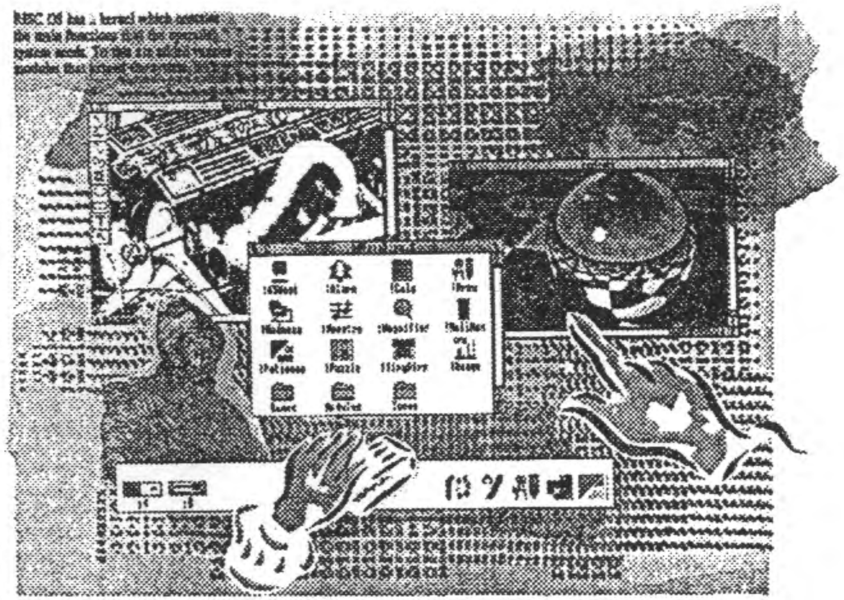
RISC OS

PROGRAMMER'S REFERENCE MANUAL

Volume III



Acorn

Published by Acorn Computers Technical Publications Department

The product described in this manual and products for use with it are subject to continuous development and improvement. All information of a technical nature and particulars of the product and its use (including the information and particulars in this manual) are given by Acorn Computers Limited in good faith. However, Acorn Computers Limited cannot accept any liability for any loss or damage arising from the use of any information or particulars in this manual.

This product is not intended for use as a critical component in life support devices or any system in which failure could be expected to result in personal injury.

If you have any comments on this manual, please complete the form at the back of the manual, and send it to the address given there.

Acorn supplies its products through an international dealer network. These outlets are trained in the use and support of Acorn products and are available to help resolve any queries you may have.

Within this publication, the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

Published by Acorn Computers Limited
ISBN 1 85250 113 2
Edition 1
Part number 0470,293
Issue 1, October 1991

Contents

Part 3 – Filing systems 3-1

Part 4 – The Window manager 4-81

Part 5 – System extensions 4-379

Appendices and tables 6-293

Part 3 – Filing systems

26 Introduction to filing systems

Filing systems

RISC OS uses filing systems to organise and access data held on external storage media. Several complete filing systems are provided as standard:

- Advanced Disc Filing System (ADFS) for use with both floppy and hard disc drives.
- Network Filing System (NetFS) for controlling your access to Econet file servers
- RAM Filing System (RamFS), for making memory appear to be a disc
- NetPrint, for controlling Econet printer servers.

Other modules provide extra filing systems:

- the Desktop filing system contains resource files needed by the Window manager and ROM-resident Desktop utilities
- the SystemDevices module provides various system devices.

FileSwitch

A module called FileSwitch is at the centre of all filing system operation in RISC OS. Although separate from the kernel, it is compiled with it, and is very closely bound to it.

FileSwitch provides a common core of functions used by all filing systems. It only provides the parts of these services that are device independent.

Obviously, FileSwitch cannot know how to control every single piece of hardware that gets added to the system. The device dependent services that control hardware are provided by separate modules, which are the actual filing systems.

Switching between filing systems

One of the main tasks that FileSwitch handles is keeping track of what filing systems are active, and switching between them as necessary. Much of the housekeeping part of the task is done for you; you just have to tell FileSwitch what to do.

Accessing hardware

When filing systems initialise, they tell FileSwitch their name, where to find their routines for controlling the hardware, and any special actions they are capable of.

Some calls you make to FileSwitch don't need to access hardware, and it deals with these itself. Other calls do need to access hardware; FileSwitch does the portion of the work that is independent of this, and calls a filing system module to access the hardware.

Finding out more...

For full details of FileSwitch, see the chapter entitled *FileSwitch* on page 3-9.

Adding filing systems

You can add filing system modules to the system, just as you can add any other module. They have to conform to the standards for modules, set out in the chapter entitled *Modules* on page 1-191; they also have to meet certain other standards to function correctly with FileSwitch as a filing system.

Because FileSwitch is already doing a lot of the work for you, you will have less work to do when you add a filing system than would otherwise be the case. Full details of how to add a filing system to FileSwitch are set out in the chapter entitled *Writing a filing system* on page 4-1.

Data format

FileSwitch does not lay down the format in which data must be laid out on a filing system, but it does specify what the user interface should look like.

FileCore

One of the filing system modules that RISC OS provides is FileCore. It takes the normal calls that FileSwitch sends to a filing system module, and converts them to a simpler set of calls to modules that control the hardware. So, like FileSwitch, it provides a common core of functions that are device independent, and it communicates with secondary *FileCore modules* that access the hardware. Unlike FileSwitch, it creates a fresh instantiation of itself for each module it supports.

Finding out more...

For full details of FileCore, see the chapter entitled *FileCore* on page 3-187.

Adding FileCore modules

You can, of course, add FileCore modules to the system. Using FileCore to build part of your filing system imposes a more rigid structure on it, as more of the filing system is predefined than if you do not use it. The filing system will appear very similar to ADFS or RamFS, both of which use FileCore. Of course, if you use FileCore to write a filing system it will be even less work for you, as even more of the system is already written.

For full details of using FileCore to implement a filing system, see the chapter entitled *Writing a FileCore module* on page 4-63.

DeviceFS

DeviceFS is another filing system module that takes the normal calls that FileSwitch sends to a filing system module, and converts them to a simpler set of calls to modules that control the hardware. It is intended for stream-based I/O. The secondary modules with which it communicates are known as *device drivers*: examples of these are the serial and parallel ports. Only a single instantiation of DeviceFS is needed.

DeviceFS is not included in RISC OS 2, and in RISC OS 3 will only support character devices. Support for block devices will be added to a future release.

Finding out more...

For full details of DeviceFS, see the chapter entitled *DeviceFS* on page 3-401.

Adding device drivers

As you'd expect, you can also add device drivers to RISC OS. For full details of using DeviceFS to implement a device driver, see the chapter entitled *Writing a device driver* on page 4-71.

Image filing systems

As well as standard filing systems, FileSwitch supports image filing systems. These provide facilities for RISC OS to handle media in foreign formats, and to support *image files* (or partitions) in those formats. They differ from standard filing systems in that they do not themselves access hardware; instead they rely on standard RISC OS filing systems to do so. DOSFS is an example of an image filing system, used to handle DOS format discs.

Image filing systems are not available in RISC OS 2.

There are three parts to an image filing system:

- The image handler manages files held within an image file, using FileSwitch and standard filing systems to do so.

  Image filing systems provide these facilities in a manner that is transparent to the end user; image files appear to be the same as any other file on the host filing system. The host filing system need not be aware of image filing systems to support this functionality.
- The identifier identifies the format of foreign media.

  To do so it communicates with a filing system using a special service call. The host filing system needs to be aware of image filing systems (ie must support the service call) to provide this functionality. Currently FileCore is the only standard filing system that does so.
- The formatter helps to format media, which is actually done by a standard filing system.

  Again, the host filing system needs to be aware of image filing systems to support this functionality. Currently ADFS is the only standard filing system that does so.

Finding out more...

For full details of DOSFS (a typical image filing system), see the chapter entitled *DeviceFS* on page 3-401.

Adding device drivers

You can add image filing systems to the system. For full details, see the chapter entitled *Writing a filing system* on page 4-1.

The Filer

The Filer module provides the facilities needed to display files and directories on the desktop, and to interact with them. It does so for all filing systems.

Finding out more...

For full details of the Filer, see the chapter entitled *The Filer* on page 3-465.

Filer_Action

Filer_Action performs file manipulation operations for the Filer without the desktop hanging whilst they are under way.

Finding out more...

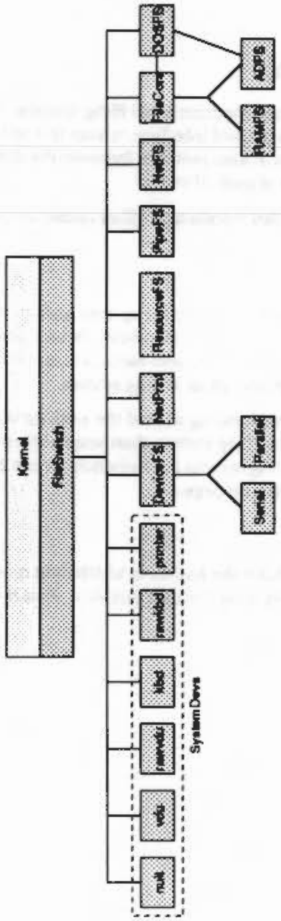
For full details of Filer_Action, see the chapter entitled *Filer_Action* on page 3-479.

Filers

Each filing system that provides an icon on the icon bar has a Filer module to do this, and to provide any associated services: for example, the ADFSFiler module. A Filer module can use service calls to interact with image filing systems, and add their formats to its menu of those it already supports.

Summary

The diagram below summarises the structure described above:



27 FileSwitch

Introduction and Overview

FileSwitch provides services common to all filing systems. It communicates with the filing systems using a defined interface; it uses this to tell the filing systems when they must do things. It also switches between the different filing systems, keeping track of the state of each of them.

See also the chapter entitled *Introduction to filing systems* on page 3-3.

Adding filing systems

You can add filing system modules to the system, just as you can add any other module. They have to conform to the standards for modules, set out in the chapter entitled *Modules* on page 1-191; they also have to meet certain other standards to function correctly with FileSwitch as a filing system.

Because FileSwitch is already doing a lot of the work for you, you will have less work to do when you add a filing system than would otherwise be the case. Full details of how to add a filing system to FileSwitch are set out in the chapter entitled *Writing a filing system* on page 4-1.

Data format

FileSwitch does not lay down the format in which data must be laid out on a filing system, but it does specify what the user interface should look like.

Technical Details

Terminology

The following terms are used in the rest of this chapter:

- a *file* is used to store data; it is distinct from a directory
- a *directory* is used to contain files
- an *object* may be either a file or a directory
- a *pathname* gives the location of an object, and may include a filing system name, a special field, a media name (eg a disc name), directory name(s), and the name of the object itself; each of these parts of a pathname is known as an *element* of the pathname
- a *full pathname* is a pathname that includes **all** relevant elements
- a *leafname* is the last element of a full pathname.

Filenames

Filename elements may be up to ten characters in length on FileCore-based filing systems (such as ADFS) and on NetFS. These characters may be digits or letters. FileSwitch makes no distinction between upper and lower case, although filing systems can do so. As a general rule, you should not use top-bit-set characters in filenames, although some filing systems (such as FileCore-based ones) support them. You may use other characters provided they do not have a special significance. Those that do are listed below:

.	Separates directory specifications, eg $.fred
:	Introduces a drive or disc specification, eg :0, :welcome. It also marks the end of a filing system name, eg adfs:
*	Acts as a 'wildcard' to match zero or more characters, eg prog*
#	Acts as a 'wildcard' to match any single character, eg $.ch##
$	is the name of the root directory of the disc
&	is the user root directory (URD)
@	is the currently-selected directory (CSD)
^	is the 'parent' directory
%	is the currently-selected library directory (CSL)
\	is the previously-selected directory (PSD – available on FileCore-based filing systems, and any others that choose to do so)

Directories

You may group files together into directories; this is particularly useful for grouping together all files of a particular type. Files in the directory currently selected may be accessed without reference to the directory name. Filenames must be unique within a given directory. Directories may contain other directories, leading to a hierarchical file structure.

The root directory, $, forms the top of the hierarchy of the media which contains the CSD. Through it you can access all files on that media. $ does not have a parent directory. Trying to access its parent will just access $. Note also that files have access permissions associated with them, which may restrict whether you can actually read or write to them.

Files in directories other than the current directory may be accessed either by making the desired directory the current directory, or by prefixing the filename by an appropriate directory specification. This is a sequence of directory names starting from one of the single-character directory names listed above, or from the current directory if none is given.

Each directory name is separated by a '.' character. For example:

`$.Documents.Memos`	File Memos in dir Documents in $
`BASIC.Games.Adventures`	File Adventures in dir Games in dir @.BASIC
`%.BCPL`	File BCPL in the current library

Filing systems

Files may also be accessed on filing systems other than the current one by prefixing the filename with a filing system specification. A filing system name may appear between '–' characters, or suffixed by a ':'. For example:

```
-net-$.SystemMesg
adfs:%.AAsm
```

You are strongly advised to use the latter, as the character '–' can also be used to introduce a parameter on a command line, or as part of a file name.

Special fields

Special fields are used to supply more information to the filing system than you can using standard path names; for example NetFS and NetPrint use them to specify server addresses or names. They are introduced by a # character; a variety of syntaxes are possible:

```
 net#MJHardy::discl.mike
    #MJHardy::discl.mike
-net#MJHardy-:discl.mike
   -#MJHardy-:discl.mike
```

The special fields here are all MJHardy, and give the name of the fileserver to use.

Special fields may use any character except for control characters, double quote '"', solidus '|' and space. If a special field contains a hyphen you may only use the first two syntaxes given above.

Special fields are passed to the filing system as null-terminated strings, with the '#' and trailing ':' or '–' stripped off. If no special field is specified in a pathname, the appropriate register in the FS routine is set to zero. See below for details of which calls may take special fields.

Current selections

FileSwitch keeps track of which filing system is currently selected. If you don't explicitly tell FileSwitch which filing system to use, it will use the current selection.

FileSwitch also keeps a record of each filing system's current selections, such as its CSD, CSL, PSD and URD. (Under RISC OS 2, this is independently recorded by individual filing systems, rather than by FileSwitch.)

File attributes

The top 24 bits of the file attributes are filing system dependent, eg NetFS returns the file server date of creation/modification of the object. The low byte has the following interpretation:

Bit	Meaning if set
0	Object has read access for you
1	Object has write access for you
2	Undefined
3	Object is locked against deletion
4	Object has read access for others
5	Object has write access for others
6	Undefined
7	Undefined

FileCore based filing systems (such as ADFS and RamFS) ignore the settings of bits 4 and 5, but you can still set these attributes independently of bits 0, 1 and 3. This is so that you can freely move files between ADFS, RamFS and NetFS without losing information on their public read and write access.

You should clear bits 2, 6 and 7 when you create file attributes for a file. They may be used in the future for expansion, so any routines that update the attributes must not alter these bits, and any routines that read the attributes must not assume these bits are clear.

Addresses / File types and date stamps

All files have (in addition to their name, length and attributes) two 32-bit fields describing them. These are set up when the file is created and have two possible meanings:

Load and execution addresses

In the case of a simple machine code program these are the load and execution addresses of the program:

Load address	&XXXLLLLL
Execution address	&GGGGGGGG

When a program is *Run, it is loaded at address &XXXLLLLL and execution commences at address &GGGGGGGG. Note that the execution address must be within the program or an error is given. That is:

XXXLLLLL ≤ GGGGGGGG < XXXLLLLL + Length of file

Also note that if the top twelve bits of the load address are all set (ie 'XXX' is FFF), then the file is assumed to be date-stamped. This is reasonable because such a load address is outside the addressing range of the ARM processor.

File types and date stamps

In this case the top 12 bits of the load address are all set. The remaining bits hold the date/time stamp indicating when the file was created or last modified, and the file type.

The date/time stamp is a five byte unsigned number which is the number of centi-seconds since 00:00:00 on 1st Jan 1900. The lower four bytes are stored in the execution address and the most-significant byte is stored in the least-significant byte of the load address.

The remaining 12 bits in the load address are used to store information about the file type. Hence the format of the two addresses is as follows:

Load address	`&FFFtttdd`
Execution address	`&dddddddd`

where 'd' is part of the date and 't' is part of the type.

The file types are split into three categories:

Value	**Meaning**
&E00 - &FFF	Reserved for Acorn use
&800 - &DFF	For allocation to software houses
&000 - &7FF	Free for the user

For a list of the file types currently defined, see the Table entitled *File types*.

If you type:

```
*Show File$Type_*
```

you will get a list of the file types your computer currently knows about.

Additional information

Some filing systems may store additional information with each file. This is dependent on the implementation of the filing system.

Load-time and run-time system variables

When a date stamped file of type ttt is *LOADed or *RUN, FileSwitch looks for the variables Alias$@LoadType_ttt or Alias$@RunType_ttt respectively. If a variable of string or macro type exists, then it is copied (after macro expansion), and the full pathname is used to find the file either on File$Path or Run$Path. Any parameters passed are also appended for *Run commands. The whole string is then passed to the operating system command line interpreter using XOS_CLI.

An example of LoadType

For example, suppose you type

```
*LOAD mySprites
```

when in the directory `adfs::HardDisc.$.Sprites`, and where the type of the file `mySprites` is &FF9. FileSwitch will issue:

```
*@LoadType_FF9 adfs::HardDisc.$.Sprites.mySprites
```

The value of the variable `Alias$@LoadType_FF9` is `SLoad %*0` by default, so the CLI converts the command via the alias mechanism to:

```
*SLoad adfs::HardDisc.$.Sprites.mySprites
```

- Note that RISC OS 2 does not expand file names to full pathnames and so would only issue:

  ```
  *@LoadType_FF9 mySprites
  ```

  which is then converted to:

  ```
  *SLoad mySprites
  ```

An example of RunType

Similarly, if you typed:

```
*Run BasicProg pl p2
```

where `BasicProg` is in the directory `adfs::HardDisc.$.Library`, and its file type is &FFB, FileSwitch would issue:

```
*@RunType_FFB adfs::HardDisc.$.Library.BasicProg pl p2
```

The variable `Alias$@LoadType_FFB` is `Basic -quit |"%0|" %*1` by default, so the CLI converts the command via the alias mechanism to:

```
*Basic -quit "adfs::HardDisc.$.Library.BasicProg" pl p2
```

Default settings

The filing system manager sets several of these variables up on initialisation, which you may override by setting new ones.

In the case of BASIC programs the settings are made as follows:

```
*Set Alias$@LoadType_FFB Basic -load |"%0|" %*1
*Set Alias$@RunType_FFB Basic -quit |"%0|" %*1
```

You can set up new aliases for any new types of file. For example, you could assign type &123 to files created by your own wordprocessor. The variables could then take be set up like this:

```
*Set Alias$@LoadType_123 WordProc %*0
*Set Alias$@RunType_123 WordProc %*0
```

File$Path and Run$Path

There are two more important variables used by FileSwitch. These control exactly where a file will be looked for, according to the operation being performed on it. The variables are:

File$Path	for read operations
Run$Path	for execute operations

The contents of each variable should expand to a list of prefixes, separated by commas.

When FileSwitch performs a read operation (eg load a file, open a file for input or update), then the prefixes in File$Path are used in the order in which they are listed. The first object that matches is used, whether it be a file or directory.

Similarly, when FileSwitch tries to execute a file (*RUN or *<filename> for example), the prefixes listed in Run$Path are used in order. If a matching object is a directory then it is ignored, unless it contains a !Run file. The first file, or directory.!Run file that matches is used.

Note that the search paths in these two variables are only ever used when the pathname passed to FileSwitch does not contain an explicit filing system reference. For example, *RUN file would use Run$Path, but *RUN adfs:file wouldn't.

Default values

By default, File$Path is set to the null string, and only the current directory is searched. Run$Path is set to ',%.', so the current directory is searched first, followed by the library.

Specifying filing system names

You can specify filing system names in the search paths. For example, if FileSwitch can't locate a file on the ADFS you could make it look on the fileserver using:

```
*SET File$Path ,%.,NET:LIB*.,NET:MODULES.
```

This would look for:

@.file, %.file, NET:LIB*.file and NET:MODULES.file.

Resulting filenames

If after expansion you get an illegal filename it is not searched for. So if you had set Run$Path like this:

```
*Set Run$Path adfs:,,net:,%.,!
```

then:

```
*Run $.mike
```

would search in turn for `adfs:$.mike`, `$.mike` and `net:$.mike`, but not for `%.$.mike` or `!$.mike` as they are illegal.

Path variables may expand to have leading and trailing spaces around elements of the path, so:

```
*Set Run$Path adfs:$.   ,  net:%.  ,  !
```

is perfectly legal. If you attempt to parse path variables, you must be aware of this and cope with it.

Avoiding using File$Path and Run$Path

Certain SWI calls also allow you to specify alternative path strings, and to perform the operation with no path look-up at all.

Using other path variables

You can set up other path variables and use them as pseudo filing systems. For example if you typed:

```
*Set Basic$Path adfs:$.basic.,net:$.basic.
```

you could then refer to the pseudo filing system as `Basic:` or (less preferable) as `-Basic-`.

These path variables work in the same way as File$Path and Run$Path.

System devices

In addition to the filing systems already mentioned, the module SystemDevices provides some device-oriented 'filing systems'. These can be used in redirection specifications in * Commands, and anywhere else where byte-oriented file operations are possible. The devices provided are:

kbd: & rawkbd:	the keyboard
null:	the 'null device'
printer:	the printer
vdu: & rawvdu:	the screen

Various other modules also provide system devices:

device:	the device filing system
netprint:	the network printer
parallel:	the parallel port

pipe:	the pipe filing system
resource:	the resource filing system
serial:	the serial port

For full details, see each chapter between *NetPrint* on page 3-367 and *System devices* on page 3-461.

Re-entrancy

FileSwitch can cope fully with recursive calls made to different streams – whether through the same or different entry points. For example:

- Handle 254 is an output file on a disc that's been removed.
- Handle 255 is a spool file.

1 You call OS_BPut to put a byte to 254; this fills the buffer and causes a flush to the filing system.

2 The filing system generates an UpCall to inform that the media is missing.

3 An UpCall handler prints a message asking the user to supply the media.

4 This goes through OS_BPut to 255, filling the buffer and causing a flush to the filing system.

If the filing systems are different then both calls to OS_BPut will work as expected. If they are the same, then it is dependent on the filing system whether it handles it. FileCore based systems, for example, do not.

Interrupt code

You must not call the filing systems from interrupt code; FileCore based systems in particular give an error if you try to do so.

FileSwitch and the kernel

Some of the * Commands and SWI calls listed below are provided by the kernel, and some by the FileSwitch module; they are grouped together here for ease of reference.

As well as the kernel and FileSwitch, the appropriate filing system module must be present for these commands to work, as it will carry out the low-level parts of each of the calls you make.

Further calls

In addition to the calls in this section, there are OS_Bytes to read/write the *Spool and *Exec file handles. See page 2-25 and page 2-384 respectively for details.

Service Calls

Service_StartUpFS
(Service Call &12)

Start up filing system

On entry

R1 = &12 (reason code)
R2 = filing system number

On exit

R1 preserved (never claim)
R2 preserved

Use

This is an old way to start up a filing system. It must not be claimed.

Service_FSRedeclare
(Service Call &40)

Filing system reinitialise

On entry

R1 = &40 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This service is called when the FileSwitch module has been reinitialised (due to a *RMReInit, for example). If you are in a filing system, you should make yourself known to FileSwitch by calling OS_FSControl 12 (see page 3-89). You must not claim this call.

Service_CloseFile
(Service Call &68)

Close an object, and any children of that object

On entry

R1 = &68 (reason code)
R2 = pointer to canonical filename (null terminated)
R3 = number of files closed as a result of this service call (initially 0)

On exit

R1, R2 preserved
R3 = number of files closed as a result of this service call (ie incremented appropriately)

Use

This call requests that the object specified by R0 be closed, and also any other objects that are beneath it in the directory tree. Your module need not close the file if this may potentially cause problems.

You must not claim this service call. Before passing this service on you must increment R3 by the number of files you closed.

For example, this call might be issued by the PC Emulator to cause a DOSFS partition file to be closed by FileSwitch. This doesn't cause problems as the partition would be spontaneously reopened if needed later.

This call is not issued by RISC OS 2.

SWI Calls

OS_Byte 127 (SWI &06)

Tells you whether the end of an open file has been reached

On entry

R0 = 127
R1 = file handle

On exit

R0 preserved
R1 indicates if end of file has been reached
R2 undefined

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call tells you whether the end of an open file has been reached, by checking whether the sequential pointer is equal to the file extent. It uses OS_Args 5 to do this; you should do so too in preference to using this call, which has been kept for compatibility only. See OS_Find (page 3-68) for details of opening a file. The values returned in R1 are as follows:

Value	Meaning
0	End of file has not been reached
Not 0	End of file has been reached

Related SWIs

OS_Args 5 (page 3-49), OS_Find (page 3-68)

Related vectors

ByteV

OS_Byte 139 (SWI &06)

Selects file options (as used by *Opt)

On entry

R0 = 139
R1 = option number (first *Opt argument)
R2 = option value (second *Opt argument)

On exit

R0 preserved
R1, R2 undefined

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call selects file options. It uses OS_FSControl 10 to do this. It is equivalent to *Opt, which is documented in detail on page 3-169.

Related SWIs

OS_FSControl 10 (page 3-87)

Related vectors

ByteV

OS_Byte 255 (SWI &06)

Reads the current auto-boot flag setting, or temporarily changes it

On entry

R0 = 255
R1 = 0 or new value
R2 = &FF or 0

On exit

R0 preserved
R1 = previous value
R2 corrupted

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call reads the current auto-boot flag setting, or changes it until the next hard reset or hard break. The auto-boot flag defaults to the value configured in the Boot/NoBoot option. If NoBoot is set, then, when the machine is reset, no auto-boot action will occur (ie no attempt will be made to access the boot file on the filing system). If Boot is the configured option, then the boot file will be accessed on reset. Either way, holding down the `Shift` key while releasing `Reset` will have the opposite effect to usual.

With this OS_Byte you can read the current state. On exit, if bit 3 of R1 is clear, then the action is Boot. If it is set, then the action is NoBoot.

The effect can be changed by writing to bit 3 of the flag, but this only lasts until the next hard reset or hard break. You should preserve the other bits of the flag.

Related SWIs

OS_FSControl 10 (page 3-87), OS_FSControl 15 (page 3-92)

Related vectors

ByteV

OS_File (SWI &08)

Acts on whole files, either loading a file into memory, saving a file from memory, or reading or writing a file's attributes

On entry

R0 = reason code
Other registers depend on reason code

On exit

R0 corrupted
Other registers depend on reason code

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_File acts on whole files, either loading a file into memory, saving a file from memory, or reading or writing a file's attributes. The call indirects through FileV.

The particular action of OS_File is given by the low byte of the reason code in R0 as follows:

R0	**Action**
0	Saves a block of memory as a file
1	Writes catalogue information for a named object
2	Writes load address only for a named object
3	Writes execution address only for a named object
4	Writes attributes only for a named object
5	Reads catalogue information for a named object, using File$Path
6	Deletes a named object

7	Creates an empty file
8	Creates a directory
9	Writes date/time stamp of a named file
10	Saves a block of memory as a file, and date/time stamps it
11	Creates an empty file, and time/date stamps it
12	Loads a named file, using specified path string
13	Reads catalogue information for a named object, using specified path string
14	Loads a named file, using specified path variable
15	Reads catalogue information for a named object, using specified path variable
16	Loads a named file, using no path
17	Reads catalogue information for a named object, using no path
18	Sets file type of a named file
19	Generates an error message
20	Reads catalogue information for a named object, using File$Path
21	Reads catalogue information for a named object, using specified path string
22	Reads catalogue information for a named object, using specified path variable
23	Reads catalogue information for a named object, using no path
24	Reads the natural block size of a file
255	Loads a named file, using File$Path

For details of each of these reason codes, see below.

FileSwitch will check the leafname for wildcard characters (* and #) before some of these operations. These are the ones which have a 'destructive' effect, eg deleting a file or saving a file (which might overwrite a file which already exists). If there are wildcards in the leafname, it returns an error without calling the filing system.

Non-destructive operations, such as loading a file and reading and writing attributes may have wildcards in the leafname. However, only the first file found will be accessed by the operation. The order of the search is filing system dependent, but is typically ASCII order.

Related SWIs

None

Related vectors

FileV

OS_File 0 and 10 (SWI &08)

Save a block of memory as a file

On entry

R0 = 0 or 10
R1 = pointer to non-wild-leaf filename
If R0 = 0
    R2 = load address
    R3 = execution address
If R0 = 10
    R2 = file type (bits 0 - 11)
R4 = start address in memory of data (inclusive)
R5 = end address in memory of data (exclusive)

On exit

Registers preserved

Use

These calls save a block of memory as a file, setting either its reload and execution addresses (R0 = 0), or its date/time stamp and file type (R0 = 10).

An error is returned if the object is locked against deletion, or is already open, or is a directory.

