by the INPUT# statement. PTR# stands for 'sequential pointer', which is a mechanism provided by the ADFS for moving to particular sections of the file. It is described in the next section.

The sequential pointer

Most of the file handling shown in examples so far has been entirely 'serial' in nature: the *TYPE-byte program reads characters from the start of the file and continues until it reaches the end, and the *BUILD-type program creates a new file and adds characters to it until the user presses <ESCAPE>. The last program in the previous section, however, illustrates a very important property of sequential files: the sequential pointer. This is used in 'direct' or 'random' access file handling.

If a file is regarded as the array of bytes mentioned above, then the sequential pointer is the subscript of the current element. It is accessed in BASIC through the PTR pseudo-variable. When a file is first opened, its pointer is set to zero, and the lines:

```
file=OPENOUT"FILE" : PRINT PTR#file
```

will print 0. Every time a byte is written to or read from a file, PTR for the file is increased by one. Thus after:

```
FOR i%=1 TO 10 : BPUT#file,i% : NEXT i%
```

the value of PTR#file will be 10, which implies that the next byte to be written is the eleventh one in the file. It is also possible to assign a value to PTR, so that reading or writing occurs at a particular position.

The sequential pointer is most useful with PRINT# and INPUT#-type file accesses. It is often convenient to treat the file as a sequence of records, where a record is a collection of fields. A field is just a value such as a string or number.

Consider a very simple stock control situation. The stock file would be made from records. Each record might consist of a part number, a description, a quantity and a price. To access any record quickly by its part number, we specify a fixed length for each record, and multiply the record number (which is the same as the part number) by the record length. This gives the position in the file (ie the PTR value) of the desired record.

If we have a maximum description length of 15 characters, the record length will be (15+2) + 5 + 6, remembering that strings take (2 plus length) bytes, integers (the quantity) take 5 bytes and reals (the price) take 6. The record length is therefore 28 bytes, and to access record (or part number) n, PTR must be set to 28*n. The program below uses these figures to give a very simple stock control database:

```
1000 REM "Stock Control"
1010 recLen=28 : maxRec=100
1020 MODE 7
1030 file=OPENUP("stock")
1040 If file=0 THEN
PROCcreate("stock",maxRec*recLen):file=OPENUP("stock")
1060 REPEAT
1070   CLS
1080   PRINT '''"1. Enter a record"'''"2. Examine a record"'''"3. Quit"
1090   PRINT''"Which one (1-3) "
1100   REPEAT
1110     INPUT TAB(16,10),choice
1120   UNTIL choice>=1 AND choice<=3
1130   IF choice=1 THEN PROCenter ELSE IF choice=2 THEN PROCexamine
1140 UNTIL choice=3
1150 CLOSE#file
1160 END
1170
1180 DEF PROCenter
1190  REPEAT
1200    CLS
1210    REPEAT
1220      INPUT TAB(0,3)"Product number",pn%
1230    UNTIL pn%>=0 AND pn%<=maxRec
1240    IF pn%=0 THEN 1310
1250    INPUT TAB(0,5)"Description",ds$
1260    ds$=LEFT$(ds$,15)
1270    INPUT TAB(0,7)"Quantity",qn%
1280    INPUT TAB(0,9)"Price",pr
1290    PTR#file=pn%*recLen
1300    PRINT#file,qn%,ds$,pr
1310  UNTIL pn%=0
1320 ENDPROC
1330
1340 DEF PROCexamine
1350  REPEAT
1360    CLS
1370    REPEAT
1380      INPUT TAB(0,3)"Product number",pn%
1390    UNTIL pn%>=0 AND pn%<=maxRec
1400    IF pn%=0 THEN 1500
1410    PTR#file=pn%*recLen
1420    type=BGET#file
1430    If type<>&40 THEN ds$="Undefined":qn%=0:pr=0:GOTO 1460
1440    PTR#file=pn%*recLen
1450    INPUT#file,qn%,ds$,pr
1460    PRINTTAB(0,5)"Description:"TAB(20)ds$
1470    PRINT'"Quantity:"TAB(20);qn%
1480    PRINT'"Price:"TAB(20);pr
1490    IF GET
1500  UNTIL pn%=0
1510 ENDPROC
1520
1530 DEF PROCcreate(file$,length)
```

```
1540  file=OPENOUT(file$)
1550  PTR#file=length
1560  CLOSE#file
1570 ENDPROC
```

The call to PROCcreate is only made if a file called 'stock' does not exist already. If it does, it is simply opened for update. The program's main loop prints a menu and performs one of the tasks: enter a record, examine a record or quit. The examine procedure has to detect if a valid item is at the record specified. It does this by testing if the first byte of the record is &40 (the integer type byte). If it is, the record is valid and its contents are printed, otherwise a dummy value is assigned and that is printed. PROCcreate simply opens the filename given in file$ for output, sets the pointer to the length (which will fill the file from position 0 to length-1 with zeros), and closes it again.

The extent of a file

The last property that may be associated with a sequential file is its extent. This is another name for 'length' and is accessed in BASIC by the function EXT#. When a file is opened for output its extent is set to zero, and is increased as information is added to the file. Files opened for update or input must already exist, and the extent is set to the length of the file.

When a file is closed, the current value of EXT# is stored as the file's length, and this value will be printed out when a *INFO command is performed in the file. A frequent use of EXT# is to check that a file to be *LOADed will fit into memory:

```
1000 REM Example of using EXT#
1010 bufferLength=2000
1020 DIM buffer bufferLength
1030 REPEAT
1040 REPEAT
1050        INPUT "Filename to load ",file$
1060        chan=OPENIN(file$)
1070     UNTIL chan<>0
1080     ok= EXT#chan <= bufferLength
1090     CLOSE#chan
1100     IF NOT ok THEN PRINT "Too large, sorry"
1110 UNTIL ok
1120 OSCLI "LOAD "+file$+" "+STR$~buffer
```

Note that when a file opened for update is appended so that its sequential pointer becomes greater than its extent, the ADFS will reserve more space on the disc for the file. In particular enough space will be allocated to allow the file's length to grow to the next multiple of 64K bytes. If there are fewer than 64K bytes free on the disc, a 'Disc full' error will be given. If there is enough space, but not in one contiguous area, a 'Compaction required' error will be given. These errors can also occur when opening a file for output.

5. SUMMARY OF ADFS COMMANDS

This chapter describes concisely all of the ADFS * commands covered in the first part of this guide. Each command is described by its syntax (ie the form that it should take) and by its action. The minimum abbreviation is also given for each command. Several abbreviations are used in the syntax descriptions, as follows:

<obspec>	This is the pathname of a single object (no wildcards)
<*obspec*>	This is the pathname of a single object (wildcards allowed)
<listspec>	This is the pathname of several objects (wildcards allowed)
<drv>	This is a disc number (usually Ø or 1)
<address>	This is a save, load, reload or execution address

Other command-specific terms are used in some cases. All other text in command descriptions should be taken literally, except round brackets which enclose items that are optional. For example:

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

This means that *LOAD is followed by a pathname that may contain wildcards (although only a single file is acted upon) and may optionally be followed by an address.

Pathnames may contain the following components:

:<drv>	Any drive number given must be at the start of the pathname
$	This signifies the root directory
&	This is another name for $
@	This means CSD, and should occur at the start of the pathname
^	This means 'the parent directory'
#	This is a wildcard standing for a single character
*	This is a wildcard standing for up to ten characters
.	This separates parts of the pathname

Examples are:

```
*Ø.fred
:1.$.*.ch#
^.^.fred.jim*
@.dirl
```

*ACCESS <listspec>(E)(L)(W)(R)(*A.)*

Changes the access permissions on all <listspec> files to the letters given.
Example: *ACCESS book* LR

*ADFS (*A.)*

Selects the ADFS and tries to reinstate the previous state of the ADFS. Also obtained by pressing <BREAK> and the A key together.
Example: *ADFS

*BACK (*BAC.)*

Sets the CSD to the previous directory, and vice versa. Used for frequent swapping between two directories.
Example: *BACK

*BYE (*BY.)*

Has exactly the same effect as *CLOSE, but is quicker to type.
Example: *BYE

*CAT (<*obspec*>) (*.)*

Lists all of the files in the first directory found conforming to <*obspec*>. Default directory is CSD or '$' if there is no CSD. The command may be written as *..
Example: *CAT $.lib*

*CDIR <obspec> (*CD.)*

Creates a new directory called <obspec>.
Example: *CDIR $.book.chl

*CLOSE (*CL.)*

Closes all open sequential files, including *SPOOL, *EXEC and BASIC OPENIN, OPENUP and OPENOUT files. This ensures that the files are up to date on the disc.
Example: *CLOSE

*COMPACT (<start page><number of pages>) (*CO.)*

Compacts the disc, converting several areas of free space into fewer, larger areas. If present, <start page> is the page address that the ADFS should use as workspace and <number of pages> gives the size of the workspace it may use. The workspace area defaults to the current screen memory.
Example: *COMPACT 1D 10

*COPY <listspec><*obspec*> (*COP.)*

Copies the files conforming to <listspec> into the directory given by <*obspec*>.
Example: *COPY * :1

*DELETE <obspec> (*DE.)*

Deletes the single file <obspec>. Gives an error if the file is locked, or if it does not exist already.
Example: *DELETE junk

*DESTROY <listspec> (*DES.)

Deletes all of the (unlocked) files conforming to <listspec>. It asks for confirmation before performing the deletion.
Example: *DESTROY temp.*

*DIR (<*obspec*>) (*DIR)

Changes the current directory to <*obspec*>. Default is '$'.
Example: *DIR ^.games

*DISMOUNT (<drv>) (*DIS.)

Closes all sequential files on the drive specified. If CSD or CSL are on that drive, the ADFS changes them to 'unset', and they must be reset using *DIR and *LIB respectively. The default <drv> is the drive of the CSD (as printed by *CAT).
Example: *DISMOUNT

*EX (<*obspec*>) (*EX)

Prints the *INFO data for all files in the directory <*obspec*). Equivalent to *INFO <*obspec*>.*. Default for <*obspec*> is CSD.
Example: *EX $

*EXEC (<*obspec*>) (*E.)

If <*obspec*> is present, it opens this file and treats its contents as keyboard input until the file is exhausted. If <*obspec*> is absent, the current *EXEC file is closed and input reverts to the keyboard.
Example: *EXEC !BOOT

*FADFS (*FA.)

Selects the ADFS as the current filing system but does not attempt to access the disc, so CSD and CSL are set to 'unset'. Also obtained by pressing F <BREAK>.
Example: *FADFS

*FREE (*FR.)

Gives the number of used sectors on the disc and the number of free sectors. Also gives the number of used bytes and number of free bytes.
Example: *FREE

*HELP (*H.)

Prints information about the ROMs in the system. The command

*HELP ADFS (or *HELP .)

will print a synopsis of the syntax of ADFS commands.
Example: *HELP

*INFO <listspec> (*I.)

Prints the information about all files conforming to <listspec>
Example: *INFO ^.*

*LCAT (*LC.)

Performs a catalogue function on the library directory.
Example: *LCAT

*LEX (*LE.)

Performs a *EX function of the library directory.
Example: *LEX

*LIB (<*obspec*>) (*LIB)

Sets the library directory to <*obspec*> or to '$' if no directory is given.
Example: *LIB @

*LOAD <*obspec*>(<address>) (*L.)

Loads the file given at <address> if this is present, or at the file's own load address if not.
Example: *LOAD data 1FØØ

*MAP (*MA.)

Displays a list of addresses of free areas on the disc and their lengths. If there are a lot of these, a *COMPACT should be executed.
Example: *MAP

*MOUNT (<drv>) (*MOU.)

Selects the drive given. Makes the CSD '$' and CSL 'unset'. Very similar to *DIR $, except that this doesn't affect CSL.
Example: *MOUNT A

*OPT1(,<n>) (*O.1)

Enables or disables the printing of files' *INFO when they are accessed. If <n> is Ø or absent, messages are disabled. If <n> is 1 they are enabled.
Example: *OPT 1 1

*OPT4(,<n>) (*O.4)

This sets the auto-boot option for the current drive. <n> determines what will be done with the file' $.!BOOT' after a <SHIFT> <BREAK> (or <BREAK> alone if the auto-boot action has been reversed):

<n>	Action
0	Nothing
1	*LOAD !BOOT
2	*RUN !BOOT
3	*EXEC !BOOT

Example: *OPT 4,2

*REMOVE <obspec> (*RE.)

Deletes the single file <obspec> but does not display an error message if the file cannot be found.
Example: *REMOVE oldfile

*RENAME <obspec><obspec> (*REN.)

Changes the file with the name of the first <obspec> to the name given in the second <obspec>. The names may be in different directories if desired. '$' may not be renamed, nor may a directory be renamed to refer to itself.
Example: *RENAME temp newfile

*RUN <*obspec*>(<parameters>) (*R.)

Loads and executes the file given by <*obspec*). The <parameters> may be read by the program (see OSARGS in chapter 6). *RUN may be abbreviated to */ and if the filename does not correspond to a built-in command, it may be executed with the command *<*obspec*>.
Example: *RUN EFORM 1 L

*SAVE <obspec><start><finish>(<exec>(<reload>)) (*S.)

Saves a file with name <obspec>. The file lies between addresses <start> and <finish>-1 inclusive. If present <exec> sets the execution address of the file, otherwise this is set to the load <start> address. <exec> may be followed by a <reload> address which sets the load address of the file (this defaults to the <start> address).
Example: *SAVE data 1D00 1F00

*SAVE <obspec><start>+<length>(<exec>(<reload>)) (*S.)

Similar to the previous command, but the area of memory saved starts at <start> and is <length> bytes long.
Example: *SAVE obj 3200+100 8000 8100

*SPOOL (<obspec>) (*SP.)

If <obspec> is present, this opens the file for output and all subsequent screen output is written to the file too. This continues until *SPOOL is executed without a filename, whereupon the file is closed.
Example: *SPOOL output

*TITLE (<string>) (*TI.)

Sets the title of the CSD to <string> which may be up to 19 characters long.
Example: *TITLE Elk+3 User Guide

6. THE FILING SYSTEM ENTRY POINTS

So far, we have only talked about dealing with the ADFS through * commands or BASIC statements. However, assembly language programmers also need some way of dealing with the ADFS and to this end six filing system calls are provided to perform most of the operations described so far (and more). These routines may also be used from BASIC (using CALL or USR) to provide facilities that aren't directly supported by the language.

Through necessity we assume that you are already familiar with 65Ø2 assembly language, as used on the Acorn Electron, and with BASIC's built-in assembler. The descriptions of the filing system routines given later in this chapter concentrate on the entry and exit conditions, so that you will know how to set up the 65Ø2 register before the routine is called, and how to interpret the results it passes back. The way in which BASIC uses each call is also mentioned.

The six routines are given names by which they are usually referred. The names have no meaning to BASIC or the assembler, and so must be defined explicitly in the program. This chapter includes several examples of calling the filing system routines from BASIC and assembly language.

Below is a table showing the filing system routines, their addresses and brief descriptions of their function.

Name	Address	Vector	Function
OSFIND	&FFCE	&21C	Open or close a file
OSGBPB	&FFD1	&21A	Read or write a group of bytes or directory information
OSBPUT	&FFD4	&218	Write a single byte to a file
OSBGET	&FFD7	&216	Read a single byte from a file
OSARGS	&FFDA	&214	Read or write file information
OSFILE	&FFDD	&212	Load or save a complete file

In addition there are three more, general purpose calls that are sometimes used with the ADFS. These are:

OSWORD	&FFF1	&2ØC	Perform low-level ADFS operations
OSBYTE	&FFF4	&2ØA	Perform miscellaneous ADFS operations
OSCLI	&FFF7	&2Ø8	Issue a * command from machine code

The column headed 'address' above gives the location of the subroutine. This address is constant for all filing systems, so performing an OSBGET uses exactly the same routine whether the cassette filing system, ADFS or network filing system is selected.

The 'Vector' column gives the address of the location in RAM that holds the address of the actual routine. This part varies between filing systems. By altering the contents of a vector, the user may intercept any routine to change the way it performs. This is a very advanced technique, and won't be described further in this guide.

Information may be passed to routines either through the 6502 registers (A, X and Y), or through a parameter block, which is simply a block of memory locations whose address is held in X and Y. The general way of using the filing system routines may be outlined as follows:

Initialise the registers (and parameter block).

JSR osroutine
Use the results in registers (and parameter block)

An example would be opening a file for input (like OPENIN):

```
LDX #name MOD &100             YX points to the filename
LDY #name DIV &100
LDA #&40                       For openin
JSR osfind                     ('osfind' set to &FFCE)
STA channel                    Save the channel number
....
.name
EQUS "$.MYFILE"+CHR$(&0D)      Filename string used by osfind
```

The first three lines set up the entry conditions, the fourth line performs the routine and the fifth line saves the result for later use.

The rest of this chapter is taken up by the detailed descriptions of the filing system calls. The abbreviations A, X and Y are used for the registers of the 6502, and C, N, V and Z refer to the status flags. Sometimes X and Y are used to form a pointer (eg to a parameter block). This is written YX and the low byte of the address is always in X.

OSFIND

Call address &FFCE (indirects through &021C).

OSFIND is used to open and close sequential files. See chapter 4 for a description of sequential files from the point of view of BASIC. Opening a file obtains a channel number (or 'handle') that is used in all subsequent processing of the file.

Closing a file tells the ADFS that it is no longer required for processing, and the area of memory used for its buffer can be deallocated. On entry the value of A tells the ADFS what to do with the file:

A=&00	Causes a file (or files) to be closed (cf CLOSE)
A=&40	Causes a file to be opened for input only (cf OPENIN)
A=&80	Causes a file to be opened for output only (cf OPENOUT)
A=&C0	Causes a file to be opened for input and output (cf OPENUP)

For the close function, the Y register must contain the channel number of the file to be closed (as returned by a previous call to OSFIND).