See also OS_File 7 and 11 (page 3-34); these create an empty file, ready to receive data.

OS_File 1, 2, 3, 4, 9, and 18 (SWI &08)

Write catalogue information for a named object

On entry

R0 = 1, 2, 3, 4, 9, or 18
R1 = pointer to (wildcarded) object name
If R0 = 1 or 2
  R2 = load address
Else If R0 = 18
  R2 = file type (bits 0 - 11)
If R0 = 1 or 3
  R3 = execution address
If R0 = 1 or 4
  R5 = object attributes

On exit

Registers preserved

Use

These calls write catalogue information for a named object to its catalogue entry, as shown below:

R0	**Information written**
1	Load address, execution address, object attributes
2	Load address
3	Execution address
4	Object attributes
9	Date/time stamp; file type is set to &FFD if not set already
18	File type, and date/time stamp if not set already

If the object name contains wildcards, only the first object matching the wildcard specification is altered.

FileCore based filing systems (such as ADFS) can write a directory's attributes; they do not generate an error if the object doesn't exist.

NetFS generates an error if you try to write a directory's attributes, or if the object doesn't exist.

OS_File 5, 13, 15 and 17 (SWI &08)

Read catalogue information for a named object

On entry

R0 = 5, 13, 15 or 17
R1 = pointer to (wildcarded) object name
If R0 = 13
  R4 = pointer to control-character terminated comma separated path string
If R0 = 15
  R4 = pointer to name of a path variable that contains a control-character terminated comma separated path string

On exit

R0 = object type
R1 preserved
R2 = load address
R3 = execution address
R4 = object length
R5 = object attributes
(R2 - R5 corrupted if object not found)

Use

The load address, execution address, length and object attributes from the named object's catalogue entry are read into registers R2, R3, R4 and R5. The value of R0 on entry determines what path is used to search for the object:

R0	**Path used**
5	FileSPath system variable
13	path string pointed to by R4
15	path variable, name of which is pointed to by R4
17	none

For a description of the path strings that are held in path variables, see the section entitled *File$Path and Run$Path* on page 3-16.

On exit, R0 contains the object type:

R0	Type
0	Not found
1	File found
2	Directory found
3	Image file found (ie both file and directory)

If the object name contains wildcards, only the first object matching the wildcard specification is read.

OS_File 6 (SWI &08)

Deletes a named object

On entry

R0 = 6
R1 = pointer to non-wildcarded object name

On exit

R0 = object type
R1 preserved
R2 = load address
R3 = execution address
R4 = object length
R5 = object attributes

Use

The information in the named object's catalogue entry is transferred to the registers and the object is then deleted from the structure. It is not an error if the object does not exist.

An error is generated if the object is locked against deletion, or if it is a directory which is not empty, or is already open.

NetFS behaves unusually in two ways:

- it always sets bit 3 of R5 on return (the object is 'locked')
- it returns the object's type as 2 (a directory) if it is successfully deleted.

OS_File 7 and 11 (SWI &08)

Creates an empty file

On entry

R0 = 7 or 11
R1 = pointer to non-wild-leaf file name
If R0 = 7
    R2 = reload address
    R3 = execution address
If R0 = 11
    R2 = file type (bits 0 - 11)
R4 = start address (normally set to 0)
R5 = end address (normally set to length of file)

On exit

Registers preserved

Use

Creates an empty file, setting either its reload and execution addresses (R0 = 7), or its date/time stamp and file type (R0 = 11).

Note: No data is transferred. The file does not necessarily contain zeros; the contents may be completely random. Some security-minded systems (such as NetFS/FileStore) will deliberately overwrite any existing data in the file.

An error is returned if the object is locked against deletion, or is already open, or is a directory.

See also OS_File 0 and 10 (page 3-29); these save a block of memory as a file.

OS_File 8 (SWI &08)

Creates a directory

On entry

R0 = 8
R1 = pointer to non-wild-leaf object name
R4 = number of entries (0 for default)

On exit

Registers preserved

Use

R4 indicates a minimum suggested number of entries that the created directory should contain without having to be extended. Zero is used to set the default number of entries.

Note: ADFS and other FileCore-based filing systems ignore the number of entries parameter, as this is predetermined by the disc format.

An error is returned if the object is a file which is locked against deletion. It is not an error if it refers to a directory that already exists, in which case the operation is ignored.

OS_File 12, 14, 16 and 255 (SWI &08)

Load a named file

On entry

R0 = 12, 14, 16 or 255
R1 = pointer to (wildcarded) object name
If bottom byte of R3 is zero
    R2 = address to load file at
R3 = 0 to load file at address given in R2, else bottom byte must be non-zero
If R0 = 12
    R4 = pointer to control-character terminated comma separated path string
If R0 = 14
    R4 = pointer to name of a path variable that contains a control-character terminated comma separated path string

On exit

R0 = object type (bit 0 always set, since object is a file)
R1 preserved
R2 = load address
R3 = execution address
R4 = file length
R5 = file attributes

Use

These calls load a named file into memory. The value of R0 on entry determines what path is used to search for the file:

R0	**Path used**
12	path string pointed to by R4
14	path variable, name of which is pointed to by R4
16	none
255	FileSPath system variable

For a description of the path strings that are held in path variables, see the section entitled *File$Path and Run$Path* on page 3-16.

If the object name contains wildcards, only the first object matching the wildcard specification is loaded.

You must set the bottom byte of R3 to zero for a file that is date-stamped, and supply a load address in R2.

An error is generated if the object does not exist, or is a directory, or does not have read access, or it is a date-stamped file for which a load address was not correctly specified.

OS_File 19 (SWI &08)

Generates an error message

On entry

R0 = 19
R1 = pointer to object name to report error for
R2 = object type

On exit

R0 = pointer to error block
V flag set

Use

This call is used to generate a friendlier error message for the specified object, such as:

```
"File 'xyz' not found"              r2 = 0
"'xyz' is a file"                   r2 = 1
"'xyz' is a directory"              r2 = 2
"Directory 'xyz' not found"         r2 = &100
```

An example of its use would be:

```
MOV         r0, #OSFile_ReadInfo
SWI         XOS_File
BVS         flurg
TEQ         R0, #object_file
MOVNE       r2, r0
MOVNE       r0, #OSFile_MakeError   ; return error if not a file
SWINE       XOS_File
BVS         flurg
```

OS_File 20, 21, 22 and 23 (SWI &08)

Read catalogue information for a named object

On entry

R0 = 20, 21, 22 or 23
R1 = pointer to (wildcarded) object name
If R0 = 21
    R4 = pointer to control-character terminated comma separated path string
If R0 = 22
    R4 = pointer to name of a path variable that contains a control-character terminated comma separated path string

On exit

R0 = object type
R1 preserved
R2 = load address, or high byte of date stamp (top three bytes of R2 are &000000)
R3 = execution address, or low word of date stamp
R4 = object length
R5 = object attributes
R6 = object filetype

Special	values:
-1	untyped (R2, R3 are load and execution address)
&1000	directory
&2000	application directory (directory whose name starts with a '!')

Use

This call reads the load and execution address (or date stamp), length, object attributes and filetype from the named object's catalogue entry into registers R2 - R6. The value of R0 on entry determines what path is used to search for the object:

R0	**Path used**
20	File$Path system variable
21	path string pointed to by R4
22	path variable, name of which is pointed to by R4
23	none

For a description of the path strings that are held in path variables, see the section entitled *File$Path and Run$Path* on page 3-16.

On exit, R0 contains the object type:

R0	Type
0	Not found
1	File found
2	Directory found
3	Image file found (ie both file and directory)

If the object name contains wildcards, only the first object matching the wildcard specification is read.

This call is not available in RISC OS 2.

OS_File 24 (SWI &08)

Reads the natural block size of a file

On entry

R0 = 24
R1 = pointer to file name

On exit

R2 = natural block size of the file in bytes

Use

This call reads the natural block size of a file in bytes, returning it in R2.

This call is not available in RISC OS 2.

OS_Args (SWI &09)

Reads or writes an open file's arguments, or returns the filing system type in use

On entry

R0 = reason code
R1 = file handle, or 0
R2 = attribute to write, or not used

On exit

R0 = filing system number, or preserved
R1 preserved
R2 = attribute that was read, or preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call indirects through ArgsV. The particular action of OS_Args is specified by R0 as follows:

Value	Action
0	Read pointer/FS number
1	Write pointer
2	Read extent
3	Write Extent
4	Read allocated size
5	Read EOF status
6	Reserve space
7	Get canonicalised name
8	Inform of changed image stamp
254	Read information on file handle
255	Ensure file/files

Related SWIs

None

Related vectors

ArgsV

OS_Args 0 (SWI &09)

Reads the temporary filing system number, or a file's sequential file pointer

On entry

R0 = 0
R1 = 0 or file handle

On exit

R0 = temporary filing system number (if R1 = 0 on entry), or preserved
R1 preserved
R2 = sequential file pointer (if R1 ≠ 0 on entry), or preserved

Use

This call reads the temporary filing system number (if R1 = 0 on entry), or a file's sequential file pointer (if R1 ≠ 0 on entry, in which case it is treated as a file handle).

This call indirects through ArgsV.

OS_Args 1 (SWI &09)

Writes an open file's sequential file pointer

On entry

R0 = 1
R1 = file handle
R2 = new sequential file pointer

On exit

R0 - R2 preserved

Use

This call writes an open file's sequential file pointer.

If the new sequential pointer is greater than the current extent, then more space is reserved for the file; this is filled with zeros. Writing the sequential pointer clears the file's EOF-*error-on-next-read* flag.

This call indirects through ArgsV.

OS_Args 2 (SWI &09)

Reads an open file's extent

On entry

R0 = 2
R1 = file handle

On exit

R0, R1 preserved
R2 = extent of file

Use

This call reads an open file's extent. It indirects through ArgsV.

OS_Args 3 (SWI &09)

Writes an open file's extent

On entry

R0 = 3
R1 = file handle
R2 = new extent

On exit

R0 - R2 preserved

Use

This call writes an open file's extent.

If the new extent is greater than the current extent, then more space is reserved for the file; this is filled with zeros. If the new extent is less than the current sequential pointer, then the sequential pointer is set back to the new extent. Writing the extent clears the file's EOF-*error-on-next-read* flag.

This call indirects through ArgsV.

OS_Args 4 (SWI &09)

Reads an open file's allocated size

On entry

R0 = 4
R1 = file handle

On exit

R0, R1 preserved
R2 = allocated size of file

Use

This call reads an open file's allocated size.

The size allocated to a file will be at least as big as the current file extent; in many cases it will be larger. This call determines how many more bytes can be written to the file before the filing system has to be called to extend it. This happens automatically.

This call indirects through ArgsV.

OS_Args 5 (SWI &09)

Reads an open file's end-of-file (EOF) status

On entry

R0 = 5
R1 = file handle

On exit

R0, R1 preserved
R2 = 0 if not EOF, else at EOF

Use

This call reads an open file's end-of-file (EOF) status.

If the sequential pointer is equal to the extent of the given file, then an end-of-file indication is given, with R2 set to non-zero on exit. Otherwise R2 is set to zero on exit.

This call indirects through ArgsV.

OS_Args 6 (SWI &09)

Ensures an open file's size

On entry

R0 = 6
R1 = file handle
R2 = size to ensure

On exit

R0, R1 preserved
R2 = bytes reserved for file

Use

This call ensures on open file's size.

The filing system is instructed to ensure that the size allocated for the given file is at least that requested. Note that this space thus allocated is not yet part of the file, so the extent is unaltered, and no data is written. R2 on exit indicates how much space the filing system actually allocated.

This call indirects through ArgsV.

OS_Args 7 (SWI &09)

Converts a file handle to a canonicalised name

On entry

R0 = 7
R1 = file handle
R2 = pointer to buffer to contain null terminated canonicalised name
R5 = size of buffer

On exit

R5 = number of spare bytes in the buffer **including** the null terminator, ie:

R5 ≥ 1 ⇒ there are (R5 – 1) completely unused bytes in the buffer; so R5 = 1 ⇒ there are 0 unused bytes in the buffer, and therefore the terminator just fitted

R5 ≤ 0 ⇒ there are (1 – R5) too many bytes to fit in the buffer, which has consequently not been filled in; so R5 = 0 ⇒ there is 1 byte too many – the terminator – to fit in the buffer

Use

This call takes a file handle and returns its canonicalised name. This may be used as a two-pass process:

Pass 1

On entry, set R1 to the file handle, and R2 and R5 (the pointer to, and size of, the buffer) to zero. On exit, R5 = –(length of canonicalised name)

Pass 2

Claim a buffer of the right size (1–R5, not just –R5, as a space is needed for the terminator). On entry, ensure that R1 still contains the file handle, that R2 is set to point to the buffer, and R5 contains the length of the buffer. On exit the buffer should be filled in, and R5 should be 0; but check to make sure.

This call indirects through ArgsV.

It is not available in RISC OS 2.

OS_Args 8 (SWI &09)

Used by an image filing system to inform of a change to an image stamp

On entry

R0 = 8
R1 = file handle
R2 = new image stamp

On exit

R0 - R2 preserved

Use

This call is made by an image filing system (eg DOSFS) when it has changed a disc's image stamp (a unique identification number). It does so to inform a handler of discs (eg FileCore) of the change, and to pass it the new image stamp. FileSwitch passes the information on to the disc handler, which typically just stores the new image stamp in that disc's record, so that the disc may be identified at a later time.

This call indirects through ArgsV.

It is not available in RISC OS 2.

OS_Args 254 (SWI &09)

Reads information on a file handle

On entry

R0 = 254 (&FE)
R1 = file handle (not necessarily allocated)

On exit

R0 = stream status word
R1 preserved
R2 = filing system information word

Use

This call returns information on a file handle, which need not necessarily be allocated.

The stream status word is returned in R0, the bits of which have the following meaning:

Bit	Meaning when set
14	Image file busy
13	Data lost on this stream
12	Stream is critical (see below)
11	Stream is unallocated (see below)
10	Stream is unbuffered
9	Already read at EOF (EOF-*error-on-next-read* flag)
8	Object written to
7	Have write access to object
6	Have read access to object
5	Object is a directory
4	Unbuffered stream directly supports GBPB
3	Stream is interactive

If bit 11 is set then no other bits in the stream status word have any significance, and the value of the filing system information word returned in R2 is undefined.

Any bits not in the above table are undefined, but you must not presume that they are zero.

Bit 12 shows when the stream is critical – in other words, when FileSwitch has made a call to a filing system to handle an open file, and the filing system has not yet returned. This is used to protect against accidental recursion **on the same file handle only**.

Bit 14 shows when an image file is busy; that is, when it is in the process of being opened by FileSwitch, but is not yet ready for use.

For a full definition of the filing system information word returned in R2, see the section entitled *Filing system information word* on page 4-2.

This call indirects through ArgsV.

OS_Args 255 (SWI &09)

Ensure a file, or all files on the temporary filing system

On entry

R0 = 255
R1 = file handle, or 0 to ensure all files on the temporary filing system

On exit

R0 - R2 preserved

Use

This call ensures that any buffered data has been written to either all files open on the temporary filing system (R1 = 0), or to the specified file (R1 ≠ 0, in which case it is treated as a file handle).

This call indirects through ArgsV.

OS_BGet (SWI &0A)

Reads a byte from an open file

On entry

R1 = file handle

On exit

R0 = byte read if C clear, undefined if C set
R1 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_BGet returns the byte at the current sequential file pointer position. The call indirects through BGetV.

If the sequential pointer is equal to the file extent (ie trying to read at end-of-file) then the EOF-error-on-next read flag is set, and the call returns with the carry flag set, R0 being undefined. If the EOF-*error-on-next-read* flag is set on entry, then an `End of file` error is given. Otherwise, the sequential file pointer is incremented and the call returns with the carry flag clear.

This mechanism allows one attempt to read past the end of the file before an error is generated. Note that various other calls (such as OS_BPut) clear the EOF-*error-on-next-read* flag.

An error is generated if the file handle is invalid; also if the file does not have read access.

Related SWIs

OS_BPut (page 3-58), OS_GBPB (page 3-59)

Related vectors

BGetV

OS_BPut (SWI &0B)

Writes a byte to an open file

On entry

R0 = byte to be written
R1 = file handle

On exit

Registers preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_BPut writes the byte given in R0 to the specified file at the current sequential file pointer. The sequential pointer is then incremented, and the EOF-***error-on-next-read*** flag is cleared. The call indirects through BPutV.

An error is generated if the file handle is invalid; also if the file is a directory, or is locked against deletion, or does not have write access.

Related SWIs

OS_BGet (page 3-56), OS_GBPB (page 3-59)

Related vectors

BPutV

OS_GBPB (SWI &0C)

Reads or writes a group of bytes from or to an open file

On entry

R0 = reason code
Other registers depend on reason code

On exit

R0 preserved
Other registers depend on reason code

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call reads or writes a group of bytes from or to an open file. It indirects through GBPBV.

The particular action of OS_GBPB is given by the reason code in R0 as follows:

R0	Action
1	Writes bytes to an open file using a specified file pointer
2	Writes bytes to an open file using the current file pointer
3	Reads bytes from an open file using a specified file pointer
4	Reads bytes from an open file using the current file pointer
5	Reads name and boot (*Opt 4) option of disc
6	Reads current directory name and privilege byte
7	Reads library directory name and privilege byte
8	Reads entries from the current directory
9	Reads entries from a specified directory

10	Reads entries and file information from a directory
11	Reads entries and full file information from a directory
12	Reads entries and file type information from a directory

Related SWIs

OS_BGet (page 3-56), OS_BPut (page 3-58)

Related vectors

GBPBV

OS_GBPB 1 and 2 (SWI &0C)

Write bytes to an open file

On entry

R0 = 1 or 2
R1 = file handle
R2 = start address of buffer in memory
R3 = number of bytes to write
If R0 = 1
    R4 = sequential file pointer to use for start of block

On exit

R0, R1 preserved
R2 = address of byte after the last one transferred from buffer
R3 = 0 (number of bytes not transferred)
R4 = initial file pointer + number of bytes transferred
C flag is cleared

Use

Data is transferred from memory to the file at either the specified file pointer (R0 = 1) or the current one (R0 = 2). If the specified pointer is beyond the end of the file, then the file is filled with zeros between the current file extent and the specified pointer before the bytes are transferred.

The memory pointer is incremented for each byte written, and the final value is returned in R2. R3 is decremented for each byte written, and is returned as zero. The sequential pointer of the file is incremented for each byte written, and the final value is returned in R4.

The EOF-*error-on-next-read* flag is cleared.

An error is generated if the file handle is invalid; also if the file is a directory, or is locked against deletion, or does not have write access.

OS_GBPB 3 and 4 (SWI &0C)

Read bytes from an open file

On entry

R0 = 3 or 4
R1 = file handle
R2 = start address of buffer in memory
R3 = number of bytes to read
If R0 = 3
R4 = sequential file pointer to use for start of block

On exit

R0, R1 preserved
R2 = address of byte after the last one transferred to buffer
R3 = number of bytes not transferred
R4 = initial file pointer + number of bytes transferred
C flag is clear if R3 = 0, else it is set

Use

Data is transferred from the given file to memory using either the specified file pointer (R0 = 3) or the current one (R0 = 4). If the specified pointer is greater than the current file extent then no bytes are read, and the sequential file pointer is not updated. Otherwise the sequential file pointer is set to the specified file location.

The memory pointer is incremented for each byte read, and the final value is returned in R2. R3 is decremented for each byte written. If it is zero on exit (all the bytes were read), the carry flag will be clear, otherwise it is set. The sequential pointer of the file is incremented for each byte read, and the final value is returned in R4.

The EOF-*error-on-next-read* flag is cleared.

An error is generated if the file handle is invalid; also if the file is a directory, or does not have read access.

OS_GBPB 5, 6 and 7 (SWI &0C)

Read information on a filing system

On entry

R0 = 5, 6 or 7
R2 = start address of buffer in memory

On exit

R0, R2 preserved
C flag corrupted

Use

These calls read information on the temporary filing system (normally the current one) to the buffer pointed to by R2. The value you pass in R0 determines the nature and format of the data, which is always byte-oriented:

- If R0 = 5, the call reads the name of the disc which contains the current directory, and its boot option. It is returned as:

  <name length byte><disc name><boot option byte>

  The boot option byte may contain values other than 0 - 3.
- If R0 = 6, the call reads the name of the currently selected directory, and privilege status in relation to that directory. It is returned as:

  <zero byte><name length byte><current directory name><privilege byte>

  The privilege byte is &00 if you have 'owner' status (ie can create and delete objects in the directory) or &FF if you have 'public' status (ie are prevented from creating and deleting objects in the directory). On ADFS and other FileCore-based filing systems you always have owner status.
- If R0 = 7, the call reads the name of the library directory, and privilege status in relation to that directory. It is returned as:

  <zero byte><name length byte><library directory name><privilege byte>

NetFS pads disc and directory names to the right with spaces; other filing systems do not. None of the names have terminators; so if the disc name were `Mike`, the name length byte would be 4.

OS_GBPB 8 (SWI &0C)

Reads entries from the current directory

On entry

R0 = 8
R2 = start address of data in memory
R3 = number of object names to read from directory
R4 = offset of first item to read in directory (0 for start)

On exit

R0, R2 preserved
R3 = number of objects not read
R4 = next offset in directory
C flag is clear if R3=0, else set

Use

This call reads entries from the current directory on the temporary filing system (normally the current one). You can also do this using OS_GBPB 9.

R3 contains the number of object names to read. R4 is the offset in the directory to start reading (ie if it is zero, the first item read will be the first file). Filenames are returned in the area of memory specified in R2. The format of the returned data is:

length of first object name	(one byte)
first object name in ASCII	(length as specified)
... repeated as specified by R3 ...	
length of last object name	(one byte)
last object name in ASCII	(length as specified)

If R3 is zero on exit, the carry flag will be cleared, otherwise it will be set. If R3 has the same value on exit as on entry then no more entries can be read and you must not call OS_GBPB 8 again.

On exit, R4 contains the value which should be used on the next call (to read more names), or -1 if there are no more names after the ones read by this call. There is no guarantee that the number of objects you asked for will be read. This is because of the external constraints some filing systems may impose. To ensure reading all the entries you want to, this call should be repeated until R4 = -1.

This call is only provided for compatibility with older programs.

OS_GBPB 9, 10, 11 and 12 (SWI &0C)

Read entries and file information from a specified directory

On entry

R0 = 9, 10, 11 or 12
R1 = pointer to directory name (control-character or null terminated)
R2 = pointer to buffer (word aligned if R0 = 10, 11 or 12)
R3 = number of object names to read from directory
R4 = offset of first item to read in directory (0 for start)
R5 = buffer length
R6 = pointer to (wildcarded) name to match

On exit

R0 - R2 preserved
R3 = number of objects read
R4 = offset of last item read (–1 if finished)
R5, R6 preserved
C flag is clear if R3=0, else set

Use

These calls read entries from a specified directory. If R0 = 10, 11 or 12 on entry the call also reads file information. If the directory name (which may contain wildcards) is null (ie R1 points to a zero byte), then the currently-selected directory is read.

The names which match the wildcard name pointed to by R6 are returned in the buffer. If R6 is zero or points to a null string then '*' is used, and all files will be matched. R3 indicates how many were read. R4 contains the value which should be used on the next call (to read more names), or –1 if there are no more names after the ones read by this call.

There is no guarantee that the number of objects you asked for will be read. This is because of the external constraints some filing systems may impose. To ensure reading all the entries you want to, this call should be repeated until R4 = –1.

If R0 = 9 on entry, the buffer is filled with a list of null-terminated strings consisting of the matched names.

If R0 = 10 on entry, the buffer is filled with records:

Offset	**Contents**
0	Load address
4	Execution address
8	Length
12	File attributes
16	Object type
20	Object name (null terminated)

Each record is word-aligned.

If R0 = 11 on entry, the buffer is filled with records:

Offset	**Contents**
0	Load address
4	Execution address
8	Length
12	File attributes
16	Object type
20	System internal name – for internal use only
24	Time/Date (cs since 1/1/1900) – 0 if not stamped
29	Object name (null terminated)

Each record is word-aligned.

If R0 = 12 on entry, the buffer is filled with records:

Offset	**Contents**
0	Load address, or high byte of date stamp (top three bytes are &000000)
4	Execution address, or low byte of date stamp
8	Length
12	File attributes
16	Object type
20	Object file type (as for OS_File 20-23)
24	Object name (null terminated)

Each record is word-aligned.

Note that even if R3 returns with 0, the buffer area may still have been overwritten: for instance, it may contain filenames which did not match the wildcard name pointed to by R6.

An error is generated if the directory could not be found.

OS_GBPB 12 is not available in RISC OS 2.

OS_Find (SWI &0D)

Opens and closes files

On entry

R0 = reason code
Other registers depend on reason code

On exit

Depends on reason code

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call opens and closes files.

If the low byte of R0 = 0 on entry, then you can either close a single file, or all files on the current filing system.

If the low byte of R0 ≠ 0 on entry then a file is opened for byte access. You can open files in the following ways:

- open an existing file with read access only
- create a new file with read/write access
- open an existing file with read/write access

When you open a file a unique file handle is returned to you. You need this for any calls you make to OS_Args (page 3-42), OS_BGet (page 3-56), OS_BPut (page 3-58) and OS_GBPB (page 3-59), and to eventually close the file using OS_Find 0.

For full details of the different reason codes, see the following pages.

Related SWIs

None

Related vectors

FindV

OS_Find 0
(SWI &0D)

Closes files

On entry

R0 = 0
R1 = file handle, or zero to close all files on current filing system

On exit

Registers preserved

Use

This call closes files. Any modified data held in RAM buffers is first written to the file(s).

If R1 = 0 on entry, then all files on the current filing system are closed. You should not use this facility within a program that runs in a multi-tasking environment such as the desktop, as it may close files being used by other programs.

Otherwise R1 must contain a file handle, that was returned by the earlier call of OS_Find that opened the file. Regardless of any errors returned, the file will always be closed on return from this call.

OS_Find 64 to 255
(SWI &0D)

Open files

On entry

R0 = reason code
R1 = pointer to object name
R2 = optional pointer to path string or path variable

On exit

R0 = file handle, or 0 if object doesn't exist
R1 and R2 preserved

Use

These calls open files. The way the file is opened is determined by bits 6 and 7 of R0:

R0	**Action**
&4X	open an existing file with read access only
&8X	create a new file with read/write access
&CX	open an existing file with read/write access

In fact there is no guarantee that you will get the access that you are seeking, and if you don't no error is returned at open time. The exact details depend on the filing system being used, but as a guide this is what any new filing system should do if the object is an existing file:

R0	**Action**
&4X	Return a handle if it has read access. Generate an error if it has not got read access.
&8X	Generate an error if it is locked, or has neither read nor write access. Otherwise return a handle, and open the file with its existing access, and with its extent set to zero.
&CX	Generate an error if it is locked and has no read access, or has neither read nor write access. Otherwise return a handle, and open the file with its existing access.

The access granted is cached with the stream, and so you cannot change the access permission on an open file.

Bits 4 and 5 of R0 currently have no effect, and should be cleared.

Bit 3 of R0 determines what happens if you try to open an existing file (ie R0 = &4X or &CX), but it doesn't exist:

Bit 3	Action
0	R0 is set to zero on exit
1	an error is generated

Bit 2 of R0 determines what happens if you try to open an existing file (ie R0 = &4X or &CX) but it is a directory:

Bit 2	Action
0	you can open the directory but cannot do any operations on it
1	an error is generated

If you are creating a new file (ie R0 = &8X) then an error is always generated if the object is a directory.

Bits 0 and 1 of R0 determine what path is used to search for the file:

Bit 0	Bit 1	Path used
0	0	File$Path system variable
0	1	path string pointed to by R2
1	0	path variable, name of which is pointed to by R2
1	1	none

For a description of the path strings that are held in path variables, see the section entitled *File$Path and Run$Path* on page 3-16.

In all cases the file pointer is set to zero. If you are creating a file, then the extent is also set to zero.

Note that you need the file handle returned in R0 for any calls you make to OS_Args (page 3-42), OS_BGet (page 3-56), OS_BPut (page 3-58) and OS_GBPB (page 3-59), and to eventually close the file using OS_Find 0 (page 3-70).

OS_FSControl (SWI &29)

Controls the filing system manager and filing systems

On entry

R0 = reason code
Other registers depend on reason code

On exit

R0 preserved
Other registers depend on reason code

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call controls the filing system manager and filing systems. It is indirected through FSControlV.

The particular action of OS_FSControl is given by the reason code in R0 as follows:

R0	Action
0	Set the current directory
1	Set the library directory
2	Inform of start of new application
3	Reserved for internal use
4	Run a file
5	Catalogue a directory
6	Examine the current directory
7	Catalogue the library directory
8	Examine the library directory

9	Examine objects
10	Set filing system options
11	Set the temporary filing system from a named prefix
12	Add a filing system
13	Check for the presence of a filing system
14	Filing system selection
15	Boot from a filing system
16	Filing system removal
17	Add a secondary module
18	Decode file type into text
19	Restore the current filing system
20	Read location of temporary filing system
21	Return a filing system file handle
22	Close all open files
23	Shutdown filing systems
24	Set the attributes of objects
25	Rename objects
26	Copy objects
27	Wipe objects
28	Count objects
29	Reserved for internal use
30	Reserved for internal use
31	Convert a string giving a file type to a number
32	Output a list of object names and information
33	Convert a file system number to a file system name
34	Reserved for future expansion
35	Add an image filing system
36	Image filing system removal
37	Convert a pathname to a canonicalised name
38	Convert file information to an object's file type
39	
40	
41	Return the defect list for an image
42	Map out a defect from an image
43	
44	
45	
46	Return an image file's used space map
47	Read the boot option of the disc or image file that holds a specified object
48	Write the boot option of the disc or image file that holds a specified object
49	Read the free space on the disc or image file that holds a specified object
50	Name the disc or image file that holds a specified object
51	Request that an image stamp be updated
52	Find the name and type of object that uses a particular offset
53	Set a specified directory to a given path without verification
54	Read the path of a specified directory

For details of each of these reason codes (except those reserved for internal use), see below.

Related SWIs

None

Related vectors

FSControlV

OS_FSControl 0 (SWI &29)

Set the current directory and (optionally) filing system

On entry

R0 = 0
R1 = pointer to (wildcarded) directory name

On exit

Registers preserved

Use

This call sets the current directory to the named one. If the name specifies a different filing system, it also selects that as the current filing system. If the name pointed to is null, the directory is set to the user root directory.

OS_FSControl 1 (SWI &29)

Set the library directory

On entry

R0 = 1
R1 = pointer to (wildcarded) directory name

On exit

Registers preserved

Use

This call sets the library directory on a filing system. If no filing system is specified, then the temporary filling system's library is set. If the name pointed to is null, the library directory is set to the filing system default (typically $.Library, if present).

Whenever a reference is made to the library on a specific filing system (eg net:%.Link), that filing system's library is used; if no filing system is specified (eg (%.Link), the temporary filing system's library is used.

If a filing system does not have a library directory set, then it searches in order the directories &.Library, $.Library and @. Under RISC OS 2, filing systems that are not FileCore based search % instead.

OS_FSControl 2 (SWI &29)

Informs RISC OS and the current application that a new application is starting

On entry

R0 = 2
R1 = pointer to command tail to set
R2 = *currently active object* pointer to write
R3 = pointer to command name to set

On exit

Registers preserved – may not return

Use

This call enables you to start up an application by hand, setting its environment string to a particular value; and allows FileSwitch and the kernel to free resources related to the current thread.

First of all, FileSwitch calls XOS_UpCall 256 (new application starting – see page 1-185), with R2 set to the *currently active object* pointer that may be written.

If the UpCall is claimed, this means that someone is refusing to let your new application be started, so the error 'Unable to start application' is **returned**.

FileSwitch then calls XOS_ServiceCall &2A (Service_NewApplication – see page 1-256), with R2 set to the *currently active object* pointer that may be written.

If the Service is claimed, this means that some module is refusing to let your new application be started; however an error cannot be returned as your calling task has just been killed, and FileSwitch would be returning to it. So FileSwitch **generates** the 'Unable to start application' error using OS_GenerateError (see page 1-41); this will be sent to the error handler of your calling task's parent, since your calling task will have restored its parent's handlers on receiving the UpCall 256.

Next, unless the Exit handler is below MemoryLimit, all handlers that are still set below MemoryLimit are reset to the default handlers (see OS_ReadDefaultHandler, page 1-313).

The *currently active object* pointer is then set to the value given and the environment string set up to be that desired. The current time is read into the environment time variable (see OS_GetEnv, page 1-291).

FileSwitch frees any temporary strings and transient blocks it has allocated and sets the temporary filing system to the current filing system.

If the call returns with V clear, all is set for your task to start up the application:

```
        MOV     R0, #FSControl_StartApplication
        LDR     R1, command_tail_ptr
        LDR     R2, execution_address
        BIC     R2, R2, #&FC000003      ; Address with no flags, USR mode
        LDR     R3, command_name_ptr
        SWI     XOS_FSControl
        BVS     return_error

; if in supervisor mode here, need to flatten SVC stack
;       LDR     R13, InitSVCStack

        TEQP    PC, #0                  ; USR mode, interrupts enabled
        MOV     R0, R0                  ; No op to avoid contention
        MOV     R12, #&80000000         ; Ensure called appl'n doesn't
        MOV     R13, #&80000000         ; assume a stack or workspace
        MOV     R14, PC                 ; Form return address
        MOV     PC, R2                  ; Enter appl'n: assumes CAO = exec

        SWI     OS_Exit                 ; In case it returns
```

OS_FSControl 4 (SWI &29)

Run a file

On entry

R0 = 4
R1 = pointer to (wildcarded) filename

On exit

Registers preserved

Use

This call runs a file. If a matching object is a directory then it is ignored, unless it contains a !Run file. The first file, or directory.!Run file that matches is used:

- A file with no type is run as an absolute application, provided its load address is not below &8000. The environment string is set to the command line, and the current time is read into the environment time variable – see OS_GetEnv (page 1-291).
- A file of type &FF8 (Absolute code) is run as an absolute application, loaded and entered at &8000. The environment string is set to the command line, and the current time is read into the environment time variable – see OS_GetEnv (page 1-291).
- A file of type &FFC (Transient code modules) is loaded into the RMA and executed there. The environment string is set to the command line, and the current time is read into the environment time variable – see OS_GetEnv (page 1-291). Transient calls are nestable; when a transient returns to the filing system manager the RMA space is freed. The RMA space is also freed (on the reset service or filing system manager death) if the transient execution stopped abnormally, eg an exception occurred or RESET was pressed. See the chapter entitled *Program Environment* on page 1-277 for details on writing transient utilities.
- Otherwise, the corresponding Alias$@RunType system variable is looked up to determine how the file is run.

This call may never return. If it is starting up a new application then the UpCall handler is notified, so any existing application has a chance to tidy up or to forbid the new application to start. It is only after this that the new application might be loaded.

The file is searched for using the variable Run$Path. If this does not exist, a path string of ',%.' is used (ie the current directory is searched first, followed by the library directory).

You cannot kill FileSwitch while it is threaded; so if you had an Obey file with the line:

```
RMKill FileSwitch
```

this will not work if the file is *Run, but would if you were to use *Obey.

An error is generated if the file is not matched, or does not have read access, or is a date/time stamped file without a corresponding Alias$@RunType variable.

OS_FSControl 5 (SWI &29)

Catalogue a directory

On entry

R0 = 5
R1 = pointer to (wildcarded) directory name

On exit

Registers preserved

Use

This call outputs a catalogue of the named subdirectory, relative to the current directory. If the name pointed to is null, the current directory is catalogued.

An error is returned if the directory does not exist, or the object is a file.

OS_FSControl 6 (SWI &29)

Examine a directory

On entry

R0 = 6
R1 = pointer to (wildcarded) directory name

On exit

Registers preserved

Use

This call outputs information on all the objects in the named subdirectory, relative to the current one. If the name pointed to is null, the current directory is examined.

An error is returned if the directory does not exist, or the object is a file.

OS_FSControl 7 (SWI &29)

Catalogue the library directory

On entry

R0 = 7
R1 = pointer to (wildcarded) directory name

On exit

Registers preserved

Use

This call outputs a catalogue of the named subdirectory, relative to the current library directory. If the name pointed to is null, the current library directory is catalogued.

An error is returned if the directory does not exist, or the object is a file.

OS_FSControl 8 (SWI &29)

Examine the library directory

On entry

R0 = 8
R1 = pointer to (wildcarded) directory name

On exit

Registers preserved

Use

This call outputs information on all the objects in the named subdirectory, relative to the current library directory. If the name pointed to is null, the current library directory is examined.

An error is returned if the directory does not exist, or the object is a file.

OS_FSControl 9 (SWI &29)

Examine objects

On entry

R0 = 9
R1 = pointer to (wildcarded) pathname

On exit

R0 preserved

Use

This call outputs information on all objects in the specified directory matching the wild-leaf-name given.

An error is returned if the pathname pointed to is null.

OS_FSControl 10 (SWI &29)

Sets filing system options

On entry

R0 = 10
R1 = option (0, 1 or 4)
R2 = parameter

On exit

Registers preserved

Use

This call sets filing system options on the temporary filing system (normally the current one). An option of 0 means reset all filing system options to their default values. See the *Opt command (page 3-169) for full details.

OS_FSControl 11 (SWI &29)

Set the temporary filing system from a named prefix

On entry

R0 = 11
R1 = pointer to string

On exit

R0 preserved
R1 = pointer to part of name past the filing system specifier if present
R2 = −1 ⇒ no filing system was specified (call has no effect)
R2 ≥ 0 ⇒ old filing system to be reselected
R3 = pointer to special field, or 0 if none present

Use

This call sets the temporary filing system from a filing system prefix at the start of the string, if one is present. It is used by OS_CLI (page 2-435) to set temporary filing systems for the duration of a command.

You can restore the temporary filing system to be the current one by calling OS_FSControl 19 (page 3-96).

OS_FSControl 12 (SWI &29)

Add a filing system

On entry

R0 = 12
R1 = module base address
R2 = offset of the filing system information block from the module base
R3 = private word pointer

On exit

Registers preserved

Use

This call informs FileSwitch that a module is a new filing system, to be added to the list of those it knows about. The module should make this call when it initialises.

R1 and R2 give the location of a filing system information block, which is used by FileSwitch to communicate with the filing system module. It contains both information about the filing system, and the location of entry points to the module's code.

The private word pointer passed in R3 is stored by FileSwitch. When it makes a call to the filing system module, the private word is passed in R12. Normally, this private word is the workspace pointer for the module.

For full information on writing a filing system module, see the chapter entitled *Writing a filing system* on page 4-1.

OS_FSControl 13 (SWI &29)

Check for the presence of a filing system

On entry

R0 = 13
R1 = filing system number, or pointer to filing system name
R2 = depends on R1

On exit

R0 preserved
R1 = filing system number
R2 = pointer to filing system control block or 0 if not found

Use

This call checks for the presence of a filing system.

If R1 < &100 then it is the filing system number; if, however, R1 ≥ &100 then it points to the filing system name. The filing system name match is case-insensitive. If R2 is 0, the filing system name is taken to be terminated with any control character or the characters: '#', ':' or '-'. If R2 is not 0, then the filing system name is terminated by any control character.

The filing system control block that is pointed to by R2 on exit is for the internal use of FileSwitch, and you should not use or alter it. You should only test the value of R2 for equality (or not) with zero.

An error is returned if the filing system name contains bad characters or is badly terminated.

OS_FSControl 14 (SWI &29)

Filing system selection

On entry

R0 = 14
R1 = filing system number, or pointer to filing system name

On exit

Registers preserved

Use

This call switches the current and temporary filing systems to the one specified by R1.

If R1 = 0 then no filing system is selected as the current or temporary one (the settings are cleared). If R1 is < &100 it is assumed to be a filing system number. Otherwise, it must be a pointer to a filing system name, terminated by a control-character or one of the characters '#', ':' or '-'. The filing system name match is case-insensitive.

This call is issued by filing system modules when they are selected by a * Command, such as *Net or *ADFS.

An error is returned if the filing system is specified by name and is not present.

OS_FSControl 15 (SWI &29)

Boot from a filing system

On entry

R0 = 15

On exit

R0 preserved

Use

This call boots off the currently selected filing system. It is called by the RISC OS kernel before entering the configured language module when the machine is reset using the Break key or reset switch, depending on the state of other keys, and on how the computer is configured.

This call may not return if boot runs an application.

For more details, see *Configure Boot (page 3-141), *Configure NoBoot (page 3-145), and the *Opt commands (page 3-169).

OS_FSControl 16 (SWI &29)

Filing system removal

On entry

R0 = 16
R1 = pointer to filing system name

On exit

Registers preserved

Use

This call removes the filing system from the list held by FileSwitch. It calls the filing system to close open files, flush buffers, and so on (except under RISC OS 2). You should use it in the finalise entry of a filing system module

Filing systems must be removed on any type of finalisation call, and added (including any relevant secondary modules) on any kind of initialisation. The reason for this is that FileSwitch keeps pointers into the filing system module code, which may be moved as a result of tidying the module area or other such operations.

R1 must be a pointer to a control-character terminated name – you cannot remove a filing system by file system number, and if you try to do so an error is returned.

Modules must not complain about errors in filing system removal. Otherwise, it would be impossible to reinitialise the module after reinitialising the filing system manager.

Under RISC OS 2, if the filing system is the temporary one then the temporary filing system is set to the current filing system. If the filing system is the current one, then both the current and temporary filing systems are set to 0 (none currently selected), and the old filing system number is stored. If it is added again before a new current filing system is selected then it will be reselected (see OS_FSControl 12 on page 3-89).

OS_FSControl 17 (SWI &29)

Add a secondary module

On entry

R0 = 17
R1 = pointer to filing system name
R2 = pointer to secondary system name
R3 = secondary module workspace pointer

On exit

Registers preserved

Use

This call is used to add secondary modules, so that extra filing system commands are recognised in addition to those provided by the primary filing system module. It is mainly used by FileCore (a primary module) to add its secondary modules such as ADFS.

OS_FSControl 18 (SWI &29)

Decode file type into text

On entry

R0 = 18
R2 = file type (bits 0 - 11)

On exit

R0 preserved
R2 = first four characters of textual file type
R3 = second four characters of textual file type

Use

This call issues Service_LookupFileType (see page 1-257). If the service is unclaimed, then it builds a default file type. For example if the file type is:

`Command`

the call packs the four bytes representing the characters:

`Comm` in R2

and the four bytes:

`and` in R3

The string is padded on the right with spaces to a maximum of 8.

This BASIC code converts the file type in `filetype%` to a string in `filetype$`, terminated by a carriage return:

```
DIM str% 8
SYS "OS_FSControl", 18,,filetype% TO ,,r2%,r3%
str%!0 = r2%
str%!4 = r3%
str%?8 = 13
filetype$ = $str%
```

OS_FSControl 31 (see page 3-108) does the opposite conversion – a textual file type to a file type number.

OS_FSControl 19 (SWI &29)

Restore the current filing system

On entry

R0 = 19

On exit

R0 preserved

Use

This call sets the temporary filing system back to the current filing system.

OS_CLI (see page 2-435) uses OS_FSControl 11 (see page 3-88)to set a temporary filing system before a command; it uses this call to restore the current filing system afterwards. This command is also called by the kernel before it calls the error handler.

OS_FSControl 20 (SWI &29)

Read location of temporary filing system

On entry

R0 = 20

On exit

R0 preserved
R1 = primary module base address of temporary filing system
R2 = pointer to private word of temporary filing system

Use

This call reads the location of the temporary filing system, and its private word. If no temporary filing system is set, then it reads the values for the current filing system instead. If there is no current filing system either, then R1 will be zero on exit, and R2 undefined.

OS_FSControl 21 (SWI &29)

Return a filing system file handle

On entry

R0 = 21
R1 = file handle

On exit

R0 preserved
R1 = filing system file handle
R2 = filing system information word

Use

This call takes a file handle used by FileSwitch, and returns the internal file handle used by the filing system which it belongs to. It also returns a filing system information word. For a full definition of this, see the section entitled *Filing system information word* on page 4-2.

The call returns a filing system file handle of 0 if the FileSwitch file handle is invalid.

You should only use this call to implement a filing system.

OS_FSControl 22 (SWI &29)

Close all open files

On entry

R0 = 22

On exit

R0 preserved

Use

This call closes all open files on all filing systems. It first ensures that any modified buffered data remaining in RAM (either in FileSwitch or in filing system buffers) is written to the appropriate files.

The call does not stop if an error is encountered, but goes on to close all open files. An error is returned if any individual close failed.

OS_FSControl 23 (SWI &29)

Shutdown filing systems

On entry

R0 = 23

On exit

R0 preserved

Use

This call closes all open files on all filing systems. It first ensures that any modified buffered data remaining in RAM (either in FileSwitch or in filing system buffers) is written to the appropriate files.

It informs all filing systems of the shutdown; most importantly this implies that it:

- logs off from all NetFS file servers
- unmounts all discs on FileCore-based filing systems
- parks the hard disc heads.

The call does not stop if an error is encountered, but goes on to close all open files. An error is returned if any individual close failed.

OS_FSControl 24 (SWI &29)

Set the attributes of objects

On entry

R0 = 24
R1 = pointer to (wildcarded) pathname
R2 = pointer to attribute string

On exit

Registers preserved

Use

This call gives the requested access to all objects in the specified directory whose names match the specified wild-leaf pattern.

If any of the characters in R2 are valid but inappropriate they are not faulted, but if they are invalid an error is returned. An error is also returned if the pathname pointed to is null, or if the pathname is not matched.

OS_FSControl 25 (SWI &29)

Rename object

On entry

R0 = 25
R1 = pointer to current pathname
R2 = pointer to desired pathname

On exit

Registers preserved

Use

This call renames an object. It is a 'simple' rename, implying that the source and destination are single objects which must reside on the same physical device, and hence on the same filing system.

An error is returned if the two objects are on different filing systems (checked by FileSwitch), or on different devices (checked by the filing system), or in different image files (checked by FileSwitch).

An error is also returned if the object is locked or is open, or if an object of the desired pathname exists, or if the directory referenced by the pathname does not already exist.

OS_FSControl 26 (SWI &29)

Copy objects

On entry

R0 = 26
R1 = pointer to source (wildcarded) pathname
R2 = pointer to destination (wildcarded) pathname
R3 = mask describing the action
R4 = optional inclusive start time (low 4 bytes)
R5 = optional inclusive start time (high byte, in bits 0 - 7)
R6 = optional inclusive end time (low 4 bytes)
R7 = optional inclusive end time (high byte, in bits 0 - 7)
R8 = optional pointer to extra information descriptor:
[R8] + 0 = information address
[R8] + 4 = information length

On exit

Registers preserved

Use

This call copies objects, optionally recursing.

The source leafname may be wildcarded. The only wildcarded destination leafname allowed is '*', which means to make the leafname the same as the source leafname.

The bits of the action mask have the following meaning when set:

Bit	Meaning when set
14	Reads destination object information and applies tests before loading any of the source object.
13	Uses extra buffer specified using R8.
12	Copies only if source is newer than destination.
11	Copies directory structure(s) recursively, but not files
10	Restamps datestamped objects – files are given the time at the start of this SWI, directories the time of their creation.
9	Doesn't copy over file attributes.

8 Allows printing during copy; printing is otherwise disabled. This option also disables any options that may cause characters to be written (bits 6, 4 and 3 are treated as cleared), and prevents FileSwitch from installing an UpCall handler to prompt for media changes.

7 Deletes the source after a successful copy (for renaming files across media).

6 Prompts you every time you **might** have to change media during the copy operation. In practise you are unlikely to need to use this option, as this SWI normally intercepts the UpCall vector and prompts you every time you *do* have to change media. (It only prompts if no earlier claimant of the vector has already tried to handle the UpCall.)

5 Uses application workspace as well as the relocatable module area.

4 Prints maximum information during copy.

3 Displays a prompt of the form 'Copy <object type> <source name> as <destination name> (Yes/No/Quiet/Abandon)?' for each object to be copied, and uses OS_Confirm to get a response. A separate confirm state is held for each level of recursion. *Yes* means to copy the object, *No* means not to copy the object, *Quiet* means to copy the object and to turn off confirmation at this level and subsequent ones (although if bit 1 is clear you will still be asked if you want to overwrite an existing file), and *Abandon* means not to copy the object and to return to the parent level. Escape abandons the entire copy without copying the object, and returns an error.

2 Copies only files with a time/date stamp falling between the start and end time/date specified in R4 - R7. (Unstamped files and directories will also be copied.) This check is made before any prompts or information is output.

1 Automatically unlocks, sets read and write permission, and overwrites an existing file. (If this bit is clear then the warning message 'File <destination name> already exists [and is locked]. Overwrite (Y/N) ? ' is given instead. If you answer *Yes* to this prompt then the file is similarly overwritten.)

0 Allows recursive copying down directories.

Buffers are considered for use in the following order, if they exist or their use is permitted:

1 user buffer

2 wimp free memory

3 relocatable module area (RMA)

4 application memory.

If either the Wimp free memory or the RMA buffers are used, they are freed between each object copied.

If application memory is used then FileSwitch starts itself up as the current application to claim application space. If on the start application service a module forbids the start-up, then the copy is aborted and an error is generated to the Error handler of the parent of the task that called OS_FSControl 26. The call does not return; it sets the environment time variable to the time read when the copy started and issues SWI OS_Exit, setting Sys$ReturnCode to 0.

OS_FSControl 27 (SWI &29)

Wipe objects

On entry

R0 = 27
R1 = pointer to wildcarded pathname to delete
R2 = not used
R3 = mask describing the action
R4 = optional start time (low 4 bytes)
R5 = optional start time (high byte, in bits 0 - 7)
R6 = optional end time (low 4 bytes)
R7 = optional end time (high byte, in bits 0 - 7)

On exit

Registers preserved

Use

This call is used to delete files. You can modify the effect of the call with the action mask in R3. Only bits 0 - 4 and 8 are relevant to this command. The function of these bits is as for OS_FSControl 26 (see page 3-103).

OS_FSControl 28 (SWI &29)

Count objects

On entry

R0 = 28
R1 = pointer to wildcarded pathname to count
R2 = not used
R3 = mask describing the action
R4 = optional start time (low 4 bytes)
R5 = optional start time (high byte, in bits 0 - 7)
R6 = optional end time (low 4 bytes)
R7 = optional end time (high byte, in bits 0 - 7)

On exit

R0, R1 preserved
R2 = total number of bytes of all files that were counted
R3 = number of files counted
R4 - R7 preserved

Use

This call returns information on the number and size of files. You can modify the effect of the call with the action mask in R3. Only bits 0, 2 - 4 and 8 are relevant to this command. The function of these bits is as for OS_FSControl 26 (see page 3-103).

Note that the command returns the amount of data that each object is comprised of, rather than the amount of disc space the data occupies. Since a file normally has space allocated to it that is not used for data, and directories are not counted, any estimates of free disc space should be used with caution.

OS_FSControl 31 (SWI &29)

Converts a string giving a file type to a number

On entry

R0 = 31
R1 = pointer to control-character terminated filetype string

On exit

R0, R1 preserved
R2 = filetype

Use

This call converts the string pointed to by R1 to a file type. Leading and trailing spaces are skipped. The string may either be a file type name (spaces within which will not be skipped):

```
Obey
Text
```

or represent a file type number (the default base of which is hexadecimal):

FEB	Hexadecimal version of Obey file type number
4_33333333	Base 4 version of Text file type number

OS_FSControl 18 (see page 3-95) does the opposite conversion – a file type number to a textual file type.

OS_FSControl 32 (SWI &29)

Outputs a list of object names and information

On entry

R0 = 32
R1 = pointer to wildcarded pathname

On exit

Registers preserved

Use

This call outputs a list of object names and information on them. The format is the same as for the *FileInfo command (see page 3-159).

OS_FSControl 33 (SWI &29)

Converts a filing system number to a filing system name

On entry

R0 = 33
R1 = filing system number
R2 = pointer to buffer
R3 = length of buffer

On exit

Registers preserved

Use

This call converts the filing system number passed in R1 to a filing system name. The name is stored in the buffer pointed to by R2, and is null-terminated. If FileSwitch does not know of the filing system number you pass it, a null string is returned.

OS_FSControl 35 (SWI &29)

Add an image filing system

On entry

R0 = 35
R1 = module base address
R2 = offset of the image filing system information block from the module base
R3 = private word pointer

On exit

Registers preserved

Use

This call informs FileSwitch that a module is a new image filing system, to be added to the list of those it knows about. The module should make this call when it initialises.

R1 and R2 give the location of an image filing system information block, which is used by FileSwitch to communicate with the image filing system module. It contains both information about the image filing system, and the location of entry points to the module's code.

The private word pointer passed in R3 is stored by FileSwitch. When it makes a call to the image filing system module, the private word is passed in R12. Normally, this private word is the workspace pointer for the module.

For full information on writing an image filing system module, see the chapter entitled *Writing a filing system* on page 4-1.

This call is not available in RISC OS 2.

OS_FSControl 36 (SWI &29)

Image filing system removal

On entry

R0 = 36
R1 = image filing system's file type

On exit

Registers preserved

Use

This call removes the image filing system from the list held by FileSwitch. It calls the image filing system to close open files, flush buffers, and so on. You should use it in the finalise entry of an image filing system module

Image filing systems must be removed on any type of finalisation call, and added on any kind of initialisation. The reason for this is that FileSwitch keeps pointers into the image filing system module code, which may be moved as a result of tidying the module area or other such operations.

R1 **must** be the image filing system's file type. You cannot remove a filing system by file system number, and if you try to do so an error is returned.

Modules must not complain about errors in filing system removal. Otherwise, it would be impossible to reinitialise the module after reinitialising the filing system manager.

This call is not available in RISC OS 2.

OS_FSControl 37 (SWI &29)

Converts a pathname to a canonicalised name

On entry

R0 = 37
R1 = pointer to pathname
R2 = pointer to buffer to contain null terminated canonicalised name
R5 = size of buffer

On exit

R5 = number of spare bytes in the buffer **including** the null terminator, ie:

$R5 \geq 1 \Rightarrow$ there are (R5 – 1) completely unused bytes in the buffer; so $R5 = 1 \Rightarrow$ there are 0 unused bytes in the buffer, and therefore the terminator just fitted

$R5 \leq 0 \Rightarrow$ there are (1 – R5) too many bytes to fit in the buffer, which has consequently not been filled in; so $R5 = 0 \Rightarrow$ there is 1 byte too many – the terminator – to fit in the buffer

Use

This call takes a pathname and returns its canonicalised name. However, case may differ, and wildcards may not be sorted out if the wildcarded object doesn't exist.

For example:

- 'a' may be resolved to 'adfs::HardDisc4.$.current.a' if the current directory is 'adfs::HardDisc4.$.current'.
- 'a*' may be resolved to the same thing if 'a' exists and is the first match for 'a*', but, if there is no match for 'a*', then 'adfs::HardDisc4.$.current.a*' will be returned.
- 'A' may be resolved to 'adfs::HardDisc4.$.current.A', which should be considered the same as 'adfs::HardDisc4.$.current.a'.

This may be used as a two-pass process:

Pass 1

On entry, set R1 to point to the pathname, and R2 and R5 (the pointer to, and size of, the buffer) to zero. On exit, R5 = –(length of canonicalised name)

Pass 2

Claim a buffer of the right size (1–R5, not just –R5, as a space is needed for the terminator). On entry, ensure that R1 still points to the pathname, that R2 is set to point to the buffer, and R5 contains the length of the buffer. On exit the buffer should be filled in, and R5 should be 0; but check to make sure.

This call is not available in RISC OS 2.

OS_FSControl 38 (SWI &29)

Converts file information to an object's file type

On entry

R0 = 38
R1 = pointer to the object's name
R2 = load address
R3 = execution address
R4 = object length
R5 = object attributes
R6 = object type (file/directory/image file)

On exit

R2 = object filetype

Special values:

–1	untyped (R2, R3 are load and execution address)
&1000	directory
&2000	application directory (directory whose name starts with a '!')

Use

This call converts file information, as returned by various calls – for example OS_File 5 – into the object's file type.

This call is not available in RISC OS 2.

OS_FSControl 39 (SWI &29)

Sets the User Root Directory

On entry

R0 = 39
R1 = pointer to User Root Directory

On exit

—

Use

This call sets the User Root Directory, which is shown as an '&' in pathnames.

This call is not available in RISC OS 2.

OS_FSControl 40 (SWI &29)

Exchanges current and previous directories

On entry

R0 = 40

On exit

—

Use

This call swaps the current and previously selected directories.

This call is not available in RISC OS 2.

OS_FSControl 41 (SWI &29)

Returns the defect list for an image

On entry

R0 = 41
R1 = pointer to name of image (null terminated)
R2 = pointer to buffer
R5 = buffer length

On exit

R0 - R5 preserved

Use

This call fills the given buffer with a defect list, which gives the byte offset to the start of each defect. The list is terminated by the value &20000000.

This call is not available in RISC OS 2.

OS_FSControl 42 (SWI &29)

Maps out a defect from an image

On entry

R0 = 42
R1 = pointer to name of image (null terminated)
R2 = byte offset to start of defect

On exit

R0 - R2 preserved

Use

This call maps out a defect from the given image.

This call is not available in RISC OS 2.

OS_FSControl 43 (SWI &29)

Unsets the current directory

On entry

R0 = 43

On exit

—

Use

This call unsets the current directory on the temporary filing system.

This call is not available in RISC OS 2.

OS_FSControl 44 (SWI &29)

Unsets the User Root Directory (URD).

On entry

R0 = 44

On exit

—

Use

This call unsets the User Root Directory on the temporary filing system.