If the channel number is zero, all currently open files will be closed. (The command *CLOSE simply calls OSFIND with A=0 and Y=0).

For the open functions, YX must contain the address of the string holding the name of the file to be opened. The string should be a pathname terminated by a carriage return character. When A=&40 or A=&C0, the file must exist already and the name may contain wildcards. When A=&80, the file will be created if necessary, and the name must not contain wildcards.

When a file is opened for output only, a certain amount of space is reserved for it on the disc. This is 64K bytes if the file is a new one. If the file exists already, it will be overwritten but the space it previously occupied on the disc will be used for the new file. On exit from OSFIND, X and Y are preserved, C, N, V and Z are undefined and D=0. The interrupt state is preserved, but interrupts may be enabled during the operation.

For a close operation, A is preserved. For an open operation, A contains the channel number, or zero if the file could not be opened.

BASIC uses OSFIND to perform OPENIN, OPENOUT, OPENUP and CLOSE.

Example: The program below obtains a filename from the user, opens the file for output and sends the 256 characters with ASCII codes 0-255 to it. Finally it closes the file.

```
1000 REM  Example of OSFIND from machine code
1010
1020 DIM code 100
1030 osfind=&FFCE
1040 osbput=&FFD4
1050 oswrch=&FFEE
1060 osword=&FFF1
1070 len=10                          :REM  maximum length of filename
1080 FOR pass=0 TO 2 STEP 2
1090 P%=code
1100 [ opt pass
1110 .osfindExample
1120    lda #ASC"?"                   Print the prompt
1130    jsr oswrch
1140    lda #0                        Input is OSWORD 0
1150    ldx #inBlk MOD &100           YX points to parameter block
1160    ldy #inBlk DIV &100
1170    jsr osword
1180    bcc noEscape
1190    brk                           He pressed ESCAPE
1200    EQUB 17
1210    EQUS "Escape"
1220    EQUB 0
1230
1240 .noEscape
1250   lda #&80                       Open for output
1260   ldx #inBuff MOD &100           YX points to the filename
1270   ldy #inBuff DIV &100
1280   jsr osfind
1290   tay                            Put channel number in Y
1300   ldx #0                         FOR X=0 To 255
1310 .writeLoop
1320   txa                            A=X
1330   jsr osbput                     BPUT#Y,A
1340   inx                            NEXT X
1350   bne writeLoop
1360   lda #0                         Close the file
1370   jmp osfind                     (Channel still in Y)
1390
1400 .inBlk                           \ Parameter block for input
1410   EQUW inBuff                    Pointer to input buffer
1420   EQUB len                       max length of filename
1430   EQUB ASC" "                    min ASCII
1440   EQUB 255                       max ASCII
1450
1460 .inBuff
1470   EQUS STRING$(len," ")          Buffer for filename
1480 ]
1490 NEXT
1500 CALL osfindExample
```

OSGBPB

Call address &FFD1 (indirects through &021A.)

This routine will transfer a number of bytes to or from an open file, and can also be used to obtain filing system information.

For data transfers, OSGBPB acts like a series of calls to OSBGET or OSBPUT (which only transfer a single byte at a time), but is much faster. It also removes the necessity to set the file's sequential pointer, as this may be set explicitly in the OSGBPB parameter block. On entry YX points to a parameter block in memory. The contents of A define the operation to be performed.

A=&01	Write bytes to disc, using new sequential pointer value
A=&02	Write bytes to disc, ignoring sequential pointer
A=&03	Read bytes from disc, using new sequential pointer value
A=&04	Read bytes from disc, ignoring sequential pointer
A=&05	Read the CSD's title, boot up option and drive number
A=&06	Read the CSD's drive and filename
A=&07	Read the CSL's drive and filename
A=&08	Read filenames from the CSD

The two groups use the parameter block in different ways, and so they will be described separately.

The parameter block for A=&01 to A=&04 is shown below (the left hand column shows addresses relative to the base address given by YX).

00	Channel number	
01 02	Pointer to memory area used to transfer data from/to	LSB

Ø3		
Ø4		MSB
Ø5 Ø6 Ø7 Ø8	Number of bytes to transfer	LSB
Ø9 ØA ØB ØC	Sequential pointer value to be used for transfer (if used)	LSB MSB

The sequential pointer value given in bytes &Ø9 to &ØC replaces the old sequential pointer value if the calls with A=&Ø1 or A=&Ø3 are used, as if the appropriate OSARGS had been performed just before the call. On exit A, X and Y are preserved. Z, N and V are undefined, D=Ø. The parameter block is updated to show how much of the transfer actually took place. For example, if an attempt was made to read in more bytes than were left in the file, the transfer would be incomplete.

If C=Ø on exit, the transfer completed successfully and all bytes were moved. If C=1, the transfer ended before all the bytes were transferred. The state of the parameter block, in all cases, is as follows:

- The channel number is unaltered.
- The memory pointer contains the address of the byte after the last one to be transferred.
- The byte count is decremented to hold the number of bytes that weren't transferred. If C=Ø, this will also be zero, if C<>Ø, it will be non-zero.
- The sequential pointer will hold the current pointer of the file, even if it was not used by the call.

For calls with A=&Ø5 to A=&Ø8, the parameter block is:

ØØ	CSD master sequence number returned here	
Ø1 Ø2 Ø3 Ø4	Pointer to memory area used to transfer data to	LSB MSB
Ø5 Ø6 Ø7 Ø8	Number of filenames to read For A=&Ø8 only	LSB MSB
Ø9 ØA ØB ØC	File counter (search begins with first file if this is zero) For A=&Ø8 only	LSB MSB

On entry,the data pointer should hold the address of the area of memory in which the data must be stored. For A=&08, bytes &05-&08 hold the number of filenames to read (remember that a directory contains at most 47 entries), and bytes &09-&0C hold a pointer into the directory of the next filename to be read. This should be used by setting it to zero for the first call to OSGBPB and letting the ADFS update it. This is necessary as different filing systems interpret the directory pointer in different ways. On exit A, X and Y are preserved. V, Z and N are undefined, D=0. The information required is read into the data area. The data pointer is updated to point to the last byte of data transferred.

For A=&08 the file counter is decremented by one for each filename read (so will be zero if they were all read), and the directory pointer is updated to point to the next filename to be read. If C=1, not all of the names could be read and the file counter will be non-zero. If C=0, all of the names were read and the file counter will also be zero.

The format of the data is:

For A=&05
Length of title (one byte)
Title in ASCII (len bytes)
Start-up option (one byte)
Drive number (one byte)

For A=&06
Length of drive number (one byte, always &01)
Drive number in ASCII (one byte)
Length of CSD pathname (one byte)
CSD pathname in ASCII (len bytes)
For A=&07
As for A=&06 but refers to CSL

For A=&08
Length of first name (one byte)
First name (len bytes)
Length of second name (one byte)
Second name (len bytes)
.....
.....

None of the BASIC filing system commands use OSGBPB.

Example: Below is a listing of a BASIC program to print out the files in the currently selected directory using OSGBPB:

```
1000 REM Example using OSGBPB to read the directory
1010
1020 osgbpb=&FFD1
1030 DIM parBlk &0C        :REM parameter block for OSGBPB
1040 DIM filename 10       :REM Buffer for filename
1050 A%=8                  :REM OSGBPB 8 reads filenames
1060 X%=parBlk MOD &100    :REM YX points to the parameter block
1070 Y%=parBlk DIV &100
```

```
1080
1090 parBlk!9=0            :REM Start from first file in the dir
1100 REPEAT
1110   parBlk!1=filename  :REM Point to filename buffer
1120   parBlk!5=1         :REM Read one filename
1130   CALL osgbpb
1140   IF parBlk!5<>1 THEN PROCprint  :REM Print if it read
1150 UNTIL parBlk!5=1     :REM Carry on until last one read
1160 END
1170
1180 DEF PROCprint        :REM Print the filename
1190  FOR i%=1 TO ?filename
1200    VDU filename?i%
1210  NEXT
1220  PRINT
1230 ENDPROC
```

Example: The program segment below reads 128 bytes from the file whose channel number is in 'chan' to the area of memory whose address if 'buffer'.

The file must be opened (eg using chan=OPENIN"myFile") before the routine is used.

```
2000 DIM block &0C, buffer &100
2010 osgbpb=&FFD1
2020 ?block=chan
2030 block!1=buffer
2040 block!5=128
2050 A%=4
2060 X%=block : Y%=block DIV &100
2070 CALL osgbpb
2080 If block!5 THEN PRINT "Incomplete transfer"
```

OSBPUT

Call address &FFD4 (indirects through &0218).

This routine writes a single byte to an open file.
On entry A contains the byte to be written; Y contains the channel number, as returned by a previous OSFIND. The byte is written to the point in the file determined by the sequential pointer.

On exit X, Y and A are preserved, C, N, V and Z are undefined. The interrupt state is preserved, but may be enabled during the call. The file's sequential pointer is incremented. If an attempt is made to write past the end of the file (its extent), more space will be allocated if possible, so that the file may grow. The new extent will be written into the file's catalogue information when it is closed.

The BASIC statements BPUT# and PRINT# use the OSBPUT call.

Example: See the example for OSFIND for a typical use of OSBPUT.

OSBGET

Call address &FFD7 (indirects through &0216).

This routine reads a single byte from an open file.
On entry Y contains the file's channel number, allocated by a previous OSFIND. The byte is read from the point in the file determined by the sequential pointer.

On exit X and Y are preserved, A contains the byte read, N, V and Z are undefined.

If C=1 on exit, the last byte in the file has already been read (by a previous OSBGET or OSGBPB) and the character in A is invalid. If a further attempt is made to read from the file, the error 'EOF' will be generated. C=Ø implies that the end of the file hasn't been reached and the character is valid. The interrupt state is preserved, but may be enabled during the call. The sequential pointer is incremented.

BASIC uses OSBGET in its INPUT# statement and BGET# function.

Example: The program below reads bytes from a file whose channel number is stored in location 'chan' and sends them to the screen. It acts as a much simplified version of the *TYPE utility.

```
1ØØØ REM   OSBGET Example
1Ø1Ø DIM code 1ØØ
1Ø2Ø osbget=&FFD7
1Ø3Ø osfind=&FFCE
1Ø4Ø osasci=&FFE3
1Ø5Ø chan=&7Ø
1Ø6Ø FOR pass=Ø TO 2 STEP 2
1Ø7Ø P%=code
1Ø8Ø [ opt pass
1Ø9Ø .osbgetExample
11ØØ    ldy chan           Get the channel number
111Ø .osbgetLoop
112Ø    jsr osbget         Read a character
113Ø    bcs endOfFile      Passed the EOF
114Ø    jsr osasci         Print the character
115Ø    jmp osbgetLoop     Do it again
116Ø .endOfFile
117Ø    lda #Ø             Close the file after use(chan in Y already)
118Ø    jmp osfind
119Ø ]
12ØØ NEXT
121Ø INPUT "Filename to type",f$
122Ø ?chan=OPENINf$
123Ø CALL code
```

OSARGS

Call address &FFDA (indirects through &Ø214).

This is a dual-purpose routine: it reads and writes an open file's sequential pointer and extent, and obtains general filing system information.

On entry the type of function is determined by the value of Y. If this is non-zero, it is taken to be a channel number, and A determines the operation to be performed on the file. If Y=0, a general filing system operation is carried out. In both cases, X must contain the address of a block of four zero-page locations which will be used when reading or writing data.

The case when Y is non-zero is described first. The operations allowed are:

A=&00	Read file's sequential pointer (cf var=PTR#)
A=&01	Write file's sequential pointer (cf PTR#=var)
A=&02	Read file's length (cf var=EXT#)
A=&03	Write file's length (not available from BASIC)
A=&FF	'Ensure' the file on to the disc (write its buffer)

For A=&00 to A=&03 the four zero-page locations pointed to by X hold the data to be read/written, least significant byte first.

If Y is zero then the following operations are carried out according to the value in A:

A=&00 returns the type of filing system in A:

A=0 - No filing system currently selected
A=1 - 1200 baud cassette
A=2 - 300 baud cassette
A=3 - ROM pack filing system
A=4 - Floppy disc filing system (DFS)
A=5 - Econet filing system
A=6 - Teletext/Pestel Telesoftware filing system
A=7 - IEEE filing system
A=8 - ADFS

A=&01 returns the address of the rest of the command line in the zero page addresses pointed to by X (see example below).
A=&FF ensures all open files onto the disc.
On exit X and Y are preserved. A is preserved except for when A=&00 and Y=&00 on entry. C, N, V and Z are undefined, and D=0. The interrupt state is preserved, but interrupts may be enabled during the operation.

BASIC uses OSARGS in PTR# and EXT#. Note that whereas PTR# is a pseudo-variable and may be used to read or write the pointer, EXT# is a function and may only be used to read the extent.

Example: The program below, when *SAVEd and then *RUN, will read the rest of the command line and print it on the screen. For example, after running the program you could save the machine code with the line:

```
OSCLI "SAVE PRINT "+STR$~code+" "+STR$~P%
```

If you then issue the command

```
*PRINT HELLO THERE
```

the string HELLO THERE will be printed on the screen.

```
1000 REM  OSARGS Example
1010
1020 DIM code 1000
1030 osargs=&FFDA
1040 osasci=&FFE3
1050 osword=&FFF1
1060 cr=&0D
1070 workSpace=&A8
1080 FOR pass=0 TO 2 STEP 2
1090 P%=code
1100 [ opt pass
1110 .osargsExample
1120    lda #1                  Get the address of the command line
1130    ldx #workSpace          at workSpace - workSpace+4
1140    ldy #0
1150    jsr osargs
1160 .printLoop
1170    lda #5                  Read a byte from the line
1180    ldx #workSpace
1190    ldy #0
1200    jsr osword
1210    lda workSpace+4
1220    jsr osasci              Print the byte
1230    inc workSpace           Increment to next character
1240    bne noCarry
1250    inc workSpace+1
1260 .noCarry
1270    cmp #cr                 Was it the last char?
1280    bne printLoop           No, do it again
1290    rts
1300 ]
1310 NEXT
1320 OSCLI "SAVE PRINT "+STR$~code+" "+STR$~P%
1330 *PRINT HELLO THERE
```

OSFILE

Call address &FFDD (indirects through &212).

This routine performs actions on whole files. These are loading a file into memory, saving a file from memory, and loading and altering a file's catalogue information. On entry YX points to an 18-byte parameter block. The format of this parameter block is shown below:

ØØ Ø1	Address of filename, which must end with a carriage return	LSB MSB
Ø2 Ø3 Ø4 Ø5	Load address of file	LSB MSB
Ø6 Ø7 Ø8 Ø9	Execution address of file	LSB MSB
ØA ØB ØC ØD	Start address of data for save operations or length of file otherwise	LSB MSB
ØE ØF 1Ø 11	End address of data to be written (ie byte after last byte) for save operations, or file attributes otherwise.	LSB MSB

On entry A indicates the function to be performed. Possible values are:

A=&ØØ -Save a block of memory as a file using the information provided in the parameter block. The file's catalogue information (see A=5) will be written into the parameter block.

A=&Ø1 -Write the named file's catalogue information from the parameter block to the file's entry in the directory.

A=&Ø2 -Write the named file's load address from the parameter block to the file's entry in the directory.

A=&Ø3 -Write the named file's execution address from the parameter block to the file's entry in the directory.

A=&Ø4 -Write the named file's attributes (see below) from the parameter block to the file's entry in the directory.

A=&Ø5 -Read a named file's catalogue information (ie load address, execution address, length, type) from the file's entry in the directory. The object type (see below) is returned in A, the other information being written to the parameter block. (If the object is a directory, default values are returned for the catalogue information).

A=&Ø6 -Delete the named file (the file's catalogue information will be put into the parameter block).

A=&07 -Create an object. This is the same as 'Save' (A=0) except that no data is transferred. This facility can be used to create very large objects for opening for output only, overriding the default length allocation of 64K and avoiding extension delays and possible 'Compaction required' errors.

A=&FF -Load the named file. The address to which it is loaded is determined by the least significant byte of the execution address given in the parameter block. If this is zero, the address given in the parameter block is used, otherwise the file's own load address is used.

Object attributes are stored in the last four bytes of the parameter block. The most significant three bytes are undefined; the least significant byte's bits, when set, have the following meanings:

Bit	Meaning
0	The file is readable by you
1	The file is writable by you
2	Undefined
3	The object is locked for you
4	The file is readable by others
5	The file is writable by others
6	Undefined
7	The object is locked for others

In ADFS, bits 4-7 are always identical to bits 0-3. In calls which write the attributes of an object, all bits except 0, 1 and 3 are ignored. If the object is a directory, bits 0 and 1 are also ignored. Note that 'others' in the above context means other users of, say, the Econet filing system.

Object types returned in the accumulator are:

0	Nothing found
1	File found
2	Directory found

On exit X and Y are preserved, A contains the object type, C, N, V and Z are undefined. Interrupt status is preserved, but may be enabled during a call.

BASIC uses OSFILE in its SAVE, LOAD and CHAIN statements.