This call is not available in RISC OS 2.

OS_FSControl 45 (SWI &29)

Unsets the library directory.

On entry

R0 = 45

On exit

—

Use

This call unsets the library directory on the temporary filing system.

This call is not available in RISC OS 2.

OS_FSControl 46 (SWI &29)

Returns an image file's used space map

On entry

R0 = 46
R1 = pointer to name of image (null terminated)
R2 = pointer to buffer
R5 = buffer length

On exit

R0 - R5 preserved

Use

This call returns an image file's used space map, filling the given buffer with 0 bits for unused blocks, and 1 bits for used blocks. The buffer will be filled to its limit, or to the file's limit, whichever is less. The 'perfect' size of the buffer can be calculated from the file's size and its block size. The correspondence of the buffer to the file is 1 bit to 1 block. The least significant bit (bit 0) in a byte comes before the most significant bit.

The used space is the total space excluding free space and defects.

For non-image files, the buffer will be filled with 1s.

This call is not available in RISC OS 2.

OS_FSControl 47 (SWI &29)

Reads the boot option of the disc or image file that holds a specified object

On entry

R0 = 47
R1 = pointer to name of object (null terminated)

On exit

R0, R1 preserved
R2 = boot option

Use

This call reads the boot option (ie the value *n* in *Opt 4,*n*) of the disc or image file that holds the specified object.

This call is not available in RISC OS 2.

OS_FSControl 48 (SWI &29)

Writes the boot option of the disc or image file that holds a specified object

On entry

R0 = 48
R1 = pointer to name of object (null terminated)
R2 = new boot option

On exit

R0 - R2 preserved

Use

This call writes the boot option (ie the value *n* in *Opt 4,*n*) of the disc or image file that holds the specified object.

This call is not available in RISC OS 2.

OS_FSControl 49 (SWI &29)

Reads the free space on the disc or image file that holds a specified object

On entry

R0 = 49
R1 = pointer to name of object (null terminated)

On exit

R0 = free space
R1 = largest creatable object
R2 = disc size

Use

This call reads the free space on the disc or image file that holds the specified object. It also returns the size of the largest creatable object, and the size of the disc.

This call is not available in RISC OS 2.

OS_FSControl 50 (SWI &29)

Names the disc or image file that holds a specified object

On entry

R0 = 50
R1 = pointer to name of object (null terminated)
R2 = new name of disc

On exit

R0 - R2 preserved

Use

This call names the disc or image file that holds the specified object.

This call is not available in RISC OS 2.

OS_FSControl 51 (SWI &29)

Used by a handler of discs to request that an image stamp be updated

On entry

R0 = 51
R1 = pointer to name of object (null terminated)
R2 = sub-reason code:

0	stamp when updated
1	stamp now

On exit

R0 - R2 preserved

Use

This call is made by a handler of discs (eg FileCore) to inform an image filing system (eg DOSFS) that it should update the disc's image stamp (a unique identification number), either when the disc is next updated (R2=0), or now (R2=1).

See the chapter entitled *Writing a filing system* on page 4-1 for more details.

This call is not available in RISC OS 2.

OS_FSControl 52 (SWI &29)

Finds the name and type of object that uses a particular offset within an image

On entry

R0 = 52
R1 = pointer to name of object (null terminated)
R2 = offset into disc or image
R3 = pointer to buffer to receive object name (if object found)
R4 = buffer length

On exit

R2 = kind of object found at offset:

0	no object found; offset is free/a defect/beyond end of image
1	no object found; offset is allocated, but not (free / a defect / beyond end of image) – eg the free space map
2	object found; cannot share the offset with other objects
3	object found; can share the offset with other objects

Use

This call finds the name and type of object that uses a particular offset within an image. On exit, if R2 = 2 or 3 then an object has been found, and the buffer will contain its full pathname; otherwise the buffer may be corrupted.

The image searched is the deepest image, eg if R1 pointed to:

```
$.pc.amiga.atari.a.b.c
```

where pc is a DOS disc image, `amiga` is an Amiga disc image, and `atari` an Atari disc image, then the image searched would be:

```
$.pc.amiga.atari
```

This call is not available in RISC OS 2.

OS_FSControl 53 (SWI &29)

Sets a specified directory to a given path without verification

On entry

R0 = 53
R1 = pointer to rest of path
R2 = directory to set
R3 = pointer to name of filing system (null-terminated)
R6 = pointer to special field (terminated by a null or '.'), or 0 if not present

On exit

Registers preserved

Use

This call explicitly tells FileSwitch to set the specified directory to the given path without it performing any form of verification on the path provided.

The 'rest of path' is a string giving the canonical path from the disc (if present) to the leaf which is the directory. It must not have wildcards in it, nor may it have any GSTransable bits to it. The string must be null-terminated. It must have a root directory of some sort (ie $, % or & must be present at the right place). For example:

- *Mount on ADFS may set the library to ':HardDisc4.$.Library'
- *Logon on NetFS may set the URD to ':FileServer.&'.

If R1 is 0 on entry then the relevant directory will be put into the unset state.

The value in R2 tells FileSwitch which directory to set:

Value	Directory	
0	@	(currently selected directory)
1	\	(previously selected directory)
2	&	(user root directory)
3	%	(library)

Other values are illegal.

The optional special field pointed to by R6 should consist of the textual part of the special field, after any # prefix that may have been present. It is terminated by a null byte or a '.'. It must not contain any wildcards or GSTransable bits.

This call is not available in RISC OS 2.

OS_FSControl 54 (SWI &29)

Reads the path of a specified directory

On entry

R0 = 54
R1 = pointer to buffer
R2 = directory to read
R3 = pointer to name of filing system (null-terminated)
R5 = size of buffer, or 0 to get required size of buffer

On exit

R1 = pointer to rest of path, or 0 if directory unset
R5 = value on entry, decremented by total size of data placed in buffer
R6 = pointer to special field (terminated by a null or '.'), or 0 if not present

Use

This call reads the path of a specified directory. It is the reverse of OS_FSControl 53 (see page 3-130). It is expected that this call will be used twice, the first time to get the buffer length (ie R5 = 0 on entry, on exit is decremented by required length), and the second time to fill the buffer. The buffer will have the special field and the rest of the path placed into it. The values in R1 and R6 are suitable for submission to OS_FSControl 53.

This call is not available in RISC OS 2.

* Commands

*Access

Controls who can run, read from, write to and delete specific files

Syntax

```
*Access object_spec [attributes]
```

Parameters

`object_spec`	a valid (wildcarded) pathname specifying a file or directory
`attributes`	The following attributes are allowed:
L(ock)	Lock object against deletion
R(ead)	Read permission
W(rite)	Write permission
/R, /W, /WR	Public read and write permission (on NetFS)

Use

*Access changes the attributes of all objects matching the wildcard specification. These attributes control whether you can run, read from, write to and delete a file.

NetFS uses separate attributes to control other people's access to your files: their 'public access'. By default, files are created without public read and write permission. If you want others on the network to be able to read files that you have created, make sure you have explicitly changed the access status to include public read. If you are willing to have other NetFS users work on your files (ie overwrite them), set the access status to public write permission. Other NetFS users cannot completely delete your files though, unless they have owner access.

The public attributes can be set within any FileCore-based filing system, except when using L-format; but they will be ignored unless the file is transferred to the NetFS. Other filing systems may work in the same way, or may generate an error if you try to use the public attributes.

Examples

```
*access myfile l
```

```
*access myfile wr/r
```

Related commands

*Ex, *FileInfo, *Info

*Append

Adds data to an existing file

Syntax

*Append *filename*

Parameters

filename a valid pathname specifying an existing file

Use

*Append opens an existing file so you can add more data to the end of the file. Each line of input is passed to OS_GSTrans before it is added to the file. Pressing Escape finishes the input.

Example

```
*type thisfile
this line is already in thisfile
*append thisfile
  1 some more text
Esc                                the Esc character terminates the file
*type thisfile
this line is already in thisfile
some more text
```

Related commands

*Build

*Back

Exchanges current and previous directories

Syntax

```
*Back
```

Use

*Back swaps the current and previously selected directories. The command is used for switching between two frequently used directories.

In RISC OS 2 this command is implemented by FileCore.

Related commands

*Dir

*Build

Opens a new file (or overwrites an existing one) and directs subsequent input to it

Syntax

```
*Build filename
```

Parameters

filename	a valid pathname specifying a file

Use

*Build opens a new file (or reopens an existing one with zero extent) and directs subsequent input to it. Each line of input is passed to OS_GSTrans before it is added to the file. Pressing Escape finishes the input.

Note that for compatibility with earlier systems the *Build command creates files with lines terminating in the carriage return character (ASCII &0D). The Edit application provides a simple way of changing this into a linefeed character, using the **CR↔LF** function from the Edit submenu.

Example

```
*Build testfile
  1 This is the first line of testfile
Esc                              the Esc character terminates the file
*Type testfile
This is the first line of testfile
```

Related commands

*Append

*Cat

Lists all the objects in a directory

Syntax

`*Cat [directory]`

Parameters

directory	a valid pathname specifying a directory

Use

*Cat (short for 'catalogue') lists all the objects in a directory, showing their access attributes and other information on the disc name, options set, etc. If no directory is specified, the contents of the current directory are shown. *Cat can be abbreviated to '*.' (a full stop), provided that you have not *Set the system variable Alias$. to a different value from its default.

Examples

`*.`	*catalogue of current directory*
`*cat net#59.254:`	*catalogue of current directory on NetFS file server 59.254*
`*.ram:$.Mike`	*catalogue of RAM filing system directory $.Mike*
`*Cat { > printer: }`	*catalogue of current directory redirected to printer*

Related commands

*Ex, *FileInfo, *Info, *LCat and *LEx

*CDir

Creates a directory

Syntax

`*CDir directory [size_in_entries]`

Parameters

directory	a valid pathname specifying a directory
size_in_entries	how many entries the directory should hold before it needs to be expanded (NetFS only)

Use

*CDir creates a directory with the specified pathname. On the NetFS, you can also give the size of the directory.

Examples

`*CDir fred`	*creates a directory called fred on the current filing system, as a daughter to the current directory*
`*CDir ram:fred`	*creates a directory called fred on the RAM filing system*

Related commands

*Cat

*Close

Closes all open files on the current filing system

Syntax

```
*Close
```

Parameters

None

Use

*Close closes all open files on the current filing system, and is useful when a program crashes, leaving files open.

If preceded by the filing system name, *Close can be used to close files on systems other than the current one. For example:

```
*adfs:Close
```

would close all files on ADFS, where NetFS is the current filing system.

You must not use this command within a program that runs in a multi-tasking environment such as the desktop, as it may close files being used by other programs.

Related commands

*Bye, *Shut, *Shutdown

*Configure Boot

Sets the configured boot action so that a power on, reset or Ctrl Break runs a boot file

Syntax

```
*Configure Boot
```

Parameters

None

Use

*Configure Boot sets the configured boot action so that a power on, reset or Ctrl Break runs a boot file, provided that the Shift key is not held down – if it is, then no boot takes place.

When a boot does take place, the file &.!Boot is looked for, and if found is loaded and run, as set by the *Opt 4 command. You might use a boot file to load a program automatically when the computer is switched on. For information on NetFS boot files, see your network manager.

You can use the *FX 255 command to override the configured boot action at any time; a typical use is to disable booting at the end of a boot file, so that the computer does not re-boot on a soft reset.

The Break key always operates as an Escape key after power on.

NoBoot is the default setting.

The change takes effect on the next power-on or hard reset.

Related commands

*Configure NoBoot, *FX 255, *Rename

*Configure DumpFormat

Sets the configured format used by the *Dump, *List and *Type commands

Syntax

```
*Configure DumpFormat n
```

Parameters

n A number in the range 0 to 15. The parameter is treated as a four-bit number.

The bottom two bits define how control characters are displayed, as follows:

Value	Meaning
0	GSTrans format is used (eg |A for ASCII 1)
1	Full stop '.' is used
2	<*d*> is used, where *d* is a decimal number
3	<&*h*> is used, where *h* is a hexadecimal number

If bit 2 is set, characters which have their top bit set are treated as printable characters; otherwise they are treated as control characters. n=5, for example, causes ASCII character 247 to be printed as + (Latin fonts only).

If bit 3 is set, characters which have their top bit set are ANDed with &7F before being processed so the top bit is no longer set; otherwise they are left as they are.

Use

*Configure DumpFormat sets the configured format used by the *Dump, *List and *Type commands, and the vdu: output device. The default value is 4 (GSTrans format, and characters with the top bit set are printed using all 8 bits).

*Dump ignores the setting of the bottom two bits of the parameter, and always prints control characters as full stops.

The change takes effect immediately.

Example

```
*Configure DumpFormat 2
```

Related commands

*Dump, *List

*Configure FileSystem

Sets the configured filing system to be used at power on or hard reset

Syntax

```
*Configure FileSystem fs_name|fs_number
```

Parameters

fs_name	a filing system name (ADFS, Net or Ram)
fs_number	a filing system number (8 for ADFS, 5 for NetFS)

Use

*Configure FileSystem sets the configured filing system to be used at power on or hard reset. The filing system is selected just before any boot action is taken, and a banner is displayed showing its name. (The banner is also shown on a soft reset.)

To specify the filing system by name (rather than by number), FileSwitch must have that name registered at the time you use this command. This is because FileSwitch needs to convert the name to the filing system number that is actually stored.

If the configured filing system is not found on a reset then FileSwitch will return an error on every subsequent command that tries to use the currently selected filing system, until a current filing system is successfully selected.

Example

```
*Configure FileSystem Net
```

*Configure NoBoot

Sets the configured boot action so that a Shift power on, Shift reset or Shift Break runs a boot file

Syntax

```
*Configure NoBoot
```

Parameters

None

Use

*Configure NoBoot sets the configured boot action so that any kind of reset doesn't run a boot file – except if the Shift key is held down, when a boot takes place.

When a boot does take place, the file &.!Boot is looked for, and if found is loaded and run, as set by the *Opt 4 command. You might use a boot file to load a program automatically when the computer is switched on. For information on NetFS boot files, see your network manager.

You can use the *FX 255 command to override the configured boot action at any time; a typical use is to disable booting at the end of a boot file, so that the computer does not re-boot on a soft reset.

The Break key always operates as an Escape key after power on.

This is the default setting.

The change takes effect on the next power-on or hard reset.

Related commands

*Configure Boot, *FX 255, *Rename

*Configure Truncate

Sets the configured value for whether or not filenames are truncated when too long

Syntax

```
*Configure Truncate On|Off
```

Parameters

`On`	long filenames are truncated
`Off`	long filenames are not truncated

Use

*Configure Truncate sets the configured value for whether or not filenames are truncated when too long for a filing system to handle.

If you are writing a filing system that is unable to handle filenames over a certain length, you should examine the bit of CMOS that this command alters (see the section entitled *Non-volatile memory* (CMOS RAM) on page 1-346). If filename truncation is off, you should generate a 'Bad name' error if you are passed too long a filename; otherwise, you should truncate all filenames.

This command is not available in RISC OS 2.

Example

```
*Configure Truncate On
```

Related commands

None

*Copy

Copies files and directories

Syntax

```
*Copy source_spec destination_spec [[~]options]
```

Parameters

source_spec	a valid (wildcarded) pathname specifying a file or directory
destination_spec	a valid (wildcarded, but see below for restrictions) pathname specifying a file or directory
options	upper- or lower-case letters, optionally separated by spaces

A set of default options is read from the system variable CopySOptions, which is set by the system as shown below. You can change these default preferences using the *Set command. You are recommended to type:

```
*Set Copy$Options <Copy$Options> extra_options
```

so you can see what the original options were before you added your extra ones. The default options are overruled by any given to the command.

To ensure an option is ON, include it in the list of options; to ensure it is OFF, immediately precede the option by a '~' (eg ~C~r to turn off the C and R options).

- **A**(ccess) Force destination access to same as source.
  Default ON.

  Important when you are copying files from ADFS to NetFS, for example, because it maintains the access rights on the files copied. You should set this option to be OFF when you are updating a common release on the network, to maintain the correct access rights on it.

- **C**(onfirm) Prompt for confirmation of each copy.
  Default ON.

  Useful as a check when you have used a wildcard, to ensure that you are copying the files you want. Possible replies to the prompt for each file are **Y**(es) (to copy the file or structure and then proceed to the next item), **N**(o) (to go on to the next item without making a copy), **Q**(uiet) (to copy the item and all subsequent items without further prompting), **A**(bandon) (to stop copying at the current level – see the R option), or **Esc** (to stop immediately).

- **D**(elete) Delete the source object after copy.
  Default OFF.
  This is useful for moving a file from one disc or other storage unit to another. The source object is copied; if the copy is successful, the source object is then deleted. If you want to move files within the same disc, the *Rename command is quicker, as it does not have to copy the files.
- **F**(orce) Force overwriting of existing objects.
  Default OFF.
  Performs the copy, regardless of whether the destination files exist, or what their access rights are. The files can be overwritten even if they are locked or have no write permission.
- **L**(ook) Look at destination before loading source file.
  Default OFF.
  Files are normally copied by reading a large amount of data into memory before attempting to save it as a destination file. The L option checks the destination medium for accessibility before reading in the data. Thus L often saves time in copying, except for copies on the same disc.
- **N**(ewer) Copy only if source is more recent than destination.
  Default OFF.
  This is useful during backups to prevent copying the same files each time, or for ensuring that you are copying the latest version of a file.
- **P**(rompt) Prompt for the disc to be changed as needed in copy.
  Default OFF.
  This is provided for compatibility with older filing systems and you should not need to use it. Most RISC OS filing systems will automatically prompt you to change media.
- **Q**(uick) Use application workspace as a buffer
  Default OFF.
  The Q option uses the application workspace, so overwrites whatever is there. It should not be used if an application is active.
  Copying in the Desktop can use the Wimp's free memory, and so you should not need to use this option. It's quicker not to use this option when you are copying from hard disc to floppy, as these operations are interleaved so well. However, in other circumstances this option can speed up the copying operation considerably.
- **R**(ecurse) Copy subdirectories and contents.
  Default OFF.
  This is useful when copying several levels of directory, since it avoids the need to copy each of the directories one by one.

- **S**(tamp) Restamp date-stamped files after copying.
  Default OFF.
  Useful for recording when the particular copy was made.
- (s)**T**(ructure) Copy only the directory structure.
  Default OFF.
  Copies the directory structure but not the files. By using this option as a first stage in copying a directory tree, access to the files is faster when they are subsequently copied.
- **V**(erbose) Print information on each object copied.
  Default ON.
  This gives full textual commentary on the copy operation.

Use

Copy makes a copy between directories of any object(s) that match the given wildcard specification. Objects may be files or directories. The only wildcard that can be used in the destination is a '' as the leafname (see example below).

A * wildcard can be used in the *Copy command. For example:

```
*Copy data* Dir2.*
```

will copy all the files in the current directory with names beginning data to Dir2. The leafname of the destination must either be a specific filename, or the character '*' (as in the above example) in which case the destination will have the same leafname as the source.

Note that it is dangerous to copy a directory into one of its subsidiary directories. This results in an infinite loop, which only comes to an end when the disc is full or Esc is pressed.

If the Copy$Options variable is unset then Copy behaves as if the variable were set to its default value.

Examples

```
*Copy fromfile tofile rfq~c~v
*Copy :fred.data* :jim.*
```

Copies all files beginning 'data' from the disc called 'fred' to the disc called 'jim'.

Related commands

*Access, *Delete, *Rename, *Wipe, and the system variable Copy$Options.

*Count

Adds up the size of data held in file objects, and the number of objects

Syntax

```
*Count object_spec [[~]options]
```

Parameters

object_spec	a valid (wildcarded) pathname specifying a file or directory
options	upper- or lower-case letters, optionally separated by spaces

A set of default options is read from the system variable Count$Options, which is set by the system as shown below. You can change these default preferences using the *Set command. You are recommended to type:

```
*Set Count$Options <Count$Options> extra_options
```

so you can see what the original options were before you added your extra ones. The default options are overruled by any given to the command.

To ensure an option is ON, include it in the list of options; to ensure it is OFF, precede the option by a '~' (eg: ~C~r to turn off the C and R options).

- C(onfirm) Default OFF. Prompt for confirmation of each count.
- R(ecurse) Default ON. Count subdirectories and contents.
- V(erbose) Default OFF. Print information on each file counted.

  This gives information on each file counted, rather than just printing the subtotal counted in directories.

Use

*Count adds up the size of data held in one or more objects that match the given wildcard specification.

Note that the command returns the amount of data that each object is comprised of, rather than the amount of disc space the data occupies. Since a file normally has space allocated to it that is not used for data, and directories are not counted, any estimates of free disc space should be used with caution.

If the Count$Options variable is unset then Count behaves as if the variable were set to its default value.

Example

```
*Count $ r~cv
```
Counts all files on disc, giving full information on each file object

Related commands

*Ex, *FileInfo, *Info, and the system variable Count$Options

*Create

Reserves space for a new file

Syntax

```
*Create filename [length [exec_addr [load_addr>]]]
```

Parameters

filename	a valid pathname specifying a file
length	the number of bytes to reserve (default 0)
exec_addr	the address to be jumped to after loading, if a program
load_addr	the address at which the file is loaded into RAM when *Loaded (default 0)

Use

*Create reserves space for a new file, usually a data file. No data is transferred to the file. You may assign load and execution addresses if you wish. The units of length, load and execution addresses are in hexadecimal by default.

If both load and execution addresses are omitted, the file is created with type FFD (Data) and is date and time stamped.

Examples

`*Create mydata 1000 0 8000`	*Creates a file &1000 bytes long, which will be loaded into address &8000*
`*Create newfile 10_4096`	*Creates a file &1000 bytes long which is date and time stamped*
`*Create bigfile &10000`	

Related commands

*Load, *Save

*Delete

Erases a single file or empty directory

Syntax

```
*Delete object_spec
```

Parameters

object_spec	a valid (wildcarded, but see below for restrictions) pathname specifying a file or an empty directory

Use

*Delete erases the single named file or empty directory. If the object does not exist, or is a directory containing files, an error message is given. Wildcards may be used in all the components of the pathname except the last one.

Examples

`*Delete $.dir*.myfile`	*Uses wildcards*
`*Delete myfile`	*Deletes myfile from the current directory*

Related commands

*Remove, *Wipe

*Dir

Selects a directory

Syntax

`*Dir [directory]`

Parameters

directory	a valid pathname specifying a directory

Use

*Dir selects a directory as the currently selected directory (CSD) on a filing system. You may set the CSD separately on each filing system, and on each server of a multi-server filing system such as NetFS. If no directory is specified, the user root directory (URD) is selected.

Examples

`*Dir`	*sets the* CSD *to the* URD
`*Dir mydir`	*sets the* CSD *to mydir*

A CSD may be set for each filing system, for instance, within NetFS, the command:

`*Dir ADFS:...`	*sets the current filing system to* ADFS *and selects the* CSD *there; it does not affect the* CSD *in* NetFS

whereas:

`*ADFS:Dir...`	*sets the* CSD *on* ADFS *only;* NetFS *remains the current filing system*

Related commands

*Back, *CDir

*Dump

Displays the contents of a file, in hexadecimal and ASCII codes

Syntax

`*Dump filename [file_offset [start_addr]]`

Parameters

filename	a valid pathname specifying a file
file_offset	offset, in hexadecimal, from the beginning of the file from which to dump the data
start_addr	display as if the file were in memory starting at this address (in hexadecimal) – defaults to the file's load address

Use

*Dump displays the contents of a file as a hexadecimal and (on the righthand side of the screen) as an ASCII interpretation. An address is given on the lefthand side of:

start_address + *current offset in file*

You can set the format used to display the ASCII interpretation using *Configure DumpFormat. This gives you control over:

- whether the top bit of a byte is stripped first
- how bytes are displayed if their top bits are set.

If a file is time/date stamped, it is treated as having a load address of zero.

Example

`*Dump myprog 0 8000`	*Dumps the file myprog, starting from the beginning of the file (offset is* 0*) but numbering the dump from* &8000, *as if the file were loaded at that address*

Related commands

*Configure DumpFormat, *List, *Type

*EnumDir

Creates a file of object leafnames

Syntax

```
*EnumDir directory output_file [pattern]
```

Parameters

`directory`	a valid pathname specifying a directory
`output_file`	a valid pathname specifying a file
`pattern`	a wildcard specification for matching against

Use

*EnumDir creates a file of object leafnames from a directory that match the supplied wildcard specification.

The default pattern is *, which will match any file within a directory. The current directory can be specified by @.

Examples

`*EnumDir $.dir myfile data*`	*Creates a file myfile, containing a list of all files beginning data contained in directory $.dir*
`*EnumDir @ listall *_doc`	*Creates a file listall, containing a list of all files in the current directory whose names end in _doc*

Related commands

*Cat, *LCat

*Ex

Lists file information within a directory

Syntax

```
*Ex [directory]
```

Parameters

`directory`	a valid pathname specifying a directory

Use

*Ex lists all the objects in a directory together with their corresponding file information. The default is the current directory.

Most filing systems also display an informative header giving the directory's name and other useful information.

Example

```
*Ex mail

Mail                    Owner
DS                      Option 0 (Off)
Dir. MHardy             Lib. ArthurLib

Current    WR     Text         15:54:37 04-Jan-1989   60  bytes
LogFile    WR     Text         15:54:37 04-Jan-1989  314  bytes
```

Related commands

*FileInfo, *Info

*Exec

Executes a command file

Syntax

```
*Exec [filename]
```

Parameters

filename	a valid pathname specifying a file

Use

*Exec instructs the operating system to take its input from the specified file, carrying out the instructions it holds. This command is mainly used for executing a list of operating system commands contained in a command file. The file, once open, takes priority over the keyboard or serial input streams.

If no parameter is given, the current exec file is closed.

Example

`*Exec !Boot`	*uses the file !Boot as though its contents have been typed in from the keyboard*

Related commands

*Obey

Related SWIs

OS_Byte 198 (page 2-384)

Related vectors

None

*FileInfo

Gives full file information about specified objects

Syntax

```
*FileInfo object_spec
```

Parameters

object_spec	a valid (wildcarded) pathname specifying one or more files and/or directories

Use

*FileInfo gives file information for the specified object(s); this consists of the filename, the access permission, the filetype and datestamp or the load and execution addresses (in hexadecimal), and the length of the file in hexadecimal.

Under RISC OS 2, the information given varies between filing systems, as does the matching (or not) of wildcards.

Example

```
*FileInfo current
Current    WR/    Text       15:54:37.40 04-Jan-1989 000007F
```

Related commands

*Ex, *Info

*Info

Gives file information about specified objects

Syntax

```
*Info object_spec
```

Parameters

`object_spec` a valid (wildcarded) pathname specifying one or more files and/or directories

Use

*Info gives file information for the specified object(s); this consists of the filename, the access permission, the filetype and datestamp or the load and execution addresses (in hexadecimal), and the length of the file.

If the file is dated, the date and time are displayed using the current Sys$DateFormat. If it is not dated, the load and exec addresses are displayed in hexadecimal.

Example

```
*Info myfile

myfile    WR  Text      15:54:37 04-Jan-1989   60  bytes
```

Related commands

*Ex, *FileInfo

*LCat

Displays objects in a library

Syntax

```
*LCat [directory]
```

Parameters

`directory` a valid pathname specifying a subdirectory of the current library

Use

*LCat lists all the objects in the named library subdirectory. If no subdirectory is named, the objects in the current library are listed. *LCat is equivalent to *Cat %.

Related commands

*Cat, *LEx

*LEx

Displays file information for a library

Syntax

```
*LEx [directory]
```

Parameters

directory	a valid pathname specifying a subdirectory of the current library

Use

*LEx lists all the objects in the named library subdirectory together with their file information. If no subdirectory is named, the objects in the current library are listed. *LEx is the equivalent of `*Ex %`.

Related commands

*Ex, *LCat

*Lib

Selects a directory as a library

Syntax

```
*Lib [directory]
```

Parameters

directory	a valid pathname specifying a directory

Use

*Lib selects a directory as the current library on a filing system. You can independently set libraries on each filing system.

If no other directory is named, the action taken will depend on which filing system is currently open: In ADFS the default is $.Library; under NetFS there is no default.

Example

`*Lib $.mylib`	*Sets the directory $.mylib to be the current library*

Related commands

*Configure Lib, *NoLib

*List

Displays file contents with line numbers

Syntax

```
*List [-File] filename [-TabExpand]
```

Parameters

`-File`	may optionally precede *filename*; it has no effect
filename	a valid pathname specifying a file
`-TabExpand`	causes Tab characters (ASCII 9) to be expanded to 8 spaces

Use

*List displays the contents of a file using the configured DumpFormat. Each line is numbered.

Example

```
*List -file myfile -tabexpand
```

Related commands

*Configure DumpFormat, *Dump, *Print, *Type

*Load

Loads the named file (usually a program file)

Syntax

```
*Load filename [load_addr]
```

Parameters

filename	a valid pathname specifying a file
load_addr	load address (in hexadecimal by default); this overrides the file's load address or any load address in the Alias$@LoadType variable associated with this file

Use

*Load loads the named file at a load address specified in hexadecimal.

The filename which is supplied with the *Load command is searched for in the directories listed in the system variable File$Path. By default, File$Path is set to ''. This means that only the current directory is searched.

If no address is specified, the file's type (BASIC, Text etc) is looked for:

- If the file has no file type, it is loaded at its own load address.
- If the file does have a file type, the corresponding Alias$@LoadType variable is looked up to determine how the file is to be loaded. A BASIC file has a file type of &FFB, so the variable Alias$@LoadType_FFB is looked up, and so on. You are unlikely to need to change the default values of these variables.

Example

```
*Load myfile 9000
```

Related commands

*Create, *Save

*NoDir

Unsets the current directory

Syntax

```
*NoDir
```

Use

*NoDir unsets the current directory.

In RISC OS 2 this command is implemented by FileCore.

Related commands

*Dir, *NoLib, *NoURD

*NoLib

Unsets the library directory.

Syntax

```
*NoLib
```

Use

*NoLib unsets the library directory.

In RISC OS 2 this command is implemented by FileCore.

Related commands

*Lib, *NoDir, *NoURD

*NoURD

Unsets the User Root Directory (URD).

Syntax

```
*NoURD
```

Use

*NoURD unsets the User Root Directory (URD).

In RISC OS 2 this command is implemented by FileCore.

Related commands

*NoDir, *NoLib, *URD

*Opt 1

*Opt 1 controls filing system messages

Syntax

```
*Opt 1 [[,]n]
```

Parameters

n	0 to 3

Use

*Opt 1 sets the filing system message level (for operations involving loading, saving or creating a file) for the current filing system:

`*Opt 1,0`	No filing system messages
`*Opt 1,1`	Filename printed
`*Opt 1,2`	Filename, hexadecimal addresses and length printed
`*Opt 1,3`	Filename, and either datestamp and length, or hexadecimal load and exec addresses printed

*Opt 1 must be set separately for each filing system.

*Opt 4

*Opt 4 sets the filing system boot action

Syntax

```
*Opt 4 [[,]n]
```

Parameters

n	0 to 3

Use

*Opt 4 sets the boot action for the current filing system. On filing systems with several media (eg ADFS using several discs) the boot action is only set for the medium (disc) containing the currently selected directory.

`*Opt 4,0`	No boot action
`*Opt 4,1`	*Load boot file
`*Opt 4,2`	*Run boot file
`*Opt 4,3`	*Exec boot file

The boot file is named !Boot, except when using NetFS, when it is called !ArmBoot.

Note that a *Exec boot file will override the configured language setting. If you want such a boot file, and want to enter the desktop after executing it, the file should end with the command *Desktop; similarly for other languages.

Example

`*Opt 4,2`	*sets the boot action to *Run for the current filing system*

Related commands

*Configure Boot, *Configure NoBoot

*Print

Displays raw text on the screen

Syntax

```
*Print filename
```

Parameters

`filename`	a valid pathname specifying a file

Use

*Print displays the contents of a file by sending each byte – whether it is a printable character or not – to the VDU. Unless the file is a simple text file, some unwanted screen effects may occur, since control characters are not filtered out.

Example

```
*Print myfile
```

Related commands

*Dump, *List, *Type

*Remove

Deletes a file

Syntax

```
*Remove filename
```

Parameters

filename	a valid pathname specifying a file

Use

*Remove deletes a named file. Its action is like that of *Delete, except that no error message is generated if the file does not exist. This allows a program to remove a file without having to trap that error.

Related commands

*Delete, *Wipe

*Rename

Changes the name of an object

Syntax

```
*Rename object new_name
```

Parameters

object	a valid pathname specifying a file or directory
new_name	a valid pathname specifying a file or directory

Use

*Rename changes the name of an object, within the same storage unit. It can also be used for moving files from one directory to another, or moving directories within the directory tree.

Locked objects cannot be renamed (unlock them first by using the *Access command with the Lock option clear).

To move objects between discs or filing systems, use the *Copy command with the **D**(elete) option set.

Examples

```
*Rename fred jim
*Rename $.data.fred $.newdata.fred
```

Moves fred into directory newdata

Related commands

*Access, *Copy

*Run

Loads and executes a file

Syntax

```
*Run filename [parameters]
```

Parameters

filename	a valid pathname specifying a file
parameters	a Command Line tail (see the chapter entitled *Program Environment* on page 1-277 for further details)

Use

*Run loads and executes a file, optionally passing a list of parameters to it. The given pathname is searched for in the directories listed in the system variable Run$Path. If a matching object is a directory then it is ignored, unless it contains a !Run file.

The first file, or *directory*.!Run file that matches is used:

- If the file has no file type, it is loaded at its own load address, and execution commences at its execution address.
- If the file has type &FF8 (Absolute code) it is loaded and run at &8000
- Otherwise the corresponding Alias$@RunType variable is looked up to determine how the file is to be run. A BASIC file has a file type of &FFB, so the variable Alias$@RunType_FFB is looked up, and so on. You are unlikely to need to change the default values of these variables.

By default, Run$Path is set to ',%.'. This means that the current directory is searched first, followed by the library. This default order is also used if Run$Path is not set.

Examples

`*Run my_prog`	
`*Run my_prog my_data`	*my_data is passed as a parameter to the program my_prog. The program can then use this filename to look up the data it needs.*

Related commands

*SetType

*Save

Copies an area of memory to a file

Syntax

```
*Save filename start_addr end_addr [exec_addr [load_addr]]
```

or

```
*Save filename start_addr + length [exec_addr [load_addr]]
```

Parameters

filename	a valid pathname specifying a file
start_addr	the address of the first byte to be saved
end_addr	the address of the byte after the last one to be saved
length	number of bytes to save
exec_addr	execution address (default is *start_addr*)
load_addr	load address (default is *start_addr*)

Use

*Save copies the given area of memory to the named file. Start_addr is the address of the first byte to be saved; end_addr is the address of the byte after the last one to be saved. Length is the number of bytes to be saved; exec_addr is the execution address to be stored with the file (it defaults to start_addr). Load_addr is the reload address (which also defaults to start_addr).

The length and addresses are in hexadecimal by default.

Examples

```
*Save myprog 8000 + 3000

*Save myprog 8000 B000 9300 9000
```

Related commands

*Load, *SetType

*SetType

Sets the file type of a file

Syntax

```
*SetType filename file_type
```

Parameters

filename	a valid pathname specifying a file
file_type	a number (in hexadecimal by default) or text description of the file type to be set. The command `*Show File$Type*` displays a list of valid file types.

Use

*SetType sets the file type of the named file. If the file does not have a date stamp, then it is stamped with the current time and date. Examples of file types are Palette, Font, Sprite and BASIC: for a list, see *Table* C: *File types* on page 6-487, or type `*Show File$Type*` at the command line.

Textual names take preference over numbers, so the sequence:

```
*Set File$Type_123 DFE
*SetType filename DFE
```

will set the type of `filename` to &123, not &DFE – the string DFE is treated in the second command as a file type name, not number. To avoid such ambiguities we recommend you always precede a file type number by an indication of its base.

Example

Build a small file containing a one-line command, set it to be a command type (&FFE), and run it from the Command Line; finally, view it from the desktop:

```
*Build x                      the file is given the name x
  1 *Echo Hello World         the line number is supplied by *Build
Esc                           the Escape character terminates the file
*SetType x Command            *SetType x &FFE is an alternative
*Run x                        the text is echoed on the screen
```

The file has been ascribed the 'command file' type, and can be run by double-clicking on the file icon.

*Shut

Closes all open files

Syntax

```
*Shut
```

Parameters

None

Use

*Shut closes all open files on all filing systems. The command may be useful to programmers to ensure that all files are closed if a program crashes without closing files.

You must not use this command within a program running in a multi-tasking environment such as the desktop, as it may close files being used by other programs.

Related commands

*Bye, *Close, *ShutDown

*ShutDown

Closes files, logs off file servers and parks hard disc heads

Syntax

```
*ShutDown
```

Parameters

None

Use

*ShutDown closes all open files on all filing systems, and also logs off all NetFS file servers and parks hard disc heads in a safe state for switching off the computer.

You must not use this command within a program running in a multi-tasking environment such as the desktop, as it may close files being used by other programs.

Related commands

*Bye, *Close, *Shut

*Spool

Sends everything appearing on the screen to the named file

Syntax

```
*Spool [filename]
```

Parameters

filename	a valid pathname specifying a file

Use

*Spool opens the specified file for output; if a file of that name already exists, it is overwritten. All subsequent characters sent to the VDU drivers will be copied to the file, using OS_BPut. (If OS_BPut returns an error, the spool file is closed – thereby restoring the spool handle location – and the error is then returned from OS_WriteC.)

This copying continues until either a *Spool or a *SpoolOn command (with or without a file name) is issued, which then terminates the spooling.

If the pathname is omitted, the current spool file, if any, is closed, and characters are no longer sent to it. If the pathname is given, then the existing spool file is closed and the new one opened.

You can temporarily disable the spool file, without closing it, using OS_Byte 3.

Example

```
*Spool %.Showdump
```

```
*Spool
```

Related commands

*SpoolOn

Related SWIs

OS_Byte 3 (page 2-18), OS_Byte 199 (page 2-25), OS_File (page 3-27), OS_BPut (page 3-58)

Related vectors

BPutV, ByteV

*SpoolOn

Adds everything appearing on the screen to the end of an existing file

Syntax

```
*SpoolOn [filename]
```

Parameters

filename a valid pathname specifying an existing file

Use

*SpoolOn is similar to *Spool, except that it adds data to the end of an existing file. All subsequent characters sent to the VDU drivers will be copied to the end of the file, using OS_BPut. (If OS_BPut returns an error, the spool file is closed – thereby restoring the spool handle location – and the error is then returned from OS_WriteC.)

This copying continues until either a *SpoolOn or a *Spool command (with or without a filename) is issued, which then terminates the spooling.

If the filename is omitted, the current spool file, if any, is closed, and characters are no longer sent to it. If the filename is given, then the existing spool file is closed and the new one opened.

You can temporarily disable the spool file, without closing it, using OS_Byte 3.

Example

```
*SpoolOn %.Showlist
```

```
*SpoolOn
```

Related commands

*Spool

Related SWIs

OS_Byte 3 (page 2-18), OS_Byte 199 (page 2-25), OS_File (page 3-27), OS_BPut (page 3-58)

Related vectors

ByteV, BPutV

*Stamp

Date stamps a file

Syntax

```
*Stamp filename
```

Parameters

filename a valid pathname specifying a file

Use

*Stamp sets the date stamp on a file to the current time and date. If the file has not previously been date stamped, it is also given file type Data (&FFD).

Example

```
*Stamp myfile
```

Related commands

*Info, *SetType

*Type

Displays the contents of a file.

Syntax

```
*Type [-File] filename [-TabExpand]
```

Parameters

`-File`	may optionally precede *filename*; it has no effect
filename	a valid pathname specifying a file
`-TabExpand`	causes Tab characters (ASCII 9) to be expanded to 8 spaces

Use

*Type displays the contents of the file in the configured DumpFormat. Control F might be displayed as '|F', for instance.

Example

```
*Type -File myfile -TabExpand
```

Related commands

*Configure DumpFormat, *Dump, *List, *Print

*Up

Moves the current directory up the directory structure

Syntax

```
*Up [levels]
```

Parameters

levels	a positive number in the range 1 to 128 (in decimal by default)

Use

*Up moves the current directory up the directory structure by the specified number of levels. If no number is given, the directory is moved up one level. *Up is equivalent to *Dir ^.

Note that while NetFS supports this command, some fileservers do not, so you may get a *File 'up' not found* error.

Example

`*Up 3`	*This is equivalent to* *Dir ^.^.^, *but note that the parent of* $ *is* $, *so you cannot go any further up the directory tree than this.*

Related commands

*Dir

*URD

Sets the User Root Directory

Syntax

```
*URD [directory]
```

Parameters

directory any valid pathname specifying a directory

Use

*URD sets the User Root Directory. This is shown as an '&' in pathnames.

If no directory is specified, the URD is set to the root directory.

In RISC OS 2 this command is implemented by FileCore.

Example

```
*URD adfs::0.$.MyDir
```

Related commands

*NoURD

*Wipe

Deletes one or more objects.

Syntax

```
*Wipe object_spec [[~]options]
```

Parameters

object_spec a valid pathname specifying one or more files and/or directories

options upper- or lower-case letters, optionally separated by spaces

A set of default options is read from the system variable Wipe$Options, which is set by the system as shown below. You can change these default preferences using the *Set command. You are recommended to type:

```
*Set Wipe$Options <Wipe$Options> extra_options
```

so you can see what the original options were before you added your extra ones. The default options are overruled by any given to the command.

To ensure an option is ON, include it in the list of options; to ensure it is OFF, precede the option by a '~' (eg: ~C~r to turn off the C and R options).

- **C**(onfirm) Default ON. Prompt for confirmation of each deletion.
- **F**(orce) Default OFF. Force deletion of locked objects.
- **R**(ecurse) Default OFF. Delete subdirectories and contents.
- **V**(erbose) Default ON. Print information on each object deleted.

Use

*Wipe deletes one or more objects that match the given wildcard specification.

If the Wipe$Options variable is unset then Wipe behaves as if the variable were set to its default value.

Example

`*Wipe Games.* ~R`	*Deletes all files in the directory Games (but not any of its subdirectories).*

28 FileCore

Introduction

FileCore is a filing system that does not itself access any hardware. Instead it provides a core of services to implement a filing system similar to ADFS in operation. Secondary modules are used to actually access the hardware.

ADFS and RamFS are both examples of such secondary modules, which provide a complete filing system when combined with FileSwitch and FileCore.

The main use you may have for FileCore is to use it as the basis for writing a new ADFS-like filing system. Because it already provides many of the functions, it will considerably reduce the work you have to do.

See also the chapter entitled *Introduction to filing systems* on page 3-3.

Overview

FileCore is a filing system module. It provides all the entry points for FileSwitch that any other filing system does. Unlike them, it does not control hardware; instead it issues calls to secondary modules that do so.

Similarities with FileSwitch

This concept of a parent module providing many of the functions, and a secondary module accessing the hardware, is very similar to the way that FileSwitch works. There are further similarities:

- there is a SWI, FileCore_Create, which modules use to register themselves with FileCore as part of the filing system
- this SWI is passed a pointer to a table giving information about the hardware, and entry points to low-level routines in the module
- FileCore communicates with the module using these entry points.

When you register a module with FileCore it creates a fresh instantiation of itself, and returns a pointer to its workspace. Your module then uses this to identify itself on future calls to FileCore.

Adding a module to FileCore

When you add a new module to FileCore, there is comparatively little work to be done. It needs:

- low-level routines to access the hardware
- a * Command that can be used to select the filing system
- any additional * Commands you feel necessary – typically very few
- a SWI interface.

The SWI interface is usually very simple. A typical FileCore-based filing system will have SWIs that functionally are a subset of those that FileCore provides. You implement these by calling the appropriate FileCore SWIs, making sure that you identify which filing system you are. RamFS implements all its SWIs like this, ADFS most of its. So unless you need to provide a lot of extra SWIs, you need do little more than provide the low-level routines that control the hardware.

For full details, see the chapter entitled *Writing a FileCore module* on page 4-63.

Technical details

FileCore-based filing systems are very like ADFS in operation and appearance (since ADFS is itself one). However, there is no reason why you need use FileCore only with discs; indeed, RamFS is also a FileCore-based filing system. The text that follows describes FileCore in terms of discs, disc drives, and so on. We felt you would find it easier to use than if we had used less familiar terminology – but please remember you can use other media too.

Disc formats

Logical layout

This table shows the logical layout of 'perfect' ADFS formats for floppy discs:

Format	Map	Zones	Directories	Boot block
L	Old	—	Old	No
D	Old	—	New	No
E	New	1	New	No
F	New	4	New	Yes

(The boot block is needed for F format floppies to specify which zone holds the map.)

and for hard discs:

Format	Map	Zones	Directories	Boot block
D	Old	—	New	Yes
E	New	≥1	New	Yes

For details of the various terms used above see the section entitled *Old maps* on page 3-191, the section entitled *New maps* on page 3-192, the section entitled *Directories* on page 3-201, and the section entitled *Boot blocks* on page 3-203.

Physical layout

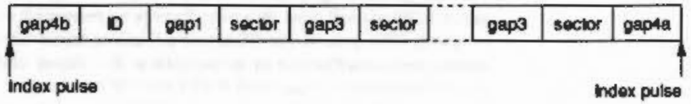
This table shows the physical layout of 'perfect' ADFS formats:

Format	Density	Sectors/track	Bytes/sector	Storage	Heads
L	Double	16	256	640K	1
D	Double	5	1024	800K	2
E	Double	5	1024	800K	2
F	Quad	10	1024	1.6M	2
Hard	—	—	—	≤512M	—

A head value of 1 means that the sides are sequenced, whereas a head value of 2 means that they are interleaved.

Track layout

A track is layed out as follows:



Due to mechanical variation in speed the time between the start and end varies, which is why there are gaps – they 'absorb' the speed variations. So, in words:

- *gap4b* is the gap between the **mechanical** index pulse and the **magnetic** index mark
- ID is the **magnetic** index mark
- *gap1* is the gap between the index mark and the first sector
- *sector* is a sector (see below)
- *gap3* is the gap between sectors
- *gap4b* is the gap between the last sector and the index pulse.

The magnetic index mark and the preceding gap4b are optional. Where they are absent, *gap1* is therefore the gap between the mechanical pulse and the first sector.

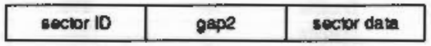
You should **never** rely on the presence or absence of the magnetic mark.

The size of *gap1* and *gap3* change between formats, whilst the other sizes remain constant. This table shows those gap sizes that vary (in bytes) and the sector skew (in sectors) of 'perfect' ADFS formats:

Format	**Gap 1 side 0**	**Gap 1 side 1**	**Gap 3**	**Sector skew**
L	42	42	57	0
D	32+271	32+0	90	0
E	32+271	32+0	90	0
F	50	50	90	2

Sector layout

A sector is layed out as follows:



- *gap2* is fixed due to hardware limitations; it is there to accommodate variations in hardware (different spin speeds etc)

- Each of *sector* ID and *sector data* have a preamble of null bytes, a synchronisation pattern, an identification byte (which says what sort of information follows: ID or Data), and the data itself (ID or data).

The reason the ID is separated from the data is that during sector writing the ID is read to determine which bit of the disc is currently going under the head, then the drive is switched to writing – which takes some time – and then a whole section of data is written (ie the sector ID or data).

Maps

A disc has a section of information, called a map, which controls the allocation of the disc to the files and directories. There are two types of maps used in RISC OS 3: the old maps used by L and D formats, and the new maps used by later formats:

Map	**Information stored**	**Compaction required**	**Recovery story**
Old	Free space	Yes	From directories
New	Space allocation	No	Two copies stored

New map discs have the following advantages over old map discs:

- Files need not be stored contiguously, so you don't need to compact the disc. (However, FileCore does try to create new map files in one block, and will also try to merge file fragments back together again if it is compacting a zone of the disc.)
- The disc map has no limit on size or number of entries, so 'Map full' errors do not occur.
- The map keeps a record of defects when the disc is formatted, so omits defective sectors.
- Defects are kept as objects on the disc, so they don't need to be taken into account when calculating disc addresses, and can be mapped out without reformatting.

Old maps

Old maps have the following format:

Name	**Bytes**	**Meaning**
FreeStart	82 × 3	Table of free space start sectors
Reserved	1	Reserved – must be zero
OldName0	5	Half disc name (interleaved with OldName1)
OldSize	3	Disc size in (256 byte) sectors
Check0	1	Checksum on first 256 bytes
FreeLen	82 × 3	Table of free space lengths
OldName1	5	Half disc name (interleaved with OldName0)

OldId	2	Disc id
OldBoot	1	Boot option (as in *Opt 4,*n*)
FreeEnd	1	Pointer to end of free space list
Check1	1	Checksum on second 256 bytes

The 82 three byte entries in the FreeStart and FreeLen tables are in units of 256 bytes. The entries are sorted low addressed free areas first. Contiguous free areas will have been merged together.

The full disc name is the joining together of the bytes in OldName0 and OldName1. The name is interleaved, with OldName0 providing the first character, OldName1 the second, and so on.

OldId is the disc's Id to identify when the disc has been modified.

If an old map does not end at a sector boundary, then it is padded with null bytes to the end of the sector. The sector immediately following the old map always holds the start of the root directory; see the section entitled *Directories* on page 3-201.

Calculating Check0 and Check1

These are checksums of the previous bytes in the map. They are calculated using repeated 8-bit ADCs on the bytes of the relevant map block, starting with a value 0:

If R0 is the accumulated checksum, then it starts at 0, and each byte is added as follows:

```
ADC  r0, r0, r1                r1 is the byte picked up
MOVS r0, r0, LSL #24           Shifts bit 8 into the carry bit
MOV  r0, r0, LSR #24           Not MOVS here to preserve the carry bit
```

Note that the check byte itself isn't included in the checksum; its value equals the checksum of the previous bytes.

New maps

A disc using a new map is divided into a number of *zones*, each of which is a contiguous section of the disc. The zones are numbered 0 upwards, so if there are *nzones* zones on a disc, the zone numbers are 0, 1, ..., *nzones* – 2 and *nzones* – 1 (ie zone 0 contains the lowest numbered sectors on the disc, and zone *nzones* – 1 the highest numbered sectors).

The map is located at the beginning of zone *nzones*/2 (rounded down). Hence, the map will sit at the beginning of the middle zone for discs with an odd number of zones, and the zone higher than the middle for discs with an even number of zones (examples: if *nzones*=7, the map is at the start of zone 3, which has 3 zones before it and after it; if *nzones* = 8 the map is at the start of zone 4, which has 4 zones before it and 3 after it).

The map is *nzones* sectors long: each sector of the map is known as a *map block*, and controls the allocation of a zone of the disc. The first map block controls zone 0, the second controls zone 1, and so on.

The general format of a map block is as follows:

Header
Disc record **(Zone 0 only)**
Allocation bytes
Unused

Header

A map block header is as follows:

Offset	Name	Meaning
0	ZoneCheck	Check byte for this zone's map block
1	FreeLink	Link to first free fragment in this zone
3	CrossCheck	Cross check byte for complete map

ZoneCheck is used to check that this zone's map block is valid; see the section entitled *Calculating ZoneCheck*... on page 3-197.

FreeLink is a fragment block giving the offset to the first free space fragment block in the allocation bytes; see page 3-195.

CrossChecks are combined to check that the whole map is self-consistent; see the section entitled *Calculating CrossCheck* on page 3-197.

Disc record

The format of a disc record is as follows:

Offset	Name	Meaning
0	*log2secsize*	Log_2 (sector size of disc in bytes)
1	*secspertrack*	Number of sectors per track
2	*heads*	Number of disc heads if sides interleaved Number of disc heads – 1 if sides sequenced (1 for old directories)
3	*density*	0 hard disc 1 single density (125Kbps FM) 2 double density (250Kbps FM) 3 double+ density (300Kbps FM) (ie higher rotation speed double density) 4 quad density (500Kbps FM)

		8 octal density (1000Kbps FM)
4	*idlen*	Length of id field of a map fragment, in bits
5	*log2bpmb*	Log_2 (number of bytes per map bit)
6	*skew*	Track to track sector skew for random access file allocation
7	*bootoption*	Boot option (as in *Opt 4,*n*)
8	*lowsector*	bits 0 - 5: lowest numbered sector id on a track
		bit 6: if set, treat sides as sequenced (rather than interleaved)
		bit 7: if set, double step disc
9	*nzones*	Number of zones in the map
10	*zone_spare*	Number of non-allocation bits between zones
12	*root*	Disc address of root directory
16	*disc_size*	Disc size, in bytes
20	*disc_id*	Disc cycle id
22	*disc_name*	Disc name
32	*disctype*	File type given to disc
36 - 59		Reserved – must be zero

Bytes 4 - 11 inclusive must be zero for old map discs.

As an example of how to use the logarithmic values, if the sector size was 1024, this is 2^{10}, so at offset 0 you would store 10.

You can use a disc record to specify the size of your media – this is how RamFS is able to be larger than an ordinary floppy disc.

The *lowsector* and *disctype* fields are not stored in the disc record kept on the disc, but are returned by ADFS_DescribeDisc.

Allocation bytes

The *allocation bytes* make up the section of the map block which controls the allocation of a zone. Together, the allocation bytes from all map blocks control the allocation of the whole disc. Each bit corresponds to an *allocation unit* on the disc. The size of the allocation units is defined in the disc record by *log2bpmb*, and so must be a power of two bytes. An allocation unit is not necessarily one sector – it may be smaller or larger.

Not only must space be logically mapped in whole allocation units; it must also be physically allocated in whole sectors. Consequently, the smallest unit by which allocation may be changed is the **larger** of the sector size and the allocation unit. This unit is known as the *granularity*.

A disc is split into a number of *disc objects*, each of which consists of one or more *fragments* spread over the surface of the disc. Fragments need not be held in the same zone, and their size can vary by whole units of granularity. Fragments have a minimum size, which is explained below.

Three disc objects are special, and contain:

- the bad sectors (for a perfect disc, this disc object will not be present)
- the boot block, map and root directory
- the free space.

All other disc objects contain either a directory (optionally with small files held within that directory), or one or more files that are held in a common directory. For a description of how disc objects can contain more than one object, see the section entitled **Internal disc addresses** on page 3-200 and the section entitled **Directories** on page 3-201.

The allocation bytes are treated as an array of bits, with the lsb of a byte coming before the msb in the array.

The array is split into a series of *fragment blocks*, each representing a fragment. The format of a fragment block is as follows:

Fragment id	length is *idlen* bits, as defined in the disc record
Zero or more 0 bits	
A terminating 1 bit	

Since each bit in the array corresponds with an allocation unit on the disc, the length of the fragment block (in bits) must be the same as the size of the fragment (in allocation units). The stream of 0 bits are used to pad the fragment block to the correct length.

There are two fragment ids with special meanings:

- A fragment id of 1 represents the object which contains all bad sectors, and the spare piece of map which hangs over the real end of the disc.
- A fragment id of 2 represents the object which contains the boot block, the map, and the root directory.

Other fragment ids represent either free space fragments, or allocated fragments:

- A fragment id for a free space fragment is the unsigned offset, in bits, from the beginning of its fragment block to the beginning of the next free space fragment block in the same map block (or 0 if there are no more).

  The chain hence always runs from the beginning of the map block to the end. The offset to the first free space fragment block is given by the FreeLink fragment block in the map block's header. Because that fragment block is 2 bytes long, and must have a terminating 1 bit, *idlen* cannot be greater than 15.

- A fragment id for an allocated fragment is a unique identifier for the disc object to which that space is allocated. Any other fragments allocated to the same disc object will have the same fragment id.

The following deductions can be made:

- The smallest fragment size on a disc is:

  (*idlen*+1) × allocation unit *rounded up to the nearest unit of granularity*

  because a fragment block cannot be smaller than *idlen*+1 bits (the *fragment id*, and the terminating 1 bit).

- *idlen* must be at least:

  log2secsize + 3 *ie* $\log_2$ *(sector size in bits)*

  to ensure that it is large enough to hold the maximum possible bit offset to the next free fragment block.

- The maximum number of *fragment ids* in a map block (and hence disc objects in a zone) is:

  allocation bytes × 8 / (*idlen* + 1) *ie allocation bits / minimum fragment size*

  This value is smaller for Zone 0 than for other zones, because it has an extra header, and hence fewer allocation bytes.

  The value for zones other than Zone 0 is – for a given disc – always the same, and is known as the *ids per zone*. It is easiest to calculate using fields from the disc record:

  ((1 << (*log2secsize* + 3)) – *zone_spare*) / (*idlen* + 1)

- The allocation unit cannot be so small as to require more than 15 bits to represent all the fragment ids possible, ie:

  (*ids per zone* × *nzones*) ≤ 2^{15}

  since the fragment id cannot be more than 15 bits long.

An object may have a number of fragments allocated to it in several zones. These fragments must be logically joined together in some way to make the object appear as a contiguous sequence of bytes. The naïve approach would be to have the first fragment of the disc be the first fragment of the object. New map discs do not do this. The first fragment in an object is the first fragment on the disc searching from zone (*fragment id / ids per zone*) upwards, wrapping round from the disc's end to its start. Any subsequent fragments belonging to the same disc object are joined in the order they are found by this search.

Object 2, being the object which carries the map with it, is special. It is always at the beginning of the middle zone, as opposed to being at the beginning of zone 0.