Example: The program below saves all of the computer's RAM from PAGE to HIMEM (or more strictly speaking from OSHWM to the bottom of screen memory) under the filename 'RAM'.

```
1000 REM  OSFILE Example
1010
1020 osfile=&FFDD
1030 osbyte=&FFF4
1040 DIM code 100
1050 FOR pass=0 TO 2 STEP 2
1060 P%=code
```

```
1070 [opt pass
1080 .osfileExample
1090    lda #&82                 Read high-order address
1100    jsr osbyte
1110    stx loadAddr+2           and store it in all four addresses
1120    stx execAddr+2
1130    stx startAddr+2
1140    stx endAddr+2
1150    sty loadAddr+3
1160    sty execAddr+3
1170    sty startAddr+3
1180    sty endAddr+3
1190
1200    clc : adc #1             Read OSHWM
1210    jsr osbyte
1220    stx loadAddr             Save it in load, exec and start
                                 addrs.
1230    stx execAddr
1240    stx startAddr
1250    sty loadAddr+1
1260    sty execAddr+1
1270    sty startAddr+1
1280
1290    clc : adc #1             Read top of user RAM
1300    jsr osbyte
1310    stx endAddr              and put in in end addr.
1320    sty endAddr+1
1330
1340    lda #0                   SAVE operation
1350    ldx #fileBlock MOD &100
1360    ldy #fileBlock DIV &100
1370    jmp osfile
1380
1390 .fileBlock
1400    EQUW filename             OSFILE parameter block
1410 .loadAddr
1420    EQUD 0
1430 .execAddr
1440    EQUD 0
1450 .startAddr
1460    EQUD 0
1470 .endAddr
1480    EQUD 0
1490
1500 .filename
1510    EQUS "RAM"+CHR$&0D
1520 ]
1530 NEXT
1540 CALL code
```

OSWORD

Call address &FFF1 (indirects through &020C).

There are four OSWORD calls recognised by the ADFS. They all require YX to point to a parameter block.

OSWORD with A=&70 - Read the master sequence number and the status byte

The master sequence number of the currently selected directory is placed in the location pointed to by YX. It is in binary decimal form in the range 0-99 inclusive. YX+1 contains a status byte, structured as shown below:

Bit number	Meaning if set
0	reserved
1	Bad free space map
2	*OPT 1,x flag - set if messages on
3	Undefined
4	Undefined
5	Winchester controller present
6	The tube is currently in use by ADFS
7	THE Tube is present

OSWORD with A=&71 - Read the free space (see *FREE)

The number of bytes of free space on the current drive is written to the parameter block pointed to by X and Y. The value given is a four-byte binary quantity, LSB first. The number is the same as the first figure printed by *FREE.

OSWORD with A=&72 - Access the disc controller (reads or writes block of bytes to or from the disc)

The parameter block is shown below:

&00	Always zero	
&01 &02 &03 &04	Start address in memory of data source of destination •	LSB MSB
&05 &06 &07 &08 &09 &0A	Command block to disc controller (see below)	
&0B &0C &0D &0E	Data length in bytes	LSB MSB

As well as the parameter block shown above, various status bytes in the ADFS workspace are used (eg a byte for the current drive number), and so this OSWORD call will only work if ADFS is the currently selected filing system (the call should not be made otherwise). If an error of any kind occurs during the execution of the command, the error number will be returned in byte 00 of the parameter block (0 will be returned otherwise). Error codes are detailed later in this description.

The command block is structured as shown below:

Bit

Byte	7	6	5	4	3	2	1	0
&05	0	0	0	Function code				
&06	X	X	X	Disc address (MSB)				
&07 &08	Disc address (LSB)							
&09	Sector count							
&0A	Unused (set to 0)							

The three bits marked X X X in byte &05 are 0Red with the current drive number to give the drive number to use. So if the current drive is 0, then any drive may be used by putting its number in these three bits. Alternatively, if the three bits are set to zero, the current drive will be used.

The function code field has these possible values:

Value	Meaning
&08	Read the sectors
&0A	Write the sector

If byte &09 is non-zero it is used as a sector count, and the data length parameter (bytes &0B-&0E of the main parameter block) is ignored.

For example: to read &1234 bytes starting from sector number &000567 of the current drive, loading into memory at location &FFFF3000 (high bytes FFFF indicating the host machine), the parameter block would be set up as shown below.

Byte	Value	Meaning
&00	&00	Controller number
&01	&00	Load address (LS byte)
&02	&30	
&03		
&FF		
&04	&FF	Load address (MS byte)

&05	&08	Read command
&06	&00	Disc address (MS byte)
&07	&05	
&08	&67	Disc address (LS byte)
&09	&00	
&0A	&00	
&0B	&34	Data length (LS byte)
&0C	&12	
&0D	&00	
&0E	&00	Data length (MS byte)

This call, if made immediately after a disc error of some kind (including a data error in sequential filing) returns error information (in the parameter block) as follows:

Byte		
&00	Disc address where error occurred,	(LSB)
&01	including drive number in three MS bits	
&02	of byte 02	(MSB)
&03	Disc error number, top bit set = valid address	
&04	Channel number of file where error occurred	

Thus if the value of byte &03 has its top bit set, the lower seven bits contain the error number and the first three bytes contain the disc address where the error was detected. If byte three has its top bit unset, byte &04 contains the channel number on which the error occurred and the disc address is invalid.

OSBYTE

Call address &FFF4 (indirects through &020A).

There are several OSBYTE calls that can be used with the ADFS. These are described in this section.

OSBYTE with A=&77 - Close the *SPOOL and *EXEC files

The command *CLOSE will close all files, including the *SPOOL and *EXEC ones. However, it is sometimes desirable to ensure that there is no *SPOOL or *EXEC file active without affecting other open files. This OSBYTE may be used, either as an *FX command or as an OSBYTE from machine code. Examples are

```
*FX119
```

as a command and from assembly language:

```
1100   lda #&77
1110   jsr osbyte
```

OSBYTE with A=&7F - Check for end of file

This call is only used from assembly language as it returns a result in the X register. (It could be used from BASIC using the USR function, but EOF# provides a more convenient way of detecting the end of a file). On entry, A should contain &7F and X (not Y as normal) should contain the file's channel number. On exit, X will be non-zero if the end of the file has been reached, or zero if not.

Since OSBGET and OSGBPB indicate the end of file when they are called, this OSBYTE is not needed very often. It is provided so that language such as BASIC can provide an end of file function (EOF#). Below is an example of the routine being called from assembly language:

```
2010   lda #&7F
2020   ldx chan
2030   jsr osbyte
2040   txa
2050   bne eof
2060   \ rest of code
2070 ....
2300 .eof
2310   \ end of file code
```

OSBYTE with A=&8B - Perform a *OPT command

This call provides the equivalent function to *OPT from machine code. For the command

```
*OPT <x>,<y>
```

the X register would contain <x>, Y would contain <y> and A would contain &8B. For example, to enable file information to be printed when files are accessed (with *OPT 1,1), this would be executed:

```
3230   lda #&8B
3240   ldx #1
3250   ldy #1
3260   jsr osbyte
```

OSBYTE with A=&8F and X=&12 - Start up a filing system

This call may be used from assembly language or as a *FX command to select a particular filing system. As a command it has the form:

```
*FX143,18,<filing system>
```

where <filing system> is the number given in the description of OSARGS. For example, to select the Econet filing system, the command:

```
*FX143,18,5
```

could be used instead of the much more convenient *NET. However, from assembly language, it is often easier to use the OSBYTE rather than a * command to select a filing system. For example, to select the ADFS:

```
2130   lda #&8F
2140   ldx #18
```

```
2150   ldy #8
2160   jsr osbyte
```

Note that only filing system numbers 4 and above may be selected using this method.

OSBYTE with A=&FF - Set BREAK/ADFS options

This call is used to read and write the BREAK/ADFS options byte. This byte is used when <BREAK> is pressed and when the ADFS is entered. It controls: the display mode used after <BREAK>, whether an auto-boot requires <SHIFT> to be pressed or not, what 'stepping speed' the disc drives use, and whether write precompensation is applied when storing data on discs.

The eight bits of the byte have the following meanings:

BITS Ø-2 — The display mode to use after <BREAK>. This is a three-bit number between Ø and 7. The default is 7 (which is the same as MODE 6 on the Electron).

Bit 3 — Auto-boot. If this bit is 1, pressing <SHIFT> <BREAK> will cause the filing system (eg the ADFS) to auto-boot (do something with the file '!BOOT'), and pressing <BREAK> alone will not cause an auto-boot. If the bit is zero, the action is reversed and pressing just <BREAK> will cause the auto-boot action. The default is <SHIFT> <BREAK> to cause an auto-boot.

Bits 4-5 — These select the speed at which the read/write head of the disc steps between tracks. The possible values are:

1	1	6
1	Ø	12
Ø	1	2Ø
Ø	Ø	3Ø

The default is 6mS, suitable for the fast PLUS 3 drive.

Bit 6 — This selects whether write precompensation is required when writing data to the disc. A value of 1 means it is required and Ø means it isn't. The Plus 3's 3.5 inch drive does require write precompensation, and the default value of the bit is 1.

Bit 7 — This is unused by the current Electron operating system and the ADFS.

When <CTRL> <BREAK> is pressed, the byte is set to &FF, so all of the options are reset to their default values.

To set the value of the byte, a *FX255 command may be used. For example, suppose it is required to set the start-up mode to 3 and the disc speed to 12mS with write precompensation. The values of the bits are:

```
Bit           7 6 5 4 3 2 1 Ø
Value         Ø 1 1 Ø 1 Ø 1 1
```

This is &6B in hexadecimal or 1Ø7 in decimal. Thus the required command is:

```
*FX255,1Ø7
```

The options will be remembered until the machine is switched off, or <CTRL> <BREAK> is pressed. From assembly language, the code is:

```
LDA #&FF
LDX #&6B
LDY #Ø
JSR OSBYTE
```

To read the current option from BASIC, the USR function must be used as follows:

```
1ØØØ A%=&FF
1Ø1Ø X%=Ø
1Ø2Ø Y%=&FF
1Ø3Ø X%=(USR &FFF4 AND &FFØØ)/&1ØØ
```

X% will contain the value of the options byte. From machine code the equivalent is:

```
LDA #&FF
LDX #Ø
TAY
JSR OSBYTE
```

Upon exit, the X register will contain the options byte. It is also possible to set the options using one of the utility programs (*SETPARAMS) mentioned in chapter 3. The program performs the *FX call automatically in response to answers given by the user. It also creates a file called 'BootParams' which will set the option byte when executed. A call to this file may be included in a '!BOOT' *EXEC file.

OSCLI

Call address &FFF7 (indirects through &Ø2Ø8)

This routine has only one entry condition: YX points to a command string terminated by a carriage return. The string will be passed to the operating system and thence to the ADFS if necessary, exactly as if it had been typed as a *command in BASIC. On exit, all registers are undefined.

The program below prints a * prompt, accepts a line of input from the keyboard and sends this to OSCLI, until <ESCAPE> is pressed, or <RETURN> at the start of the line. Note that commands set to OSCLI do not require a * at the front: this is just a marker used to let BASIC know that an operating system command has been entered.

```
1ØØØ REM Example using OSCLI
1Ø1Ø buffLen=4Ø                  :REM  Length of input buffer
```

```
1020 DIM code 100                :REM  Space for the machine code
1030 DIM buffer buffLen          :REM  The space for the input buffer
1040 osnewl=&FFE7                :REM  MOS entry points
1050 oswrch=&FFEE
1060 osword=&FFF1
1070 osbyte=&FFF4
1080 oscli =&FFF7
1090 FOR pass=0 TO 2 STEP 2
1100 P%=code
1110 [ opt pass
1120 .oscliExample
1130   lda #ASC"*"                Prompt with *
1140   jsr oswrch
1150   ldx #inBlock MOD &100      Get a line of input using OSWORD 0
1160   ldy #inBlock DIV &100
1170   lda #0
1180   jsr osword
1190   bcs escape                 ESCAPE was pressed
1200   tya
1210   beq exit                   Null line, so exit
1220   ldx #buffer MOD &100       Point YX at the command string
1230   ldy #buffer DIV &100
1240   jsr oscli                  Execute the command
1250   jmp oscliExample           Do it again
1260 .escape
1270   lda #&7E                   Acknowledge the ESCAPE
1280   jsr osbyte
1290   jsr osnewl                 Print a newline
1300 .exit
1310   rts                        Return to the language
1320 .inBlock
1330   EQUW buffer                OSWORD 0 parameter block
1340   EQUB buffLen
1350   EQUB &20                   Min ascii
1360   EQUB &FF                   Max ascii
1370 ]
1380 NEXT
1390 CALL code
```

Note that any * command may be executed and if an error occurs (eg you try to load a file that doesn't exist), the program will return to BASIC which will print the error message.

7. ERROR MESSAGES

In this chapter all of the ADFS error messages are explained. The error codes associated with the errors are also given. These may be used to trap certain classes of fault in a BASIC ON ERROR routine. For example, to trap a 'Bad command' error, a program might contain the lines:

```
10 ON ERROR GOSUB 2300
........
........
2300 IF ERR=&FE THEN PRINT "Illegal command, try again" ELSE VDU 7
2310 RETURN
```

This causes the prompt which asks the user to try again to be printed if the error number is &FE ('Bad command'), or the bell to be sounded otherwise (indicating that the user has pressed <ESCAPE>, for example).

Errors in alphabetical order

Aborted (Error &92)

Something other than YES (or Yes, or yes, etc) <RETURN> has been typed in in response to a confirmation prompt, eg

Destroy?

Access violation (Error &BD)

An attempt has been made to read (or load) a file with the 'R' attribute not set, or to write to a file with the 'W' attribute not set. An associated error is 'Locked', which is caused by an attempt to overwrite a file with its 'L' attribute set.

Already open (Error &C2)

An attempt has been made to delete (or save a new version of) a file which is open for sequential access. It also occurs if an attempt is made to open a file which is already open (unless both 'opens' are for input only). Use *CLOSE (or CLOSE#0 from BASIC) to ensure that all files are closed.

Already exists (Error &C4)

An attempt has been made to create a new object with the same name as an already existing object. This includes *CDIR and *RENAME but not *SAVE or BASIC's SAVE, which will overwrite the existing file (as long as it isn't locked).

Bad command (Error &FE)

The command given was not recognised by the ADFS, nor was it found as a utility in the CSD or the current library. This can occur after *RUN, */ or just *<*obspec*>.

Bad FS map (Error &A9)

Either the ADFS's workspace in RAM has been corrupted, or disc sectors Ø or 1 are corrupt. If you get this, try <CTRL> A <BREAK>. If the error still occurs, the disc has become corrupt and you should discard it and start using the backup. You did keep a backup, didn't you?

Bad name (Error &CC)

An illegal filename was used, ie one including $ (dollar) or : (colon) outside the context of a root specification, or with a zero length component of a pathname, or other special characters in the wrong context, eg

```
*EX $$ *DIR FILE:ONE *DIR DIR..XDIR1 *EX A@B *EX A B
```

Bad opt (Error &CB)

An invalid argument has been assigned to a *OPT command. The only valid numbers that may follow a *OPT are Ø, 1 and 4.

Bad parms (Error &94)

Invalid parameters were given with a *COMPACT command to specify the RAM area to be used. The start page specified should not be below BASIC's PAGE pseudo variable, and the length should not cause the upper limit of the area to be greater than &8ØØØ).

Bad rename (Error &BØ)

An attempt has been made to rename a directory in such a way as to produce an illegal directory structure, eg

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

so that directory A contains a reference to itself. This is illegal.

Bad sum (Error &AA)

Some of the information which the ADFS keeps in RAM has been corrupted, which prevents the ADFS from being able to close a file or read or write to it. The system must be restarted by a hard break.

Broken directory (Error &A8)

An attempt has been made to access a directory which is in some way corrupt and as such should not be accessed. It may be possible to recover from this error by pressing <CTRL> A <BREAK>, but if the error is given repeatedly the disc is in an inconsistent state and should be reformatted if possible.

Can't delete CSD (Error &96)

An attempt has been made to delete the currently selected directory, which is illegal.

Can't delete library (Error &97)

An attempt has been made to delete the current library, which is illegal.

Channel on channel <nn> (Error &DE)

A sequential file operation has been attempted with an unassigned file handle. <nn> is the illegal channel number in decimal. All sequential file operations should begin with the file being opened (using one of the BASIC functions or the call OSFIND) and end with the file being closed (with CLOSE or OSFIND with A=0).

Compaction required (Error &98)

A creation operation (eg SAVE, *CDIR, *COPY) has been attempted on a disc where the free space has become too fragmented. If you *COMPACT the disc, many small areas of free space will be combined into fewer larger ones, one of which should be large enough to hold the new file. You might have to issue more than one *COMPACT command before a large enough area becomes available.

Data lost on channel <nn> (Error &CA)

This is given when a hardware or RAM problem occurs when the ADFS is accessing a sequential file. <nn> is the channel number of the affected file in decimal. You should issue a *CLOSE command and reset the system.

Dir full (Error &B3)

An attempt has been made to create a new object in a directory already containing 47 entries, which is the maximum number it can hold. This limit should not be a problem because of the way directories may contain other directories which may also contain up to 47 entries.

Dir not empty (Error &B4)

An attempt has been made to delete a directory which still contains objects. You can't do this as it would leave files on the directory which can't be accessed by a pathname.

Disc changed (Error &C8)

A disc has been accessed without being *MOUNTed or selected using *DIR. Before a disc is removed from a drive, it should be *DISMOUNTed. Then when another disc is inserted, it should be *MOUNTed to ensure that the ADFS knows that the disc has been changed.

Disc error <nn> at:<drv>/<sector number> (Error &C7)

A fault on the disc was detected by the disc controller during the last operation. <nn> is the error code, <drv> is the drive number, <sector number> is the start sector number (all in hexadecimal) of the file in which the error was discovered (if appropriate). Some error codes are:

48 - Cyclic redundancy check error
5Ø - Sector not found
61 - Bad address
63 - Volume error
65 - Bad drive
67 - Bad command

(the error codes are hexadecimal values, the same as would be returned by an OSWORD &73 call). Errors 48 and 5Ø are not recoverable - the bad area of the disc should be made inaccessible by renaming the file being accessed to, for example, 'badfile'. Alternatively, as many files as possible should be copied on to a good disc and the old disc reformatted.