Maximum disc sizes

As observed above, there are a number of limitations placed on discs by new maps, depending on your choice of various parameters. The table below gives some idea of the theoretical maximum disc sizes that can be supported, depending on the sizes of the allocation unit and of the sectors:

Allocation unit	512 byte secs	1024 byte secs
256	up to 124Mb	up to 127Mb
512	up to 249Mb	up to 255Mb
1024	up to 503Mb	up to 511Mb
2048	up to 1007Mb	up to 1023Mb

In fact, other limitations in FileCore mean that discs can be no larger than 512Mbytes.

Calculating disc addresses

To translate an allocation bit in the map to a disc address, take the allocation bit's bit offset from the beginning of the bit array (ie the concatenation of all allocation bytes) and multiply this offset by the bytes per map bit (this multiplication is equivalent to shifting the offset left by *log2bpmb*, which is why the log2 value is stored in the disc record).

This result is the byte offset across the disc of the beginning of the section of the disc which corresponds to the given map bit. This quantity can be passed to FS_DiscOp SWIs directly.

Calculating CrossCheck

These bytes provide a means to check that the set of zones match each other. To check the set matches, these bytes are exclusive-ORd (EOR) with each other: the answer must be &FF. They are modified whenever more than one zone map is modified. (The algorithm is not important, just so long as the bytes of the changed maps change and that the EOR of all these bytes remains at &FF).

Calculating ZoneCheck...

This, as described previously, is a check byte on a given zone sector. Below are some code fragments you can use to calculate this value, using either C or assembler:

...using C

```
unsigned char map_zone_valid_byte
(
    void const * const map,
    disc_record const * const discrec,
    unsigned int zone
)
{
    unsigned char const * const map_base = map;
    unsigned int sum_vector0;
    unsigned int sum_vector1;
    unsigned int sum_vector2;
    unsigned int sum_vector3;
    unsigned int zone_start;
    unsigned int rover;

    sum_vector0 = 0;
    sum_vector1 = 0;
    sum_vector2 = 0;
    sum_vector3 = 0;

    zone_start = zone<<discrec->log2_sector_size;
    for ( rover = ((zone+1)<<discrec->log2_sector_size)-4 ;
          rover > zone_start;
          rover-=4 )
{
        sum_vector0 += map_base[rover+0] + (sum_vector3>>8);
        sum_vector3 &= 0xff;
        sum_vector1 += map_base[rover+1] + (sum_vector0>>8);
        sum_vector0 &= 0xff;
        sum_vector2 += map_base[rover+2] + (sum_vector1>>8);
         sum_vector1 &= 0xff;
         sum_vector3 += map_base[rover+3] + (sum_vector2>>8);
         sum_vector2 &= 0xff;
}
/*
    Don't add the check byte when calculating its value
*/
    sum_vector0 += (sum_vector3>>8);
    sum_vector1 += map_base[rover+1] + (sum_vector0>>8);
    sum_vector2 += map_base[rover+2] + (sum_vector1>>8);
    sum_vector3 += map_base[rover+3] + (sum_vector2>>8);

    return (unsigned char)
          ((sum_vector0^sum_vector1^sum_vector2^sum_vector3)
          & 0xff);
}
```

...using assembler

```
; ========
; NewCheck
; ========

;entry
; R0 -> start
; R1 length ( must be multiple of 32 )

;exit
; LR check byte, Z=0 <=> good

NewCheck ROUT
 Push "R1-R9,LR"
 MOV LR, #0
 ADDS R1, R1, R0   ;C=0
05                 ;loop optimised as winnies may have many zones
 LDMDB R1!,(R2-R9)
 ADCS LR, LR, R9
 ADCS LR, LR, R8
 ADCS LR, LR, R7
 ADCS LR, LR, R6
 ADCS LR, LR, R5
 ADCS LR, LR, R4
 ADCS LR, LR, R3
 ADCS LR, LR, R2
 TEQS R1, R0       ;preserves C
 BNE  %BT05
 AND  R2, R2, #&FF ;ignore old sum
 SUB  LR, LR, R2
 EOR  LR, LR, LR, LSR #16
 EOR  LR, LR, LR, LSR #8
 AND  LR, LR, #&FF
 CMPS R2, LR
 Pull "R1-R9,PC"
```

Disc addresses

In reading the following description, you should take special care over the difference between an *object* (**ie a single file or a directory**) and a *disc object* (ie **a logical group of fragments** on a new map disc, that may contain one or more objects).

FileCore uses two different types of disc address.

- The first is a normal physical disc address, giving the offset in bytes of data from the start of the disc.
- The second is an internal format used with new map discs, that specifies an object in terms of its fragment id, and its offset in sectors within that fragment.

  This is how a single disc object can hold many objects. The internal address of each object within the disc object will have the same fragment id, but a different offset within that fragment.

Physical disc addresses

The physical disc address of a byte gives the number of bytes it is into the disc, when it is read in its sequential order from the start. To calculate the physical disc address of a byte you need to know:

- its head number **h**
- its track number **t**
- its sector number **s**
- the number of bytes into the sector **b**
- the number of heads on the drive **H**
- the number of sectors per track **S**
- the number of bytes per sector **B**
- the number of defective sectors earlier on the disc **x** (for old map hard discs only – use zero for old map floppy discs or new map discs)

You can use this formula for any disc – except an L-format one – to get the values of bits 0 - 28 inclusive:

$$\text{address} = ((t \times H + h) \times S + s - x) \times B + b$$

Tracks, heads and sectors are all counted from zero.

Bits 29 - 31 contain the drive number.

See also the section entitled *Calculating disc addresses* on page 3-197, which tells you how to calculate a physical disc address from the position of an allocation bit in a new map.

Internal disc addresses

Internal disc addresses are used by new map discs only. An object's internal disc address is in the following binary form:

*ddd*00000 0*fffffff ffffffff* ssssssss

- *ddd* is the disc number (not useful outside FileCore)
- *fffffffffffffff* is the fragment id
- ssssssss is the sector offset within the object.

If the sector offset is 0, then the object does not share its disc object, and is located at the start of the disc object.

If the sector offset is non-zero (eg is s), then the object shares its disc object, and is located at the start of the sth sector of the disc object. So disc address:

0x00000233

means that this object (in fact the directory $) starts at the &33th sector in object 2. Note that the &33th sector starts &32 sectors into the disc object (ie the 1st sector is at the start of the object).

Directories

There are two types of directories used in RISC OS: the old directories used by L format, and the new directories used by later formats:

Directories	**Size (entries)**	**Size (bytes)**	**Top bit set chars**
Old	47	1280	No
New	77	2048	Yes

For both formats the directory is arranged as follows:

DirHeader
Entries[*n*] where *n* = 47 or 77, as above
DirTail

The header and tail contain information about this directory, and the entries are the directory entries.

DirHeaders

The two directory formats have the same DirHeader:

Name	**Bytes**	**Meaning**
StartMasSeq	1	Update sequence number to check dir start with dir end
StartName	4	'Hugo' or 'Nick'

Entries

The two directory formats have mostly the same entry format:

Name	**Bytes**	**Meaning**
DirObName	10	Name of object
DirLoad	4	Load address of object
DirExec	4	Exec address of object
DirLen	4	Length of object
DirIndDiscAdd	3	Indirect disc address of object
OldDirObSeq **or** NewDirAtts	1	

DirTails

The DirTail formats are, however, quite different:

Old DirTail

Name	Bytes	Meaning
OldDirLastMark	1	0 to indicate end of entries
OldDirName	10	Directory name
OldDirParent	3	Indirect disc address of parent directory
OldDirTitle	19	Directory title
Reserved	14	Reserved – must be zero
EndMasSeq	1	To match with StartMasSeq
EndName	4	'Hugo' or 'Nick', to match with StartName
DirCheckByte	1	Check byte on directory

New DirTail

Name	Bytes	Meaning
NewDirLastMark	1	0 to indicate end of entries
Reserved	2	Reserved – must be zero
NewDirParent	3	Indirect disc address of parent directory
NewDirTitle	19	Directory title
NewDirName	10	Directory name
EndMasSeq	1	To match with StartMasSeq
EndName	4	'Hugo' or 'Nick', to match with StartName
DirCheckByte	1	Check byte on directory

Notes

The last entry is indicated by there being a 0 in the first byte of the next entry's DirObName. The xxDirLastMark entry is there so that when the directory is full, and hence the last entry is not followed by a null DirObName, it is still followed by a null byte to indicate the end of the directory.

DirObNames and DirNames are control character terminated, and may be the full length of their fields they occupy (in which case there is no terminator).

The indirect disc address of an object on an old map disc is the least significant 3 bytes of its physical disc address. Likewise, the indirect disc address of an object on a new map disc is the least significant 3 bytes of its internal disc address. For an explanation, see the section entitled *Disc addresses* on page 3-199.

Calculating StartMasSeq and EndMasSeq

StartMasSeq and EndMasSeq are there to check whether the directory was completely written out when it was last written out. For an unbroken directory they are always equal, and are increased by one (wrapping at 255 back to 0) whenever the directory is updated. This means that if the writing of the directory was stopped halfway through then the start and end master sequence numbers will not be the same, and so the directory will then be identified as broken. Their values should equal each other, but, apart from that, they can be anything.

Calculating DirCheckByte

This is an accumulation of the used bytes in a directory. The used bytes are all the bytes excluding the hole between the last directory entry and the beginning of the structure at the tail of the directory. The generation of the check byte is best described as an algorithm:

- Starting at 0 an accumulation process is performed on a number of values. Whatever the sort of the value (byte or word) it is accumulated in the same way. Assuming r0 is the accumulation register and r1 the value to accumulate this is the accumulation performed:

```
EOR r0, r1, r0, ROR #13
```

- All the whole words at the start of the directory are accumulated. This will leave a number of bytes (0 to 3) in the last directory entry (or at the end of the start structure in a directory if it's empty).
- The last few bytes at the start of the directory are accumulated individually.
- The first few bytes at the beginning of the end structure of the directory are accumulated. This is done to leave only a whole number of words left in the directory to be accumulated.
- The last whole words in the directory are accumulated, except the very last word which is excluded as it contains the check byte.
- The accumulated word has its four bytes exclusive-ORd (EOR) together. This value is the check byte.

Boot blocks

Hard discs contain a 512 byte *boot block* at disc address &C00, which contains important information. (On a disc with 256-byte sectors, such as ADFS uses, this corresponds to sectors 12 and 13 on the disc.) A boot block has the following format:

Offset	Contents
&000 upwards	Defective sector list
&1BF downwards	Hardware-dependent information
&1C0 - &1FB	Disc record (see page 3-193)
&1FC - &1FE	Reserved – must be zero
&1FF	Check sum byte

Note that in memory, this information would be stored in the order disc record, then defect list/hardware parameters. This is to facilitate passing the values to FileCore SWIs.

Defect list

A *defect list* is a list of words. Each word contains the disc address of the first byte of a sector which has a defect. This address is an absolute one, and does not take into account preceding defective sectors. The list is terminated by a word whose value is &200000*xx*. The byte *xx* is a check-byte calculated from the previous words. Assuming this word is initially set to &20000000, it can be correctly updated using this routine:

On entry

Ra = pointer to start of defect list

On exit

Ra	corrupt
Rb	check byte
Rc	corrupt

```
        MOV     Rb,#0                   ;init check
loop
        LDR     Rc,[Ra],#4;             get next entry
        CMPS    Rc,#&20000000           ;all done ?
        EORCC   Rb,Rc,Rb,ROR #13
        BCC     loop
        EOR     Rb,Rb,Rb,LSR #16        ;compress word to byte
        EOR     Rb,Rb,Rb,LSR #8
        AND     Rb,Rb,#&FF
```

Hardware-dependent information

There is no guarantee how many bytes the hardware-dependent information may take up. As an example of use of this space, for the HD63463 controller the hardware parameters have the following contents:

Offset	Contents
&1B0 - &1B2	Unused
&1B3	Step pulse low
&1B4	Gap 2
&1B5	Gap 3
&1B6	Step pulse high
&1B7	Gap 1
&1B8 - &1B9	Low current cylinder
&1BA - &1BB	Pre-compensation cylinder
&1BC - &1BF	Unadjusted parking disc address

The boot block's disc record

The purpose of the boot block's disc record is to give the necessary information to find the disc's map. You should not rely on the information it contains for any other purpose, unless it is unavailable in the disc's map. Consequently:

- For an old map disc, you should use the boot block's disc record to find the map. If information you require is held in the map, you must use that in preference to the boot block's disc record.
- For a new map disc, you should use the boot block's disc record to find the map. Once you have found the map you should then always use its disc record, rather than the boot block's.

For the format of a disc record, see the section entitled *Disc record* on page 3-193.

Calculating the boot block's check sum byte

The last byte of the boot block is a check sum byte whose value is calculated as follows:

- Perform an 8 bit add with carry on each of the other bytes in the block, starting with value 0.

In assembler this might be done as follows:

```
; entry: R0=start, R1=block length
; exit: R0,R1 preserved, R2=checksum

CheckSum ROUT
        STMFD   R13!, {R1, LR}

        ADDS    LR, R0, R1              ;->end+1 C=0
        SUB     R1, LR, #1              ;->check byte
        MOV     R2, #0
        B       %FT20
10
        LDRB    LR, [R1,#-1] !          ;get next byte
        ADC     R2, R2, LR              ;add into checksum
        MOVS    R2, R2, LSL #24         ;bit 8 = carry
        MOV     R2, R2, LSR #24
20
        TEQS    R0, R1
        BNE     %BT10                   ;loop until done

        LDMFD   R13!, {R1, LR}
```

Note that the checksum doesn't include the last byte.

Data format

Files stored using FileCore are sequences of bytes which always begin at the start of a sector and extend for the number of sectors necessary to accommodate the data contained in the file. The last sector used to accommodate the file may have a number of unused bytes at the end of it. The last 'data' byte in the file is derived from the file length stored in the catalogue entry for the file, or if the file is open, from its extent.

Disc identifiers

Many of the commands described below allow discs to be specified. Generally, you can refer to a disc by its physical drive number (eg 0 for the built-in floppy), or by its name.

Drive numbers

FileCore supports 8 drives. Drive numbers 0 - 3 are 'floppy disc drives', and drive numbers 4 - 7 are 'hard disc drives'. You cannot implement a filing system under FileCore that has more than four drives of the same physical type.

Disc names

The disc name is set using *NameDisc (see page 3-248). When you refer to a disc by name it will be used if it is in a drive. Otherwise a 'Disc not present' error will be given if the disc has been previously seen, or a 'Disc not known' error if the disc has not been seen.

Machine code programs can trap these errors before they are issued. This allows the user to be prompted to insert the disc into the drive. See OS_UpCall 1 *and* 2 (SWI &33) on page 1-169 for details.

In fact, disc names may be used in any pathname given to the system. When used in a pathname, the disc name (or number) must be prefixed by a colon. Examples of pathnames with disc specifiers are:

```
*Cat :MikeDisc.fonts
*Info :4.LIB*.*
```

Note that :drive really means :drive.$.

Disc names can have wildcards in them, so long as the name only matches one of the discs that FileCore knows about for the filing system. If more than one name matches FileCore will return an 'Ambiguous disc name' error.

You are very strongly recommended to use disc names rather than drive numbers when you write programs.

Changing discs

FileCore keeps track of eight disc names per filing system, on a first in, first out basis. When you eject a floppy disc from the drive, FileCore still 'knows' about it. This means that if there are any directories set on that disc (the current directory, user root directory, or library), they will still be associated with it. Thus any attempt to load or run a file will result in a 'Disc not present/known' error.

However, this means that you can replace the disc and still use it, as if it had never been ejected. The same applies to open files on the disc; they remain open and associated with that disc until they are closed.

You can cause the old directories to be overridden by `*Mounting` a new disc once it has been inserted. This resets the CSD and so on. Alternatively, if you unset the directories (using *NoDir, *NoLib and *NoURD), then FileCore will use certain defaults when operations on these are required.

If there is no current directory, FileCore will use $ on the default drive. This is the configured default, or the one set by the last *Drive command.

If there is no user root directory set, then references to that directory will use $ on the default drive.

If there is no library set, then FileCore will try &.Library, $.Library and then the current directory, in that order.

Current selections

The currently selected directory, user root directory and library directory are all stored independently for each FileCore-based filing system.

Service Calls

Service_IdentifyDisc (Service Call &69)

Identify the disc format

On entry

R1 = &69 (reason code)
R5 = pointer to disc record
R6 = sector cache handle
R8 = pointer to FileCore instance private word to use

On exit

If the format has been identified:

R1 = 0 to claim call
R2 = filetype number for given disc format
R5 = pointer to disc record, which has been modified
R6 = new sector cache handle
R8 preserved

Otherwise:

R1, R5 preserved
R6 = new sector cache handle
R8 preserved

Use

When an image filing system receives this service call it should:

1 Check the sector size, sectors per track, density, heads and lowest numbered sector id on a track (held in the disc record – see the section entitled *Disc record* on page 3-193) to see whether these correspond to a format it understands. However, it should not do so if any of the sector size, sectors per track, density or heads are 0, since this means they were not supplied by FileCore_MiscOp 0 (see page 3-227); this should only occur on hard discs.

2 If it does not recognise the sector scheme, it should pass on the service call, unclaimed.

3 If it does recognise the sector scheme, it should then update the disc record's values for the disc size, sequence sides, double step and heads so they correspond with the recognised format.

It should only adjust the heads field in line with the sequence sides value: when clearing the sequence sides bit from being set it should increment the heads field by one, and when setting the sequence sides bit from being clear it should decrement the heads field by one – but if the heads field was 0 it must remain so.

4 Check the sector contents to see whether these correspond to a format it understands. It should read the sectors using FileCore_DiscOp 9 (see page 3-210) with:

- the options bits in R1 set to 2_01x0 (1 second timeout; ignore escape; scatter list optional; no alternative defect list)
- the pointer to an alternative disc record in R1 addressing the one supplied in the service call
- the disc number within the disc address in R2 matching that given in the service call disc record's root directory address (which is set to byte 0 on the relevant disc).

5 If it does not recognise the sector contents, it should pass on the service call, unclaimed, with, if necessary, the new value for R6 set up by FileCore_DiscOp 9.

6 If it does recognise the sector contents, it should then update the disc record's values for the disc cycle id and disc name, and claim the service call. The returned disc record will be used in further accesses, and so must have the heads and disc size correct. The disc cycle id should be one of:

- an id stored on the disc which changes each time the disc is updated'
- a value (eg CRC) calculated from a proportion of the disc which is likely to change when the disc is updated, such as the map.

FileCore itself claims this service call to recognise those discs it knows about.

In summary:

- Check sector size, sectors per track, density, heads and low sector
- Pass on service call if no match
- Update disc size and heads fields and sequence sides and double step bits
- Check sector contents
- Pass on service call if no match
- Update disc cycle id and disc name
- Claim service.

SWI Calls

FileCore_DiscOp (SWI &40540)

Performs various operations on a disc

On entry

R1 bits 0 - 3 = reason code
bits 4 - 7 = option bits
bits 8 - 31 = bits 2 - 25 of pointer to alternative disc record, or zero
R2 = disc address
R3 = pointer to buffer
R4 = length in bytes
R6 = cache handle
R8 = pointer to FileCore instance private word

On exit

R1 preserved
R2 = disc address of next byte to be transferred
R3 = pointer to next buffer location to be transferred
R4 = number of bytes not transferred

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call performs various disc operations as specified by bits 0 - 3 of R1:

Value	Meaning	Uses	Updates
0	Verify	R2, R4	R2, R4
1	Read sectors	R2, R3, R4	R2, R3, R4
2	Write sectors	R2, R3, R4	R2, R3, R4
3	Floppy disc: read track	R2, R3	
	Hard disc: read Id	R2, R3	
4	Write track	R2, R3	
5	Seek (used only to park)	R2	
6	Restore	R2	
7	Floppy disc: step in		
8	Floppy disc: step out		
9	Read sectors via cache	R2, R3, R4, R6	R2, R3, R4, R6
15	Hard disc: specify	R2	

Option bits

The option bits have the following meanings:

Bit 4

This bit is set if an alternate defect list for a hard disc is to be used. This is assumed to be in RAM 64 bytes after the start of the disc record pointed to by R5.

This bit may only be set for old map discs.

Bit 5

If this bit is set, then the meaning of R3 is altered. It does not point to the area of RAM to or from which the disc data is to be transferred. Instead, it points to a word-aligned list of memory address/length pairs. All but the last of these lengths must be a multiple of the sector size. These word-pairs are used for the transfer until the total number of bytes given in R4 has been transferred.

On exit, R3 points to the first pair which wasn't fully used, and this pair is updated to reflect the new start address/bytes remaining, so that a subsequent call would continue from where this call has finished.

This bit may only be set for reason codes 0 - 2.

Bit 6

If this bit is set then escape conditions are ignored during the operation, otherwise they cause it to be aborted.

Bit 7

If this bit is set, then the usual time-out for floppy discs of 1 second is not used. Instead FileCore will wait (forever if necessary) for the drive to become ready.

Disc address

The disc address must be on a sector boundary for reason codes 0 - 2, and on a track boundary for other reason codes. Note that you must make allowances for any defects, as the disc address is not corrected for them.

For reason code 6 (restore), the disc address is only used for the drive number; the bottom 29 bits should be set to zero.

The *specify disc* command (reason code 15) sets up the defective sector list, hardware information and disc description from the disc record supplied. Note that in memory, this information must be stored in the order disc record, then defect list/hardware parameters.

Read Track/ID (reason code 3)

If the alternate defect list option bit (bit 4) is set in R1 on entry when reading a track/ID, then a whole track's worth of ID fields is read. This usage is not available under RISC OS 2.

The call reads 4 bytes of sector ID information into the buffer pointed to by R3 for every sector on the track. The order of data is:

Cylinder
Head
Sector number
Sector size (0= 128, 1= 256, etc)

The operation is terminated after 200mS (1 revolution).

The first sector ID transferred will normally be that following the index mark (it may be the second if there is abnormal interrupt latency from the index pulse interrupt). The first two ID's read may also be duplicated at the buffer end due to interrupt latency. Consequently the buffer should be at least 16 bytes longer than the maximum number of IDs expected (512 bytes at most).

The disc record provided is updated to return the actual number of sectors per track found (at offset 1). Note to use this option you **must** provide a valid defect list, which at a minimum is a word of &20000000 following on after the disc record.

Write Track (reason code 4)

If R3 (the buffer pointer) is zero on entry when writing a track, then R4 is instead used to point to a disc format structure. This usage is not available under RISC OS 2.

The call formats a track of the specified disc. An error is generated if the specified format is not possible to generate, or if the track requested is outside the valid range. The tracks are numbered from 0 to (number of tracks) – 1. The mapping of the address is controlled by the disc structure record.

The disc format structure is as follows:

Offset	Length	Meaning		
0	4	Sector size in bytes (which must be a multiple of 128)		
4	4	Gap1		
8	4	Reserved – must be zero		
12	4	Gap3		
16	1	Sectors per track		
17	1	Density:		
		1	single density (125Kbps FM)	
		2	double density (250Kbps FM)	
		3	double+ density (300Kbps FM) (ie higher rotation speed double density)	
		4	quad density (500Kbps FM)	
		8	octal density (1000Kbps FM)	
18	1	Options:		
		bit 0	1	index mark required
		bit 1	1	double step
		bits 2-3	0	interleave sides
			1 - 3	sequence sides
		bits 4-7		reserved – must be 0
19	1	Sector fill value		
20	4	Cylinders per drive (normally 80)		
24	12	Reserved – must be 0		
36	?	Sector ID buffer, 1 word per sector:		
		bits 0 - 7		Cylinder number mod 256
		bits 8 - 15		Head (0 for side 1, 1 for side 2)
		bits 16 - 23		Sector number
		bits 24 - 31		Log_2 (sector size) – 7, eg 1 for 256 byte sector

Read sectors via cache (reason code 9)

This reason code reads sectors via a cache held in the RMA. It is not available under RISC OS 2.

To start a sequence of these operations, set R6 (the cache handle) to zero on entry. Its value may be updated on exit, and subsequent calls should use this new value.

Bits 4 - 7 of R1 should be zero, and are ignored if set.

To discard the cache once finished, call FileCore_DiscardReadSectorsCache (see page 3-222).

Related SWIs

None

Related vectors

None

FileCore_Create (SWI &40541)

Creates a new instantiation of an ADFS-like filing system

On entry

R0 = pointer to descriptor block
R1 = pointer to calling module's base
R2 = pointer to calling module's private word
R3 bits 0 - 7 = number of floppies
bits 8 - 15 = number of hard discs
bits 16 - 24 = default drive
bits 25 - 31 = start up options
R4 = suggested size for directory cache
R5 = suggested number of 1072 byte buffers for file cache
R6 = hard disc map sizes

On exit

R0 = pointer to FileCore instance private word
R1 = address to call after completing background floppy op
R2 = address to call after completing background hard disc op
R3 = address to call to release FIQ after low level op

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call creates a new instantiation of an ADFS-like filing system. It must be called on initialisation by any filing system module that is adding itself to FileCore.

The descriptor block is described in the chapter entitled *Writing a FileCore module* on page 4-63.

The only start-up option (passed in bits 25 - 31 of R3) currently supported is *No directory state* which is indicated by setting bit 30. All other bits representing start-up options must be clear.

If the filing system does not support background transfers of data, R5 must be zero.

The hard disc map sizes are given using 1 byte for each disc. The byte should contain *map size*/256 (ie 2 for the old map). This is just a good guess and should not involve starting up the drives to read from them You might store this in the CMOS RAM.

You must store the FileCore instance private word returned by this SWI in your module workspace; it is your module's means of identifying itself to FileCore.

When your module calls the addresses returned in R1 - R3, it must be in SVC mode with R12 holding the value of R0 that this SWI returned. Interrupts need not be disabled. R0, R1, R3 - R11 and R13 will be preserved by FileCore over these calls.

Related SWIs

None

Related vectors

None

FileCore_Drives (SWI &40542)

Returns information on the filing system's drives

On entry

R8 = pointer to FileCore instance private word

On exit

R0 = default drive
R1 = number of floppy drives
R2 = number of hard disc drives

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns information on the filing system's drives.

Related SWIs

None

Related vectors

None

FileCore_FreeSpace (SWI &40543)

Returns information on a disc's free space

On entry

R0 = pointer to disc specifier (null terminated)
R8 = pointer to FileCore instance private word

On exit

R0 = total free space on disc
R1 = size of largest object that can be created

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the total free space on the given disc, and the largest object that can be created on it.

Related SWIs

None

Related vectors

None

FileCore_FloppyStructure (SWI &40544)

Creates a RAM image of a floppy disc map and root directory entry

On entry

R0 = pointer to buffer
R1 = pointer to disc record describing shape and format
R2 bit 7 set for old directory structure
bit 6 set for old map
R3 = pointer to list of defects

On exit

R3 = total size of structure created

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call creates a RAM image of a floppy disc map and root directory entry.

The pointer to a list of defects is only needed for new map discs. They must be byte addresses giving the start of defective sectors, and terminated with &20000000.

You do not need to know a FileCore instantiation private word to use this call; instead the disc record tells FileCore which filing system is involved.

Related SWIs

None

Related vectors

None

FileCore_DescribeDisc (SWI &40545)

Returns a disc record describing a disc's shape and format

On entry

R0 = pointer to disc specifier (null terminated)
R1 = pointer to 64 byte block
R8 = pointer to FileCore instance private word

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns a disc record in the 64 byte block passed to it. The record describes the disc's shape and format. For a definition of the format of a disc record, see the section entitled *Disc record* on page 3-193.

Related SWIs

None

Related vectors

None

FileCore_DiscardReadSectorsCache (SWI &40546)

Discards the cache of read sectors created by FileCore_DiscOp 9

On entry

R6 = Cache handle

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call discards the cache of read sectors created by FileCore_DiscOp 9 (see page 3-213).

This call is not available under RISC OS 2.

Related SWIs

None

Related vectors

None

FileCore_DiscFormat (SWI &40547)

Fills in a disc format structure with parameters for the specified format

On entry

R0 = pointer to disc format structure to be filled in
R1 = SWI number to call to vet disc format (eg ADFS_VetFormat)
R2 = parameter in R1 to use when calling vetting SWI
R3 = format specifier

On exit

R0 - R3 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call fills in the disc format structure pointed to by R0 with the 'perfect' parameters for the specified format, taking no account of the abilities of the available hardware that will have to perform the format. Once filled in, this SWI calls the vetting SWI to check the format structure for achievability on the available hardware. The vetting SWI may generate an error if the format differs widely from what can be achieved; alternatively it may alter the format structure to the closest match that can be achieved. The vetting SWI then returns to this SWI, which checks whether the format block – as updated by the vetting SWI – is still an adequate match for the desired format. If it is, this SWI returns to its caller; otherwise it generates an error.