The other errors are all traceable to errors in the command block of an OSWORD &72 call - in particular, errors 61 and 63 indicate that an attempt was made to read off the end of the disc.

Disc full (Error &C6)

There is not enough free space on the drive to carry out the requested operation. This includes *CDIR, *SAVE (and BASIC's SAVE), and opening new files or extending existing files. When this happens you should format a new disc and move some of the files from the full disc on to it. They can then be deleted to obtain more free space.

Note that a new file opened for output requires at least 64K bytes to be free on the disc. Also, when a file opened for update is extended (by setting its pointer past its extent) extra space is allocated to the file making its length a multiple of 64K bytes. If there is no room for this, 'Disc full' will be given.

Disc protected (Error &C9)

An attempt has been made to write to, or to delete a file on, a disc with the write protect tab in the 'protected' position. It may be set to 'unprotected' by sliding it to the opposite side with, for example, a ball-point pen (3.5"), or removing the write protect label (5.25").

EOF on channel <nn> (Error &DF)

This occurs when an attempt is made to read a byte (using BASIC's BGET or OSBGET or OSGBPB from machine code) from a file whose end has already been reached. The end of file condition is indicated by the C flag being set on return from OSBGET or OSGBPB. You can check for the end of a given file using OSBYTE &7F (or EOF# in BASIC). The channel number <nn> is in decimal.

Locked (Error &C3)

An attempt has been made to remove, rename or overwrite an object which is locked. Use *ACCESS to unlock a file. Note that directories are locked when they are created; normal files aren't.

Map full (Error &99)

The free space map is full, ie there are 8Ø address/length entries in it. The disc should be *COMPACTed to reduce the number of entries, otherwise it may not be possible to save further information to it.

Not found (Error &D6)

The object referred to was not found. This could be given in response to a *DELETE (but not *REMOVE) or *LOAD. It may also be given when no filenames are found to match a wildcard specification in a command such as *INFO.

Not open for update on channel <nn> (Error &C1)

An attempt has been made to write to a random access file which is only open for reading. <nn> is the channel number in decimal. If you want to update an existing file, OPENUP (OSFIND with A=&CØ) should be used. If you want to write to a new file, OPENOUT (OSFIND with A=&8Ø) should be used.

Outside file on channel <nn> (Error &B7)

An attempt has been made to set the pointer of a file which is only open for reading to a value beyond the end of the file. <nn> is the channel number in decimal. Use OPENUP (OSFIND with A=&CØ) if you want to extend a file by setting its pointer past its extent.

Too many open files (Error &CØ)

An attempt has been made to open an eleventh file. Only ten files may be open at once. Always remember to CLOSE (OSFIND with A=&ØØ) a file after use.

Wild cards (Error &FD)

A wildcard character ('*' or '#') was found where a unique object specification is required, eg in *DELETE, *SAVE, *CDIR.

Won't (Error &93)

An attempt has been made to *RUN a file whose load address is &FFFFFFFF. You are prevented from doing this as the chances are that the address will 'wrap round' and some of the file will be written over the start of the important workspace stored at location &ØØØØ onwards.

Errors in numerical order

Hex	Decimal	
&92	146	Aborted
&93	147	Won't
&94	148	Bad parms
&96	150	Can't delete CSD
&97	151	Can't delete library
&98	152	Compaction required
&99	153	Map full
&A8	168	Broken directory
&A9	169	Bad FS map
&AA	170	Bad checksum
&B0	176	Bad rename
&B3	179	Dir full
&B4	180	Dir not empty
&B7	183	Outside file
&BD	189	Access violation
&C0	192	Too many open files
&C1	193	Not open for update
&C2	194	Already open
&C3	195	Locked
&C4	196	Already exists
&C6	198	Disc full
&C7	199	Disc error
&C8	200	Disc changed
&C9	201	Disc protected
&CA	202	Data lost, channel
&CB	203	Bad opt
&CC	204	Bad name
&D6	214	Not found
&DE	222	Channel
&DF	223	EOF
&FD	253	Wild cards
&FE	254	Bad command

8. TECHNICAL INFORMATION

General

Sectors 0 and 1 on a drive contain the total number of sectors on the drive, the boot option number, and the free sector gap list. Sectors 2 to 6 inclusive are the root directory.

The free space map

The free space map (FSM) is stored in sectors 0 and 1 on each drive. The format is:

Sector 0	
0	Disc address of first free space (LS byte)
1	Disc address of first free space
2	Disc address of first free space (MS byte)
3	Disc address of second free space (LS byte)
4	Disc address of second free space
5	Disc address of second free space (MS byte)
6	Disc address of third free space (LS byte)
	:
	:
	:
	etc for all other free space up to 82 entries
	:
	:
246	Reserved
247	Reserved
248	Reserved
249	Reserved
250	Reserved
251	Reserved
252	Total number of sectors on disc (LS byte)
253	Total number of sectors on disc
254	Total number of sectors on disc (MS byte)
255	Checksum on free space map, sector 0
Sector 1	
0	Length of first free space (LS byte)
1	Length of first free space
2	Length of first free space (MS byte)
3	Length of second free space (LS byte)
4	Length of second free space
5	Length of second free space (MS byte)
6	Length of third free space (LS byte)
	:
	:
	etc for all other free space up to 82 entries
	:
246	Reserved
247	Reserved
248	Reserved
249	Reserved
250	Reserved
251	Disc identifier
252	Disc identifier

253	Boot option number as set by *OPT4,<n>
254	Pointer to end of free space list
255	Checksum on free space map, sector 1

The disc addresses and length are in sectors. The free space map is stored in RAM from &0E00 to &0FFF when ADFS is selected, so the first free space pair is held at &0E00, the second at &0E03, and so on.

Directory information

A directory consists of five contiguous sectors on the disc drive (1280 bytes). The first byte of the first sector contains the master sequence number for the directory. The next four bytes contain the text 'Hugo' to uniquely identify the sector as the start of a directory.

The directory entry for the first file starts at the sixth byte of the first sector. There may be up to 47 entries, each entry consisting of 26 bytes as follows:

Name and access string	10 bytes
Load address	4 bytes
Execution address	4 bytes
Length in bytes	4 bytes
Start sector on drive	3 bytes
Sequence number (in BCD)	1 byte
Total	26 bytes

The top bits of the first four characters of the name contain the access flags in the order 'R', 'W', 'L', 'D'. Thus if you are reading filenames straight from the directory instead of using OSGBPB, each character in the name should be ANDed with &7F.

The last 58 bytes of the directory are: one zero byte, a copy of the master sequence number, 19 bytes of directory title, three bytes for disc address of the directory's parent (^), the filename of the directory, and another copy of the four bytes 'Hugo'.

The master sequence number is incremented every time the directory is rewritten. When an entry is made or changed in the directory the entry's sequence number is set to the directory master sequence number (which is then incremented).

The currently selected directory is stored in RAM from &1200 to &16FF when ADFS is selected. The end of the list of entries is denoted by a 0 in the first character position of the first unused entry, hence the 0 after entry 47. A store map of locations &1200 to &16FF is shown below.

1200		Master sequence number (in BCD)	
1201 1204		Text to identify the directory	
1205 121E		First directory entry	
121F		Second directory entry	
		End of last directory entry	
	0	Last entry marker	
('garbage')			
16CB	0	Last entry marker (dummy)	
16CC 16D5		Directory name	
16D6 16D8		Parent pointer	(LSB) (MSB)
16D9 16EC		Directory title	
16ED 16F9		Reserved	
16FA		Master sequence number (in BCD)	
16FB 16FE		Text to identify the directory	
16FF		Reserved	

Location &1200 in the above example contains byte 0 of the first sector of the directory (sector 2 for directory $). Location &16CB contains byte &CB of the fifth sector of the directory (sector 6 for directory $).

Version numbers

V 1:0	This is the original Acorn released with the Acorn Plus 3.
V 1:1	Released by PRES Sep. '87. ZYSysHelp, disc protect and compact bugs fixed. Also, with extra software, 256k RAM disc support in place of Winchester code.
V 1:2	As V 1:1 but leaves PAGE set to &E00. Requires two banks of sideways RAM.

APPENDIX A

Fault finding

This appendix describes some of the problems that may be encountered when installing and using the ADFS or the Advanced Plus 3.

Failure of the hardware

When the ADFS / AP3 is added to the Electron, the start-up message should be:-

```
Acorn Electron

PRES ADFS

BASIC

>_
```

If the 'PRES ADFS' part is missing, something is preventing the Electron from detecting the presence of the ADFS. This may be due to a bad contact in the edge connector. After disconnecting the power and other leads, remove the AP3. Clean the edge connector fingers at the bottom of the AP3 using a non-abrasive pencil eraser and a soft cloth. Reconnect the AP3 and power up again. If the ADFS message still doesn't appear, take the AP3 to your dealer or contact PRES.