The following format specifiers are recognised:

Value	Meaning
&80	L format floppy
&81	D format floppy
&82	E format floppy
&83	F format floppy

The returned disc format structure contains the following information:

Offset	Length	Meaning		
0	4	Sector size in bytes (which will be a multiple of 128)		
4	4	Gap1 side 0		
8	4	Gap1 side 1		
12	4	Gap3		
16	1	Sectors per track		
17	1	Density:		
		1	single density (125Kbps FM)	
		2	double density (250Kbps FM)	
		3	double+ density (300Kbps FM) (ie higher rotation speed double density)	
		4	quad density (500Kbps FM)	
		8	octal density (1000Kbps FM)	
18	1	Options:		
		bit 0	1	index mark required
		bit 1	1	double step
		bits 2-3	0	interleave sides
			1	format side 1 only
			2	format side 2 only
			3	sequence sides
		bits 4-7		reserved – must be 0
19	1	Start sector number on a track		
20	1	Sector interleave		
21	1	Side/side sector skew (signed)		
22	1	Track/track sector skew (signed)		
23	1	Sector fill value		
24	4	Number of tracks to format (ie cylinders/drive: normally 80)		
28	36	Reserved – must be zero		

This structure tells you how to format a disc. Note that it differs from that used in FileCore_DiscOp to actually format a track (see page 3-213). The differences are because the DiscOp structure only specifies the format of a single track.

This call is not available under RISC OS 2.

Related SWIs

ADFS_VetFormat (SWI &40426), DOSFS_DiscFormat (SWI &41AC0)

Related vectors

None

FileCore_LayoutStructure (SWI &40548)

Lays out into the specified file a set of structures for its format

On entry

R0 = identifier of particular format to lay out
R1 = pointer to bad block list
R2 = pointer to null-terminated disc name
R3 = file handle

On exit

R0 - R3 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call lays out into the specified file a set of structures corresponding to the identified format. The format identifier is a pointer to a disc record. An error is returned if the specified format can not map out defects, and there were defects in the defect list.

This call is not available under RISC OS 2.

Related SWIs

None

Related vectors

None

FileCore_MiscOp (SWI &40549)

Perform miscellaneous functions for accessing drives

On entry

R0 = reason code
R1 = drive
R2 - R5 depend on reason code
R8 = pointer to FileCore instance private word

On exit

R0 - R6 depend on reason code

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call performs miscellaneous functions for accessing drives, depending on the reason code in R0. Valid reason codes are:

Value	Meaning
0	Mount
1	Poll changed
2	Lock drive
3	Unlock drive
4	Poll period

This call is not available under RISC OS 2.

Related SWIs

None

Related vectors

None

FileCore_MiscOp 0 (SWI &40549)

Mounts a disc, reading in the data asked for

On entry

R0 = 0
R1 = drive
R2 = disc address to read from
R3 = pointer to buffer
R4 = length to read into buffer
R5 = pointer to disc record to fill in (floppies and floppy-like hard discs only)
R8 = pointer to FileCore instance private word

On exit

R1 - R5 preserved

Use

This call mounts a disc, reading in the data asked for.

Floppy discs, and hard discs that may be mounted like floppies

For a floppy disc, and for hard discs where bit 4 of the descriptor block flags is set, this call asks the given filing system to first identify the disc's format. The suggested density to try first is given in the disc record; if this is not successful, the filing system should then try other densities. The following order is suggested:

1 Quad density

2 Double density

3 Octal density

4 Single density

5 Double+ density

Once the filing system has identified the disc's format, it fills in the *log2secsize*, *secspertrack*, *heads*, *density*, *lowsector* and *root* values in the disc record (see the section entitled Disc *record* on page 3-193).

- If *log2secsize* ≤ 8, then it gives *heads* the value (actual number of heads–1), and sets bit 6 of *lowsector*, so sides are treated as sequenced. Otherwise (ie when *log2secsize* > 8) it gives *heads* the value (actual number of heads), and clears bit 6 of *lowsector*, so sides are treated as interleaved.

- The filing system clears bit 7 of *lowsector*; this is used as an initial value, which FileCore subsequently corrects if necessary.

Having filled in the disc record, the filing system then reads in the data asked for.

Other hard discs

For hard discs where bit 4 of the descriptor block flags is clear (see the section entitled *Descriptor block* on page 4-63), this merely asks the given filing systems to read in the data asked for. This typically necessitates it reading the boot block off the disc; if the disc doesn't have one, the filing system generates one itself.

FileCore_MiscOp 1 (SWI &40549)

Poll changed

On entry

R0 = 1
R1 = drive
R2 = sequence number
R8 = pointer to FileCore instance private word

On exit

R2 = sequence number
R3 = result flags

Use

The sequence number is to ensure no changes are lost due to reset being pressed. Both the given filing system and the FileCore incarnation should start with a sequence number of 0 for each drive. The filing system increments the sequence number with each change of state. If the filing system finds the entry sequence number does not match its copy it should return changed/maybe changed, depending on whether the disc changed line works/doesn't work.

The bits in the result flags have the following meanings:

Bit	Meaning when set
0	not changed
1	maybe changed
2	changed
3	empty
4	ready
5	drive is 40 track
6	empty works
7	changed works
8	disc in drive is high density
9	density sensing works
10	ready works
11 - 31	reserved – must be zero

Exactly one of bits 0 - 3 must be set. Once bit 6 or 7 is returned set for a given drive, they must always be so.

FileCore_MiscOp 2 (SWI &40549)

Locks a disc in a floppy drive

On entry

R0 = 2
R1 = floppy drive
R8 = pointer to FileCore instance private word

On exit

—

Use

This call locks a disc in a drive; you can only use it for a floppy drive. It should at least ensure that the drive light stays on until unlocked. Note that locks are counted, so each 'Lock drive' must be matched by an 'Unlock drive'.

FileCore_MiscOp 3 (SWI &40549)

Unlocks a disc in a floppy drive

On entry

R0 = 3
R1 = drive
R8 = pointer to FileCore instance private word

On exit

—

Use

This call can only be called for a floppy drive. It reverses a single 'Lock drive' MiscOp. Note that locks are counted, so 'Unlock drive' must be called for each 'Lock drive'.

FileCore_MiscOp 4 (SWI &40549)

Informs FileCore of the minimum period between polling for disc insertion

On entry

R0 = 4
R1 = drive
R8 = pointer to FileCore instance private word

On exit

R5 = minimum polling period (in centi-seconds), or –1 if disc changed doesn't work
R6 = pointer to media type string: eg 'disc' for ADFS

Use

This call informs FileCore of the minimum period between polling for disc insertion under the given filing system. This is so that drive lights do not remain continuously illuminated.

The values are re-exported by FileCore in the up calls MediaNotPresent and MediaNotKnown. The value applies to all drives rather than a particular drive.

* Commands

*Backup

Copies the used part of a floppy disc.

Syntax

```
*Backup source_drive dest_drive [Q]
```

Parameters

source_drive	the number of the source floppy drive (0 to 3)
dest_drive	the number of the destination floppy drive (0 to 3)
Q	speeds up the operation, by using the application work area as a buffer if extra room is needed to perform the backup, so fewer disc accesses are done. You must save any work you have done and quit any applications you are using before using this option.

Use

*Backup copies the used part of one floppy disc to another; free space is not copied. If the source drive is the same as the destination (as it is on a single floppy drive system), you will be prompted to swap the disc, as necessary.

The command only applies to floppy, not hard discs.

Example

```
*Backup 0 1
```

Related commands

*Copy

*Bye

Ends a filing system session.

Syntax

```
*Bye
```

Use

*Bye ends a filing system session by closing all files, unsetting all directories and libraries, forgetting all floppy disc names and parking the heads of hard discs to their 'transit position' so that the hard disc unit can be moved without risking damage to the read/write head.

You should check that the correct filing system is the current one before you use this command, or alternatively precede the command by the filing system name. For example you could end an ADFS session when another filing system is your current one by typing:

```
*adfs:Bye
```

Related commands

*Close, *Dismount, *Shut, *Shutdown

*CheckMap

Checks a disc map for consistency.

Syntax

```
*CheckMap [disc_spec]
```

Parameters

disc_spec	the name of the disc or number of the disc drive

Use

*CheckMap checks that the map of an E- or F-format disc (whether floppy or hard) has the correct checksums and is consistent with the directory tree. If only one copy of the map is good, it allows you to rewrite the bad one with the information in the good one.

Example

```
*CheckMap :Mydisc
```

Related commands

*Defect, *Verify

*Compact

Collects together free space on a disc

Syntax

```
*Compact [disc_spec]
```

Parameters

disc_spec	the name of the disc or number of the disc drive

Use

*Compact collects together free space on a disc by moving files. If no argument is given, the *Compact command is carried out on the current disc. *Compact works on either hard or floppy discs.

You cannot add a file to an old map disc (ie an L- or D- format disc, or an old map hard disc) that is larger than the biggest single free space. Because *Compact gathers together free space, the maximum size of file you can fit on the disc will be as high as is possible after you use this command.

The maximum size of file you can add to an E-format disc does not depend on how fragmented the free space is, so there is not the same need to compact them. This command is still useful, as it will attempt to gather together any fragmented files, and generally tidy the disc up.

Example

```
*Compact :0
```

Related commands

*CheckMap, *FileInfo, *Map

*Configure Dir

Sets the configured disc mounting so that discs are mounted at power on.

Syntax

```
*Configure Dir
```

Use

*Configure Dir sets the configured disc mounting so that, for each FileCore-based filing systems that support mounting:

- a disc gets mounted at power on
- the current directory is set to the root directory of the actual mounted disc (eg adfs::SystemDisc.$).

NoDir is the default setting.

This command is in fact provided by the kernel; however, since it is FileCore that looks at the configured value, it is included in this chapter for clarity.

Related commands

*Configure Drive, *Configure NoDir, *Mount

*Configure NoDir

Sets the configured disc mounting so that discs are not mounted at power on.

Syntax

```
*Configure NoDir
```

Use

*Configure NoDir sets the configured disc mounting so that for each FileCore-based filing system that supports mounting:

- nothing gets mounted at power on.
- the current directory is set to the root directory of the configured drive (eg adfs::0.$).

This is the default setting.

This command is in fact provided by the kernel; however, since it is FileCore that looks at the configured value, it is included in this chapter for clarity.

Related commands

*Configure NoDir, *Configure Drive, *Mount

*Defect

Reports what object contains a defect, or (if none) marks the defective part of the disc so it will no longer be used

Syntax

*Defect *disc_spec disc_addr*

Parameters

disc_spec — the name of the disc or number of the disc drive

disc_addr — the hexadecimal disc address where the defect exists, which must be a multiple of 256 – that is, it must end in '00'

Use

*Defect reports what object contains a defect, or (if none) marks the defective part of the disc so it will no longer be used. *Defect is typically used after a disc error has been reported, and the *Verify command has confirmed that the disc has a physical defect, and given its disc address.

If the defect is in an unallocated part of the disc, *Defect will render that part of the disc inaccessible by altering the 'map' of the disc.

If the defect is in an allocated part of the disc, *Defect tells you what object contains the defect, and the offset of the defect within the object. This may enable you to retrieve most of the information held within the object, using suitable software. You must then delete the object from the defective disc. *Defect may also tell you that some other objects must be moved: you should copy these to another disc, and then delete them from the defective disc. Once you have removed all the objects that the *Defect command listed, there is no longer anything allocated to the defective part of the disc; so you can repeat the *Defect command to make it inaccessible.

Sometimes the disc will be too badly damaged for you to successfully delete objects listed by the *Defect command. In such cases the damage cannot be repaired, and you must restore the objects from a recent backup.

Example

```
*Verify mydisc
Disc error 08 at :0/00010400
*Defect mydisc 10400
$.mydir must be moved
.myfile1 has defect at offset 800
.myfile2 must be moved
```

Related commands

*CheckMap, *Verify

*Dismount

Ensures that it is safe to finish using a disc

Syntax

`*Dismount [disc_spec]`

Parameters

disc_spec	the name of the disc or number of the disc drive

Use

*Dismount ensures that it is safe to finish using a disc by closing all its files, unsetting all its directories and libraries, forgetting its disc name (if a floppy disc) and parking its read/write head. If no disc is specified, the current disc is used as the default. *Dismount is useful before removing a particular floppy disc, and is essential if the disc is to taken away and modified on another computer. However, the *Shutdown command is usually to be preferred, especially when switching off the computer.

Example

```
*Dismount
```

Related commands

*Mount, *Shutdown

*Drive

Sets the current drive

Syntax

```
*Drive drive
```

Parameters

drive	the number of the disc drive, from 0 to 7

Use

*Drive sets the current drive if NoDir is set. Otherwise, *Drive has no meaning. The command is provided for compatibility with early versions of ADFS.

Example

```
*Drive 3
```

Related commands

*Dir, *NoDir

*Free

Displays the total free space remaining on a disc

Syntax

```
*Free [disc_spec]
```

Parameters

disc_spec	the name of the disc or number of the disc drive

Use

*Free displays the total free space remaining on a disc. If no disc is specified, the total free space on the current disc is displayed.

Example

```
*Free 0
Bytes free &000C1C00=793600
Bytes used &00006400=25600
```

Related commands

*Map

*Map

Displays a disc's free space map

Syntax

```
*Map [disc_spec]
```

Parameters

disc_spec	the name of the disc or number of the disc drive

Use

*Map displays a disc's free space map. If no disc is specified, the map of the current disc is displayed.

Example

```
*Map :Mydisc
```

Related commands

*Compact, *Free,

*Mount

Prepares a disc for general use

Syntax

```
*Mount [disc_spec]
```

Parameters

disc_spec	the name of the disc or number of the disc drive

Use

*Mount prepares a disc for general use by setting the current directory to its root directory, setting the library directory (if it is currently unset) to $.Library, and unsetting the User Root Directory (URD). If no disc spec is given, the default drive is used. The command is preserved for the sake of compatibility with earlier Acorn operating systems.

Example

```
*Mount :mydisc
```

Related commands

*Dismount

*NameDisc

Changes a disc's name

Syntax

```
*NameDisc disc_spec new_name
```

Parameters

disc_spec	the present name of the disc or number of the disc drive
new_name	the new name of the disc, which may be up to 10 characters long

Use

*NameDisc (or alternatively, *NameDisk) changes a disc's name.

Example

```
*NameDisc :0 DataDisc
```

Related commands

None

*Title

Sets the title of the current directory

Syntax

```
*Title [text]
```

Parameters

text	a text string of up to 19 characters

Use

*Title sets the title of the current directory. Titles take no place in pathnames, and should not be confused with disc names. Spaces are permitted in *Title names.

Titles are output by some * Commands that print headers before the rest of the information they provide: for example *Ex.

This command is not available in versions of RISC OS after 2.0, and you should no longer use it.

Related commands

*Cat, *Ex

*Verify

Checks a disc for readability.

Syntax

```
*Verify [disc_spec]
```

Parameters

disc_spec	the name of the disc or number of the disc drive

Use

*Verify checks that the whole disc is readable, except for sectors that are already known to be defective. The default is the current disc.

Use *Verify to check discs which give errors during writing or reading operations. It can check both floppy discs and hard discs.

*Verify uses a hard disc controller 'primitive' routine which does not attempt retries if a read error occurs. Occasional misreads are not abnormal in hard disc systems, and in normal operation FileCore corrects these by retrying. *Verify may therefore occasionally indicate an error which under normal use would not be encountered. Only if an error is reported consistently at the same sector address should further action be taken.

Example

```
*Verify 4
```

```
*Verify :Mydisc
```

Related commands

*Defect

29 ADFS

Introduction

ADFS is the Advanced Disc Filing System. It is a module that, together with FileSwitch and FileCore, provides a disc-based filing system.

Most of the facilities that you will use with ADFS are in fact provided by FileCore and FileSwitch, and you should read the chapters on those modules in conjunction with this one.

Overview

ADFS is a module that provides the hardware-dependent part of a disc-based filing system. It uses FileCore, and so conforms to the standards for a module that does so; see the chapter entitled *FileCore* for details.

It provides:

- a * Command to select itself (*ADFS)
- a * Command to format discs (*Format)
- various configure options, accessed using *Configure
- four SWIs that give access to corresponding FileCore SWIs
- two further SWIs to set the address of an alternative hard disc controller, and to set the number of retries used for various operations
- the entry points and low-level routines that FileCore needs to access the disc controllers and associated hardware.

Except for the low-level entry points and routines (which are for the use of FileCore only) all of these are described below.

Technical details

Formats

For a full summary of 'perfect' ADFS formats, see from page 3-189 onwards of the chapter entitled *FileCore*.

Disc Drives

For the purposes of formatting, the speed stability of disc drives will be assumed to be 1.5%.

Drives which fit into the following spec will never have a data overrunning:

Variation in speed:	±1.5%
Min. Write to read changeover time:	696 µS (2Meg mode) (43 bytes) 1300 µS (1Meg mode) (40 bytes) (values for one particular drive)
Track length (nominal)	12500 bytes (2Meg mode) 6250 bytes (1Meg mode)
Assuming the drive is always running fast gives an actual workable track length of:	12312 bytes (2Meg mode) 6156 bytes (1Meg mode)

Fit within track lengths

If evaluating the total byte usage of the given formats gives a number less than the minimum track length, then that format fits and will be reliable.

Here are the parameters of the parts of a track:

(soft) Index mark	96 bytes
Minimum gap 4	30 bytes (2Meg mode) 40 bytes (1Meg mode)
Sector overhead	62 bytes (includes gap 2 and pre-ambles):

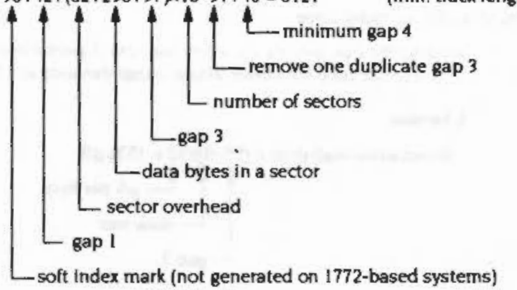
Bytes	Use
12	00-bytes (preamble)
3	A1-bytes
1	FE-ID of address field
1	Track
1	Side
1	Sector

1	Length
1	CRC 1
1	CRC 2
22	4e-gap 2
12	00-bytes (preamble)
3	A1-bytes
1	FB-ID of data field
n	(data – not included in sector overhead)
1	CRC 1
1	CRC 2
62	**Total**

Plugging the numbers in gives:

L formats

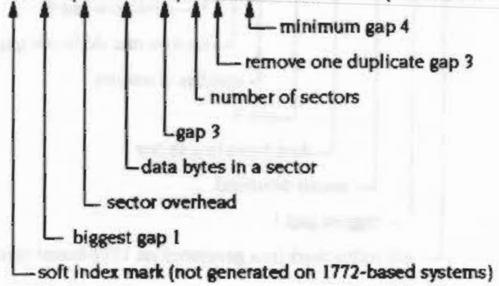
96+42+(62+256+57)×16–57+40 = 6121 (min. track length = 6156)


minimum gap 4
remove one duplicate gap 3
number of sectors
gap 3
data bytes in a sector
sector overhead
gap 1
soft index mark (not generated on 1772-based systems)

D and E formats

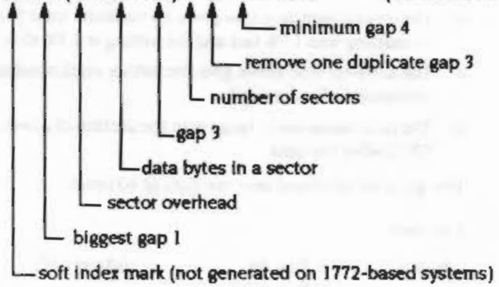
1772-based system without Index mark:

0 +303+(62+1024+90)×5–90+40 = 6133 (minimum track length = 6156)



710/711-based system with index mark (gap 1 forced to 50 bytes by the 710/711):

96+50+(62+1024+90)×5–90+40 = 5976 (minimum track length = 6156)



F format

96+50+(62+1024+90)×10–90+30 = 11846 (min. track length = 12312)

minimum gap 4

remove one duplicate gap 3

number of sectors

gap 3

data bytes in a sector

sector overhead

biggest gap 1

soft index mark (not generated on 1772-based systems)

Minimum Gap3 size

In checking the gap 3 value assuming worst case drive speed variation:

- The drive speed variation gives 3% variation total (assuming the drive used for formatting was 1.5% fast and for writing is 1.5% slow).
- The write-to-read times give the further slack needed which gives the minimum value for gap3.
- The total variation in bytes is in the section of a sector from gap2 to the end of CRC2 after the data.

This gives an overhead over the data of 40 bytes.

L format

Min. gap 3 = 9 + 40 = 49 (actually 57)

write-to-read time

data size (256+40) × 3%

D and E formats

Min. gap 3 = 32 + 40 = 72 (actually 90)

write-to-read time

data size (1024+40) × 3%

F format

Min. gap 3 = 32 + 43 = 75 (actually 90)

write-to-read time

data size (1024+40) × 3%

Worst write to read time

Working the calculations the other way round gives the worst case values for the write-to-read time for a drive whose speed variation is 1.5%:

L format

Worst write-read-time = (57–9)×32 = 1536 µS

µS per byte

data size

gap 3

D and E formats

Worst write-read-time = (90–32)×32 = 1856 µS

µS per byte

data size

gap 3

F format

$$\text{Worst write-read-time} = (\underbrace{90}_{\text{gap 3}}-\underbrace{32}_{\text{data size}})\times\underbrace{16}_{\mu\text{S per byte}} = 928\ \mu\text{S}$$

Hardware Limits

Controllers

These are the limit parameters for the two floppy controllers ADFS supports:

Controller	**1772**	**710/711**
Sectors per track, low	1	0
Sectors per track, high	240	255
Track, low	0	0
Track, high	240	255
Log_2 (sector length), low	7	7
Log_2 sector length, high	10	14
Sector number, low (formatting)	0	0
Sector number, high (formatting)	255	255
Format fill values always allowed	00-&F4, &FF	00-&FF
Formatting with ID mark	optional	forced
Gap3 maximum length (formatting)	track length	255

Recommended formats

(These values are extracted from the 1772 data sheet)

Dens	**gap1**	**gap3**	**~gap4**
FM	≥16	≥11	≥16
MFM	≥32	≥24	≥16

Evaluation of 'does it fit' is:

Low track length – gap1 + gap3 – (secsize + SecOvrhead + gap3)×secs ≥ min. gap4

If 'no', does it fit using minimum gap1 and minimum gap3?

- If so, divide slack amongst gaps (including gap4); else return error

Does the side/side skew invalidate gap4?

- If so, shorten it to minimum gap4

Floppy drive types supported by 710/711 driver

The range of floppy drives supported by the 82C710/82C711 driver is considerably wider than that supported by older drivers. In general **any** PC/XT/AT compatible 3½"/5¼" 40/80 track drive can be used. The following minimal requirements will ensure optimal performance:

- Disc changed support should be available on pin 34, and should be resettable with a step pulse.
- The drive should mask index pulses when selected but without a disc present.
- The drive should not mask index pulses whilst step pulses are being issued.
- The drive should support a 'density in' signal (from FDC) that is active high for high density (≥500Kbps).
- The drive should supply media ID signals that indicate the greatest density supported by the current drive/media.
- Drives 0/1 should be ready to use within 500mS of motor startup.
- Drives 2/3 should be ready to use within 1000mS of motor startup.

Motor on and drive select signals

The following table illustrates the combination of motor on and drive select signals supplied for various drive selections:

Drive Selected	**/DS0**	**/DS1**	**/ME0**	**/ME1**
0	L	H	L	H
1	H	L	H	L
2	H	H	H	L
3	H	H	L	L
None	H	H	H	H

Drives 2 and 3 do not result in any drive select line being asserted, but can be decoded by an external decoder.

Drive interface signal description

To help you understand the floppy disc drive interface, this section discusses further the function and use of each of the interface signals.

General

All interface signals are open-collector, and therefore require a pull-up resistor of nominally 1kΩ for 3½" systems or 150Ω in older 5¼" systems. The pull-up should be present in one place only – either on the drive furthest from the controller (for outputs), or on the controller (for inputs).

Due to the nature of open collector signals no damage will occur if several outputs drive one signal; thus it is safe, for instance, to connect 'motor on' to 'Sel2' and force motor on true whenever Sel2 is asserted.

All signals are active (asserted) low, ie active when at 0 Volts. Inputs are only valid when a drive is selected.

Drive Select 0, 1, 2 and 3 – Output

Used to select the drive; only one should be active at any given time. Most 'AT' compatible drives assume only drive select 1 will ever be asserted, since there is a physical twist in the cable to determine the actual drive number.

Motor On – Output

Asserted to turn the drive motor on (and load the head on 5¼" drives). A period of 0.5 seconds (1 second for drives 2 and 3) is allowed before any data transfer occurs to allow the drive motor to come up to speed.

Side1 – Output

Asserted to select the under surface of a disc

Step – Output

Asserted to step the head in the direction given by DirIn. Also used to reset DiscChanged. A period of 15-20 ms is required to allow for head settling after any movement.

DirIn – Output

Asserted to move the head inwards (to the centre) during head movements.

WriteData – Output

Data from the controller to be written to disc.

WriteGate – Output

Qualifies WriteData. Asserted prior to and after WriteData is true to enable recording of the data.

Density – Output

Informs the drive of the current data rate. Asserted for 500Kbps and 1Mbps operations (1.6 and 3.2 Mbyte formats). Normally on pin2, some drives may require an inverted signal if intended for use with PS/2 systems.

Track00 – Input

Asserted by the drive when the head is on track 0.

WriteProtect – Input

Asserted by the drive when the disc is write protected.

ReadData – Input

Data stream read from the disc.

Index – Input

Index pulses are produced every disc revolution (200mS). The 82C710/82C711 driver uses the presence of index pulses to detect a disc in. If a drive does not support "DiscChanged" then in order to function with the 82C710 /82C711 driver it **must** inhibit index pulses with the drive empty; this is the normal situation. Performance is improved if index pulses are not masked during seek or motor startup. Index pulses must be present within 900mS (1400mS for drives 2 and 3) of asserting drive select/motor on, otherwise the drive will be deemed to be empty.

DiscChanged – Input

This signal is normally available on pin34 or pin2 and when asserted indicates that the disc in the selected drive has been changed. Neither the 1772 nor the 82C710/82C711 driver require DiscChanged in order to function, but give better performance if available. The signal must never be asserted if non-functional.

Dependent upon drive type the disc changed signal may either be reset by issuing a step pulse (82C710/82C711 driver) and/or by asserting the disc changed reset signal (1772 driver). If DiscChanged is reset by 'step', the wimp polling period is set to 1 per second; otherwise it is set to 10 times per second.

Ready – Input

Often available on $5^1/_4$" drives, and available from drives for A440/540 series machines on pin34. Asserted when the drive is ready for read/write operations. This feature is **required** by the 1772 driver. If not present, Ready must be tied low for the driver to function.

Disc errors

Disc errors are errors returned by the controller. The following sections list the disc error codes returned for all controllers currently used in RISC OS computers.

1772 (floppy disc) error codes

1772 disc error codes are basically the error codes returned in the status byte of the 1772. These are the status bits in that status byte:

Bit	Name	Meaning
7	FdcMotorOnBit	
6	WProtBit	Write protect (translated to disc write protected error)
5	WFaultBit	Write fault
4	RnfBit	Record not found
3	CrcBit	CRC error
2	LostBit	Lost data
1	Track0Bit	
0	BusyBit	

So, disc error 8 is a CRC error

ST506 (hard disc) error codes

ST506 disc error codes are the error codes returned by the HD63463 (ST506) controller shifted right by 2 bits, which gives:

Value	Name	Meaning
&01	ABT	Command abort has been accepted
&02	IVC	Invalid command
&03	PER	Command parameter error
&04	NIN	Head positioning, disc access, or drive check command before SPC has been issued
&05	RTS	TST command after SPC command
&06	NUS	USELD for a selected drive has not been returned
&07	WFL	Write fault (WFLT) has been detected on the ST506 interface
&08	NRY	Ready signal has been negated
&09	NSC	Seek complete (SCP) wasn't returned before timeout
&0A	ISE	SEK, or disc access command issued during a seek
&0B	INC	Next cylinder address greater than number of cylinders
&0C	ISR	Invalid step rate: highest-speed seek specified in normal seek mode.
&0D	SKE	SEK or disc access command issued to drive with seek error
&0E	OVR	Data overrun (memory slower than drive)
&0F	IPH	Head address greater than number of heads
&10	DEE	Error Correction Code (ECC) detected an error
&11	DCE	CRC error in data area
&12	ECR	ECC corrected an error
&13	DFE	Fatal ECC error in data area
&14	NHT	In CMPD command data mismatched from host and disc
&15	ICE	CRC error in ID field (not generated for ST506)
&16	TOV	ID not found within timeout
&17	NIA	ID area started with an improper address mark
&18	NDA	Missing address mark
&19	NWR	Drive write protected

IDE error codes

IDE disc errors are, where possible, mapped onto a similar error from an ST506 – in which case the name of the ST506 error is shown below. Other IDE disc errors are given error codes outside the range used by the ST506:

Value	Name	Meaning
&02	IVC	command aborted by controller
&07	WFL	write fault
&08	NRY	drive not ready
&09	NSC	track 0 not found
&13	DFE	uncorrected data error
&16	TOV	sector id field not found
&17	NIA	bad block mark detected
&18	NDA	no data address mark
&20		no DRQ when expected
&21		drive busy when commanded
&22		drive busy on command completion
&23		controller did not respond within timeout
&24		unknown code in error register

710/711 (floppy disc) error codes

710/711 disc error codes are the error codes returned by the (functionally equivalent) 82C710 and 82C711 controllers, which gives:

Value	Meaning
&01	Fatal – controller hardware error
&02	Fatal – command timed out, drive problem
&03	Fatal – Track 0 not found, drive problem
&10	Critical – seek fault
&20	Recoverable – non specific command error
&21	Data overrun
&22	Data CRC error
&23	Sector or ID not found
&24	Missing address mark

Service Calls

Service_IdentifyFormat (Service Call &6B)

Identify format

On entry

R0 = pointer to format specification string (null terminated)
R1 = &6B (reason code)

On exit

All registers preserved (if not claimed)

If claimed:
R0 preserved
R1 = 0
R2 = SWI number to call to obtain raw disc format information
R3 = parameter in R3 to use when calling disc format SWI
R4 = SWI number to call to lay down a disc structure
R5 = parameter in R0 to use when calling disc structure SWI

Use

This call is issued by a handler of discs (such as ADFS) to find how to initialise a disc to a specified format. The format specification string is the same as the `format` parameter specified in the *Format command (see page 3-294).