If, when powering up, nothing at all is displayed on the screen, try cleaning the edge connector as described in the previous paragraph and try again. Also, check that the Electron and Plus 1 starts OK without the AP3 connected. If the Electron and Plus 1 appears to be functioning, and a blank screen is still obtained when the AP3 is fitted, take the AP3 to your dealer or contact PRES. If you get a blank screen without the AP3, take both it, your Electron and Plus 1 to your dealer. Never attempt to service the AP3, Electron or Plus 1 yourself. There are no user-serviceable parts inside, and you run the risk of damage to the device and injury to yourself by attempting any repair.

Problems when using the ADFS

The ADFS is a very complex piece of software that can only be understood fully after much use. Sometimes you will encounter a problem, usually in the form of an error message, which you don't understand. This section describes common examples of these problems.

1. 'Bad program' after CHAIN or LOAD. This is caused by the file you are trying to CHAIN or LOAD not being a BASIC program. For example, trying to CHAIN the machine code utility EFORM will cause this error.

2. *RUN causes the machine to 'crash'. This is the opposite to the last problem: the file you are attempting to *RUN (or */ or *<file>) is not a proper machine code program. The exact effect of *RUNning, say, a BASIC program is indeterminate and you should press <CTRL> <BREAK> to ensure that the system is reinitialised properly.

3. 'No directory' error. This occurs when the ADFS has been started

without accessing the disc (eg by *FADFS) and has no current directory. Typing a *DIR or *MOUNT command will cause a directory to be read and the CSD to be set. Alternatively, typing *. will cause the root directory '$' to be read in, and this may then be used even though the CSD is still 'unset'.

4. Repeated 'Disc error's. This error occurs either when the disc has not been formatted, or when the disc has been physically damaged. If you are sure that the disc is formatted, then there may be a scratch or fingerprint on the surface of the disc. In this case you should format a new disc, copy as many files as possible on to it and then discard the damaged disc.

If a 'Disc error' is given while you are using an external 5.25 inch drive, there are two other possibilities: the disc has been inserted the wrong way (this cannot be done with 3.5 inch discs), or you may be trying to read or write to a disc that was formatted on a BBC Microcomputer, using its DFS. This is not compatible with the ADFS.

5. 'Not found' and 'Bad command'. These are often caused by the user's not keeping track of the current 'environment', ie the CSD and CSL and drive. For example, you may think that CSD is '$', and wish to go into '$.book' using:-

```
                    *DIR book
```

However, if the CSD is already 'book', this will cause a 'Not found' error, as there is probably not a directory in '$.book' which is also called 'book'. Similarly, 'Bad command' can be given because the library directory is not what you think it is.

The best way to check on the current environment is to type *. or *CAT, and <CTRL> A <BREAK> will ensure that CSD is '$' and CSL is either '$' or a file that starts with '$.LIB' if one is present.

6. 'Not found' on <SHIFT> <BREAK>. This occurs when an auto-boot is attempted on a disc which has its boot option set to 1 when there is no file called '$.!BOOT' on the disc. Similarly, if the option is 2 a 'Bad command' error may be given, and if the option is 3 a 'File not found' error may be given. To prevent this, disable the boot action of the disc by typing *OPT4,Ø.
(Note that when one of the above mentioned errors is given, the Electron will 'hang-up', and you must press <BREAK> to start again.)

7. Non-appearance of the 'PRES ADFS' message. This can happen after running a program which corrupts the ADFS workspace from location &EØØ up. Examples are BASIC programs copied down to this address (see chapter 4) and certain Acornsoft cartridge programs. When <BREAK> is pressed after such a program is run, the ADFS recognises that it is no longer in a valid state and automatically prevents itself from being selected. To re-enable the ADFS, type *FX2ØØ, 2 <RETURN> and then press <CTRL> <BREAK>, or turn the machine off then on.

Sometimes, the message 'Bad FS map' will be given instead, and you should press <CTRL> <BREAK> to start the ADFS again.

Note that you can deliberately disable the ADFS, for example if you want to run a tape-based game which is incompatible with the ADFS. See 'Disabling the ADFS' in Appendix C for details.

APPENDIX B

Methods of calling the ADFS

As noted in chapter 2, there are many ways in which the ADFS may be called. When it is entered by pressing <BREAK>, the setting of CSD, CSL and whether the disc is accessed depends on the other keys that are also pressed, and also whether the ADFS has already been used since the last hard reset. Possible keys are A for *ADFS, F for *FADFS, <SHIFT> for auto-boot, and <CTRL> for hard reset (as if the machine had just been turned on). Note that pressing <CTRL> and <SHIFT> is equivalent to pressing only <CTRL>.

The table below gives the action for every possible combination of keys.

	After hard reset			After a previous ADFS		
	CSD	CSL	Disc?	CSD	CSL	Disc?
<BREAK>	Unset	Unset	No	Prev	Prev	Yes
<SHIFT> <BREAK>	$	Unset	Yes	$	Prev	Yes
<CTRL> <BREAK>	Unset	Unset	No	Unset	Unset	No
A <BREAK>	Unset	Unset	No	Prev	Prev	Yes
<SHIFT> A <BREAK>	$	Unset	Yes	$	Prev	Yes
<CTRL> A <BREAK>	$	$ or LIB*	Yes	$	$ or LIB*	Yes
F <BREAK>	Unset	Unset	No	Unset	Unset	No
<SHIFT> F <BREAK>	Unset	Unset	No	Unset	Unset	No
<CTRL> F <BREAK>	Unset	Unset	No	Unset	Unset	No
*ADFS	Unset	Unset	No	Prev	Prev	Yes
*FADS	Unset	Unset	No	Unset	Unset	No

where CSD means 'currently selected directory', CSL means 'currently selected library' and Prev means 'same as last time the ADFS was selected'.

Although there are 22 different combinations, many of the possibilities are duplicated. The important things to note are:

Anything involving *FADFS or the F key will leave the CSD and CSL unset and will not access the disc.

Anything involving <CTRL> <BREAK> is the same as switching the machine on.

Actions involving *ADFS or the A key have the same effect as *FADFS after a hard reset, or will restore the ADFS to its previous state if it has been used since the last hard reset

APPENDIX C

The Welcome disc

You will have received a 'Welcome disc' containing the ADFS utilities. It's contents may be divided into three sections:

1. The Welcome programs themselves. These are BASIC programs which introduce some of the facilities of the Electron. They are described in the Electron User Guide (the programs are also on the Welcome cassette which comes with every Acorn Electron). To see a menu of the Welcome programs, insert the disc into the drive, hold down <SHIFT>, press and release <BREAK>, and finally release <SHIFT>.

2. The utility programs. We have already used two utilities, EFORM and *DIRCOPY. There are several more on the disc. To see a complete list, follow the instructions given earlier in Chapter 3.

3. The utility documentation. The instructions for using the utility programs are also stored on the Welcome disc. They are accessible through the utilities menu described earlier. It is envisaged that the instructions will only need to be consulted once or twice for each utility, as the programs themselves provide quite a lot of 'help' to users.

Using the ADFS in MODEs Ø to 3

When a command that accesses the disc drive is issued, and one of MODEs Ø to 3 is in use, the screen will go blank (or flicker) for the duration of the access. More precisely, the screen will clear to the current background colour. As soon as the access is completed, and the drive light goes out, the screen will be restored to its former state.

The reason for this blanking is the very high speed at which data is transferred between the disc and the computer. In order to 'keep up' with the disc, the Electron has to 'forget' temporarily about keeping the picture on the screen. This only happens in MODEs Ø to 3 because these require more processing time than the others.

Another side-effect of using the disc (in any mode) is that interrupts are disabled. The implication of this is that the cursor stops flashing, and the Electron's timers (eg the one accessed by BASIC's TIME pseudo-variable) are not updated. If you are using discs a lot, you should not rely on TIME being accurate over long periods. Also, sound processing stops, colours stop flashing and the keyboard (including the ESCAPE key but excluding the BREAK key) is disabled.

Disabling the ADFS

Occasionally you will want to disable the ADFS and Plus 3. An example is when you want to run a game from tape which does not work with the Plus 3 fitted. One way to disable the Plus 3 is to remove it from the Electron. However, repeated fitting and removing is not recommended, so a 'software solution' is provided on the Welcome disc.

Start up the ADFS by inserting the Welcome disc, holding down <CTRL> A and pressing and releasing <BREAK>. Then type the command:

```
*NOADFS
```

When the red drive light goes out, press and release <BREAK>. the 'Pres ADFS' message will not appear, and for all practical purposes the Plus 3 is no longer active. The Electron can now be used exactly as it was before the Plus 3 was installed.

To re-enable the Plus 3, simply switch the computer off for a second, then switch it on again, or type *FX200,2 <RETURN> then hold down the <CTRL> key, press and release <BREAK>, and then release the <CTRL> key. The ADFS will reappear.

INDEX

NOTES