You should claim this call if your module recognises the format specification string as one that you support. If you do not recognise the format – or if you don't support disc formats at all – you should pass the call on with all registers preserved.

For an example of a call used to obtain raw disc format information, see DOSFS_*DiscFormat* (SWI &41AC0) on page 3-316. Similarly, for an example of a call used to lay down a disc structure, see DOSFS_*LayoutStructure* (SWI &41AC1) on page 3-319.

Service_DisplayFormatHelp (Service Call &6C)

Display list of available formats

On entry

R0 = 0
R1 = &6C (reason code)

On exit

If no error occurred whilst displaying the help:
R0, R1 preserved to pass on

If an error occurred whilst displaying the help:
R0 = pointer to error block
R1 = 0 to claim

Use

This service call is issued when the user requests help on the available formats (eg types `*Help Format`). Your module should list the formats it will recognise in response to Service_IdentifyFormat. The list should be displayed one format per line in this format:

format – description

Where *format* is the text as recognised by Service_IdentifyFormat, and *description* is a description of the format. For example:

```
F - 1600K, 77 entry directories, new map, Archimedes ADFS 2.50 and above.

DOS/Q - 1.44M, MS-DOS 3.20, 3.5" high mdensity disc
```

You should display the list using OS_WriteC or a derivative of that (eg OS_Write0, OS_WriteS etc).

SWI calls

ADFS_DiscOp (SWI &40240)

Calls FileCore_DiscOp

On entry

See FileCore_DiscOp (SWI &40540)

On exit

See FileCore_DiscOp (SWI &40540)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_DiscOp (SWI &40540), after first setting R8 to point to the FileCore instantiation private word for ADFS.

This call is functionally identical to FileCore_DiscOp (SWI &40540).

Related SWIs

FileCore_DiscOp (SWI &40540)

Related vectors

None

ADFS_HDC (SWI &40241)

Sets the address of an alternative hard disc controller

On entry

R2 = address of alternative hard disc controller
R3 = address of poll location for IRQ/DRQ
R4 = bits for IRQ/DRQ
R5 = address to enable IRQ/DRQ
R6 = bits to enable IRQ/DRQ

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets up the address of the hard disc controller to be used by the ADFS. For instance, an expansion card can supply an alternative controller to the one normally used.

The polling and interrupt sense is done using:

```
        LDRB    Rn, [poll location]
        TST     Rn, [poll bits]
```

The IRQ/DRQ must be 1 when active.

Related SWIs

None

Related vectors

None

ADFS_Drives (SWI &40242)

Calls FileCore_Drives

On entry

See FileCore_Drives (SWI &40542)

On exit

See FileCore_Drives (SWI &40542)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_Drives (SWI &40542), after first setting R8 to point to the FileCore instantiation private word for ADFS.

This call is functionally identical to FileCore_Drives (SWI &40542).

Related SWIs

FileCore_Drives (SWI &40542)

Related vectors

None

ADFS_FreeSpace (SWI &40243)

Calls FileCore_FreeSpace

On entry

See FileCore_FreeSpace (SWI &40543)

On exit

See FileCore_FreeSpace (SWI &40543)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_FreeSpace (SWI &40543), after first setting R8 to point to the FileCore instantiation private word for ADFS.

This call is functionally identical to FileCore_FreeSpace (SWI &40543).

Related SWIs

FileCore_FreeSpace (SWI &40543)

Related vectors

None

ADFS_Retries (SWI &40244)

Sets the number of retries used for various operations

On entry

R0 = mask of bits to change
R1 = new values of bits to change

On exit

R0 preserved
R1 = R0 AND entry value of R1
R2 = old value of retry word
R3 = new value of retry word

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets the number of retries used by writing to the retry word. The format of this word is:

Byte	**Number of retries for**
0	hard disc read/write sector
1	floppy disc read/write sector
2	floppy disc mount (per copy of the disc map)
3	verify after *Format, before sector is considered a defect

The new value is calculated as follows:

(old value AND NOT R0) EOR (R1 AND R0)

Related SWIs

None

Related vectors

None

ADFS_DescribeDisc (SWI &40245)

Calls FileCore_DescribeDisc

On entry

See FileCore_DescribeDisc (SWI &40545)

On exit

See FileCore_DescribeDisc (SWI &40545)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_DescribeDisc (SWI &40545), after first setting R8 to point to the FileCore instantiation private word for ADFS.

This call is functionally identical to FileCore_DescribeDisc (SWI &40545).

Related SWIs

FileCore_DescribeDisc (SWI &40545)

Related vectors

None

ADFS_VetFormat (SWI &40246)

Vets a disc format structure for achievability with the available hardware

On entry

R0 = pointer to disc format structure to be vetted
R1 = parameter previously passed by ADFS in R2 to *Image*FS_DiscFormat
(ie disc number)

On exit

R0, R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call vets the given disc format structure for achievability with the available hardware. ADFS updates the disc format structure with values that it can actually achieve with the hardware available. For example the only fill byte value available when formatting might be 0, but the requested value may be &4E, hence 0 would be filled in as the fill byte value.

If ADFS cannot sensibly downgrade the parameters given in the disc format structure, it will generate an error.

This call is typically made by FileCore or by the image filing system *Image*FS, in response to ADFS calling FileCore_DiscFormat (page 3-223) or *Image*FS_DiscFormat (eg DOSFS_DiscFormat (SWI &41AC0) on page 3-316) respectively.

This call is not available under RISC OS 2.

The value in R1 is used to pass enough information on the hardware on which the format is to take place for the disc format structure to be vetted. ADFS uses the disc number for this; other handlers of discs may pass different information if they implement a VetFormat SWI.

Related SWIs

None

Related vectors

None

ADFS_FlpProcessDCB (SWI &40247)

For internal use only

Use

This call is for internal use only. It is not available under RISC OS 2.

ADFS_ControllerType (SWI &40248)

Returns the controller type of a disc

On entry

R0 = drive number (0 - 7)

On exit

R0 = controller type
0 ⇒ disc not present
1 ⇒ 1772
2 ⇒ 710/711
3 ⇒ ST506
4 ⇒ IDE

Flags corrupted

Interrupts

Interrupt status is undefined `
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the controller type of the given disc.

This call is not available under RISC OS 2.

Related SWIs

None

Related vectors

None

ADFS_PowerControl (SWI &40249)

Controls the power-saving features of the ADFS system

On entry

R0 = reason code
0 ⇒ read drive spin status
1 ⇒ set drive autospindown
2 ⇒ control drive spin directly without affecting autospindown
R1 = drive
R2 = drive autospindown, if R0 = 1:
= 0 ⇒ disable autospindown and spinup drive
≠ 0 ⇒ set autospindown to (R2 × 5) seconds
or action to take, if R0 = 2:
= 0 ⇒ spin down immediately
≠ 0 ⇒ spin up immediately

On exit

R2 = drive spin status, if R0 = 0 on entry:
= 0 ⇒ drive is not spinning
≠ 0 ⇒ drive is spinning
R3 = previous value for drive autospindown, if R0 = 1 on entry

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call controls the power-saving features of the ADFS system.

It can be dangerous to use this call on drives that do not fully support drive spin control. The controllers on at least two drives tested hang up when autospindown is enabled; a reset does not recover the situation, although a power-on reset does.

This call is not available under RISC OS 2.

Related SWIs

None

Related vectors

None

ADFS_SetIDEController (SWI &4024A)

Gives the IDE driver the details of an alternative controller

On entry

R2 = pointer to IDE controller
R3 = pointer to interrupt status of controller
R4 = AND with status, NE ⇒ IRQ
R5 = pointer to interrupt mask
R6 = OR into mask enables IRQ
R7 = pointer to data read routine (0 for default)
R8 = pointer to data write routine (0 for default)
R12 = pointer to static workspace

On exit

All registers preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call gives the IDE driver the details of an alternative controller.

This call is not available under RISC OS 2.

Related SWIs

None

Related vectors

None

ADFS_IDEUserOp (SWI &4024B)

Direct user interface for low-level IDE commands

On entry

R0 bit 0 set ⇒ reset controller, clear ⇒ process command
bits 24 - 25 = transfer direction:
00 ⇒ no transfer
01 ⇒ read (ie bit 24 set)
10 ⇒ write (ie bit 25 set)
11 reserved
R2 = pointer to parameter block for command and results
R3 = pointer to buffer
R4 = length to transfer
R5 = timeout in centiseconds (0 ⇒ use default)
R12 = pointer to static workspace

On exit

R0 = command status (0 or a disc error number)
R2 preserved
R3, R4 updated
R5 corrupted

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call provides the direct user interface for low-level IDE commands. It must not be called in background.

This call is not available under RISC OS 2.

Related SWIs

None

Related vectors

None

ADFS_ECCSAndRetries (SWI &40250)

For internal use only

Use

This call is for internal use only. It is not available under RISC OS 2.

* Commands

*ADFS

Selects the Advanced Disc Filing System as the current filing system

Syntax

```
*ADFS
```

Parameters

None

Use

*ADFS selects the Advanced Disc Filing System as the filing system for subsequent operations. Remember that it is not necessary to switch filing systems if you use the full pathnames of objects. For example, you can refer to NetFS objects (on a file server, say) when ADFS is the current filing system.

Example

```
*ADFS
```

Related commands

*Net, *RAM, *ResourceFS

*Configure ADFSbuffers

Sets the configured number of ADFS file buffers.

Syntax

```
*Configure ADFSbuffers n
```

Parameters

n number of buffers

Use

*Configure ADFSbuffers sets the configured number of 1 Kbyte file buffers reserved for ADFS in order to speed up operations on open files. A value of 1 sets a default value appropriate to the computer's RAM size; a value of 0 disables fast buffering on open files.

Example

```
*Configure ADFSbuffers 8
```

*Configure ADFSDirCache

Sets the configured amount of memory reserved for the directory cache

Syntax

```
*Configure ADFSDirCache size[K]
```

Parameters

`size` kilobytes of memory reserved

Use

*Configure ADFSDirCache sets the configured amount of memory reserved for the directory cache. Directories are stored in the cache to save reading them from the disc; this speeds up disc operations, and reduces disc wear. A value of 0 sets a default value appropriate to the computer's RAM size.

Example

```
*Configure ADFSDirCache 16K
```

*Configure Drive

Sets the configured number of the drive that is selected at power on

Syntax

```
*Configure Drive n
```

Parameters

n	drive number

Use

*Configure Drive sets the configured number of the drive that is selected at power on. 0–3 correspond to floppy disc drives; 4–7 correspond to hard disc drives. Since most Acorn computers have only one floppy disc drive and no more than one hard disc drive, the most common values are 0 or 4.

Example

```
*Configure Drive 0
```

Related commands

*Configure Floppies, *Configure HardDiscs, *Configure FileSystem

*Configure Floppies

Sets the configured number of floppy disc drives recognised at power on

Syntax

```
*Configure Floppies n
```

Parameters

n	0 to 4

Use

*Configure Floppies sets the configured number of floppy disc drives recognised at power on. The default value is 1.

Example

```
*Configure Floppies 0
```

Related commands

*Configure HardDiscs

*Configure HardDiscs

Sets the configured number of ST506 hard disc drives recognised at power on

Syntax

```
*Configure HardDiscs n
```

Parameters

n	0 to 2

Use

*Configure HardDiscs sets the configured number of ST506 hard disc drives recognised at power on. These disc drives are the standard ones fitted to These disc drives are the standard ones fitted to early models of RISC OS computers (eg the Archimedes 300, 400 and 500 series, and the A3000). More recent models use IDE discs; for such models, you should set the configured number of ST506 drives to zero, and use the *Configure IDEDiscs command to set the number of hard discs.

The default value depends on the model of computer (for example, an Archimedes 305 is not supplied with a hard disc, so the value is 0). Note however that a delete power-on will **not** preserve this default value, but will set it to zero.

Example

```
*Configure HardDiscs 2
```

Related commands

*Configure Floppies, *Configure IDEDiscs

*Configure IDEDiscs

Sets the configured number of IDE hard disc drives recognised at power on

Syntax

```
*Configure IDEDiscs n
```

Parameters

n	0 to 2

Use

*Configure IDEDiscs sets the configured number of IDE hard disc drives recognised at power on. These disc drives are the standard ones fitted to more recent models of RISC OS computers. Early models (eg the Archimedes 300, 400 and 500 series, and the A3000) use ST506 discs; for such models, you should set the configured number of IDE drives to zero, and use the *Configure HardDiscs command to set the number of hard discs.

The default value depends on the model of computer. Note however that a delete power-on will **not** preserve this default value, but will set it to zero.

Example

```
*Configure IDEDiscs 2
```

Related commands

*Configure Floppies, *Configure HardDiscs

*Configure Step

Sets the configured step rate of one or all floppy disc drives.

Syntax

```
*Configure Step n [drive]
```

Parameters

n	step time in milliseconds
drive	drive number (0 to 3)

Use

*Configure Step sets the configured step rate of one or all floppy disc drives to *n*, the step time in milliseconds. If the drive parameter is omitted, the step rate is set for all floppy disc drives. This command should only be used with non-Acorn disc drives.

The setting of this value affects disc performance. The optimum setting will vary, and is not necessarily the shortest step time. The default value is 3 milliseconds. It is possible to set values of 2, 3, 6 and 12 milliseconds: if other numbers are supplied, the request will be rounded up to the nearest step available.

Limitations of 710/711 controllers

Due to limitations in the 710/711 controllers it is not always possible to set exactly the step rate configured. The following table shows the configured and actual rates used for various densities:

	Actual 710/711 step rate (ms)				
Configured step rate	**Single**	**Double**	**Double+**	**Quad**	**Octal**
2	2	2	1.7	2	2
3	4	4	3.3	3	3
6	6	6	6.7	6	6
12	26	26	25.0	12	8

In single and double density modes, selection of the 12mS step rate actually results in a 26mS rate being used; this is intentional to support older 40/80 track 5$^1/_4$" discs. At octal density it is not possible to step at 12mS; this is a limitation of the hardware, but should not cause problems since drives capable of supporting octal density can normally be stepped at 2 or 3 ms rates.

The limitations are because the step rates provided by the 710/711 controllers depend on the data clock rate selected. Before every command ADFS calls a routine to check the selected clock rate against the selected data rate and the configured step rate, and hence to determine whether the step rate needs first to be altered.

Example

```
*Configure Step 3
```

*Format

Prepares a new floppy disc for use, or erases a used disc for re-use

Syntax

```
*Format drive [format [disc_name]] [Y]
```

Parameters

drive — the number of the disc drive, from 0 to 3

format — the type of format required, selected from:

F	1.6M	RISC OS 3	77-entry directories, new map
E	800K	RISC OS	77-entry directories, new map
D	800K	Arthur 1.2	77-entry directories, old map
L	640K	all ADFS	47-entry directories, old map
Q	1.44M	MS-DOS 3.20	double sided
M	720K	MS-DOS 3.20	double sided
H	1.2M	MS-DOS 3	double sided
N	360K	MS-DOS 2, 3	double sided
P	180K	MS-DOS 2, 3	single sided
T	320K	MS-DOS 1, 2, 3	double sided
U	160K	MS-DOS 1, 2, 3	single sided
A	720K	Atari	double sided
B	360K	Atari	single sided

disc_name — the name to be given to the disc

Y — no prompt for confirmation

Use

*Format prepares a new floppy disc for use, or erases a used disc for re-use.

Early models of RISC OS computers (eg the Archimedes 300, 400 and 500 series, and the A3000) do not have the disc drives and controllers necessary to use H, Q and F formats. RISC OS 2.0 only supports L, D and E formats.

The default is to use F-format if possible; otherwise E-format is used. These formats offer improved handling of file fragmentation on the disc and therefore do not need to be periodically compacted (see the *Compact command).

Example

`*Format 0`	*Formats to default format*
`*Format 0 L`	*Formats the disc in drive 0 for use with ADFS on the BBC Master range of computers*

Related commands

*Compact

30 RamFS

Introduction

RamFS is the RAM Filing System. It is a module that, together with FileSwitch and FileCore, provides a RAM-based filing system.

Most of the facilities that you will use with RamFS are in fact provided by FileCore and FileSwitch, and you should read the chapters on those modules in conjunction with this one.

Overview

RamFS is a module that provides the hardware-dependent part of a RAM-based filing system. It uses FileCore, and so conforms to the standards for a module that does so; see the chapter entitled *FileCore* for details.

It provides:

- a * Command to select itself (*RamFS)
- four SWIs that give access to corresponding FileCore SWIs
- the entry points and low-level routines that FileCore needs to access the RAM-based filing system.

Except for the low-level entry points and routines (which are for the use of FileCore only) all of these are described below.

SWI calls

RamFS_DiscOp (SWI &40780)

Calls FileCore_DiscOp

On entry

See FileCore_DiscOp (SWI &40540)

On exit

See FileCore_DiscOp (SWI &40540)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_DiscOp (SWI &40540), after first setting R8 to point to the FileCore instantiation private word for RamFS.

This call is functionally identical to FileCore_DiscOp (SWI &40540).

Related SWIs

FileCore_DiscOp (SWI &40540)

Related vectors

None

RamFS_Drives (SWI &40782)

Calls FileCore_Drives

On entry

See FileCore_Drives (SWI &40542)

On exit

See FileCore_Drives (SWI &40542)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_Drives (SWI &40542), after first setting R8 to point to the FileCore instantiation private word for RamFS.

This call is functionally identical to FileCore_Drives (SWI &40542).

Related SWIs

FileCore_Drives (SWI &40542)

Related vectors

None

RamFS_FreeSpace (SWI &40783)

Calls FileCore_FreeSpace

On entry

See FileCore_FreeSpace (SWI &40543)

On exit

See FileCore_FreeSpace (SWI &40543)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_FreeSpace (SWI &40543), after first setting R8 to point to the FileCore instantiation private word for RamFS.

This call is functionally identical to FileCore_FreeSpace (SWI &40543).

Related SWIs

FileCore_FreeSpace (SWI &40543)

Related vectors

None

RamFS_DescribeDisc (SWI &40785)

Calls FileCore_DescribeDisc

On entry

See FileCore_DescribeDisc (SWI &40545)

On exit

See FileCore_DescribeDisc (SWI &40545)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_DescribeDisc (SWI &40545), after first setting R8 to point to the FileCore instantiation private word for RamFS.

This call is functionally identical to FileCore_DescribeDisc (SWI &40545).

Related SWIs

FileCore_DescribeDisc (SWI &40545)

Related vectors

None

* Commands

*Configure RamFsSize

Sets the configured amount of memory reserved for the RAM filing system

Syntax

```
*Configure RamFSSize mK|n
```

Parameters

mK	number of kilobytes of memory reserved
n	number of pages of memory reserved; n ≤ 127

Use

*Configure RamFsSize sets the configured amount of memory reserved for the RAM Filing System to use (when the RAMFS module is present) after the next hard reset. The default value is 0, which disables the RAM filing system.

Example

```
*Configure RamFSSize 128K
```

Related commands

None

Related SWIs

OS_ReadRAMFsLimits (SWI &4A), OS_ChangeDynamicArea (SWI &2A)

Related vectors

None

*Ram

Selects the RAM Filing System as the current filing system

Syntax

```
*Ram
```

Parameters

None

Use

*Ram selects the RAM Filing System as the filing system for subsequent operations. Remember that it is not necessary to switch filing systems if you use the full pathnames of objects. For example, you can refer to NetFS objects (on a file server, say) when RamFS is the current filing system.

Memory must have previously been reserved for the RAM filing system; the simplest ways to do so are to use the command *Configure RamFSSize, or to use the Task Manager from the desktop.

Example

```
*Ram
```

Related commands

*ADFS, *Configure RamFSSize, *Net, *ResourceFS

31 DOSFS

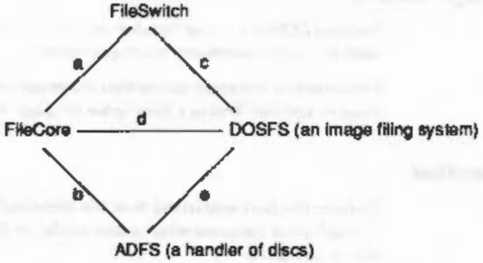
Introduction

DOSFS is an image filing system used to provide DOS disc access from RISC OS.

The description that follows both describes how image filing systems work, and how DOSFS itself works.

Overview

The diagram below shows how DOSFS communicates with other modules in RISC OS 3 to provide the full functionality of an image filing system:



The names identify the component parts. The lines identify links between them:

- Link a is the standard link from FileSwitch to FileCore.
- Link b is the standard link between FileCore and ADFS.
- Link c is a link from FileSwitch to an image filing system – in this case DOSFS.
- Link d is a link between a host filing system – in this case FileCore – and an image filing system – in this case DOSFS.
- Link e is a link between an image filing system – in this case DOSFS – and a handler of discs – in this case ADFS.

Components of an image filing system

There are three links to DOSFS shown in the diagram above. An image filing system can be considered as having three parts, each of which handles one of the links:

- the Image Handler (uses link c)
- the Identifier (uses link d)
- the Formatter (uses link e)

In practice it is best to have these parts in one module as this ensures a complete, working, system is loaded, rather than a partial system. Also, having the parts in one module saves a small quantity of space, due to the sharing of the module overhead, and, possibly, of code.

The Image Handler

This is the most complex component of an image filing system. Its job is to manage files held within an image file (or partition).

The image filing system's image handler communicates only with FileSwitch, accessing images as files. FileSwitch tells the image handler when it has found an image file which is relevant to the image handler. FileSwitch makes requests to the image handler for it to access files and directories held within the image file. The image handler then translates these requests into file access requests which it makes to FileSwitch, which then passes these requests on to the relevant filing system using standard calls. Thus a filing system need not provide any special support for image filing systems to be able to hold image files.

Any image handler must identify itself to FileSwitch as such. This process is similar to that done by a native filing system, but the number of calls the image handler needs to support is fewer, the rest of the work being handled by FileSwitch.

The Identifier

This part of an image filing system is used to identify the format of a disc. It does so by checking an image's format and contents against all the formats of which it knows.

The request to check an image is made by issuing a service call. If the image filing system's identifier recognises the image, it claims the service call; if not it passes it on. The issuer of the service call waits for its return. An unclaimed service call indicates the image wasn't recognised, and so the issuer can complain about the disc being unreadable.

The Formatter

This part of an image filing system is used to help format a disc, which is done by other sections of the system.

Before a disc can be formatted, the user has to specify a format. The image filing system's formatter responds to service calls to help this process. The service calls – one for desktop menu format selection, and one for * Command format selection – are used to identify parameters defining a format. These parameters are in the form of two SWI numbers – both provided by the formatter – with parameters to be passed to them.

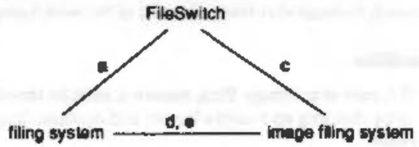
The first of these SWIs is called by the disc handler to negotiate a physical format that is both achievable by the disc handler, and acceptable to the image filing system. Once the disc handler has formatted the disc it then calls the second SWI, with which the formatter lays out the structure of an empty disc.

Points to note

Each module involved in the system only needs to know how to handle a small part of the whole system. For example, the DOSFS image handler doesn't need to know how to identify or format a disc for itself, nor does it need to know how to drive the ADFS disc driver – all it needs to know is how to access a file. Similarly, FileSwitch need make no distinction between discs in a foreign format and image files – they are both presented to FileSwitch as files of a given type.

Once one image filing system is in place, other image filing systems may easily be added to the system by soft loading them.

There is no reason why a single filing system cannot host image filing systems by providing the combined functionality that FileCore and ADFS provide to image filing systems. In such a case, the structure would appear:



Writing image filing systems and host filing systems

If you are writing either an image filing system or a host filing system, you may use this chapter as an example of how an image filing system must behave, and the interfaces it must support; and as pointers to how a host filing system should interact with an image filing system. You should also see

Technical Details

The Image Handler

Because DOSFS's image handler only communicates with FileSwitch, it does not offer any direct interfaces to programmers.

For details of the entry points that an image handler must make available, see the chapter entitled *Writing a filing system* on page 4-1.

The Identifier

Perhaps the best way to see how the identifier works is an example. This follows through what happens when a user clicks on ADFS's floppy disc icon with a DOS disc in the drive.

1. The user clicks on the floppy disc icon.
2. ADFSFiler (the module running the floppy disc icon) sends the Filer (the module running directory viewers) a Filer_OpenDir message for directory adfs::0.$
3. The Filer first checks to see whether it has already got adfs::0.$ open, and, if it hasn't, it creates an internal structure for it and then calls OS_GBPB 10 (read directory entries and information).
4. FileSwitch receives the OS_GBPB 10 with the name 'adfs::0.$' and does an FSEntry_File 5 on ':0.$' to adfs:
5. adfs: uses the FileCore module to process requests from FileSwitch. FileCore, which knows about which discs are in which drives, does not yet know what sort of disc is in drive :0 and so makes a request to the ADFS module to mount the disc.
6. ADFS identifies what physical format the disc has (density, sectors per track, sector numbering etc) and returns to FileCore.
7. FileCore, having had the physical format identified by ADFS, makes a Service_IdentifyDisc quoting the disc record as filled in by ADFS.
8. DOSFS receives the Service_IdentifyDisc, updates the disc record and makes various reads and tries to match the answers with valid DOS disc formats. If a valid format is found it claims the service, if no valid format is found it passes the service on. In this example the service will be claimed and DOSFS will pass back the disc record (which includes the disc name and disc cycle id) and a file type to associate with the disc's contents.

9. FileCore receives the claimed service and records in its own internal drive record that the disc in that drive has the given name and file type. FileCore then returns back to FileSwitch that :0.$ is a file of the type returned to FileCore by DOSFS.
10. FileSwitch notices that :0.$ is a file of a given type and looks up that type in its table of registered image filing systems. FileSwitch opens adfs::0.$ as a file and notifies DOSFS that it has a new image to handle.

    (If the file type isn't found because DOSFS hasn't registered itself with FileSwitch, FileSwitch returns an 'adfs::0.$ is a file' error.)
11. DOSFS receives the notification of an image it has to handle, records internally the FileSwitch handle it was quoted and returns its own handle back at FileSwitch.
12. FileSwitch records against adfs::0.$ the DOSFS handle DOSFS gave it.
13. FileSwitch calls the DOSFS entry point ImageEntry_Func 15 (read directory entries and information), quoting the DOSFS handle for adfs::0.$ and the name of the directory of ''.
14. DOSFS enumerates '' (the root directory of the image) and returns to FileSwitch.
15. FileSwitch filters out any unwanted entries and returns to ADFSFiler.
16. ADFSFiler displays the directory viewer.

Points to note

- The host filing system (ie FileCore) issues the service call Service_IdentifyDisc (see page 3-208) to request that image filing systems identify a disc.
- When an image filing system (eg DOSFS) identifies the disc, it fills in the disc name, disc cycle id and other details in the disc record, and then claims the service call, returning to the host filing system.
- Each image filing system has one (or more) filetypes allocated to it which identifies how the contents of a file of that type should be interpreted as a directory tree with files as leaves.

Disc cycle ids

The host filing system (eg FileCore) keeps two pieces of information about a disc which it uses to identify the same disc at a later time. These are the disc's name and its disc cycle id. The name is the public bit of the identification and is what the user sees; the disc cycle id is used to distinguish between different discs with the same name. Clearly the host filing system needs to be kept abreast of any changes made to the disc's name or disc cycle id.

The only way the disc name can be changed is the FileSwitch call OS_FSControl 50 (see page 3-127), in which case FileSwitch calls an entry point in the host filing system to inform it of the change.

The host filing system can request image filing systems, where appropriate, to update a disc cycle id when the disc is next altered. It does so by calling OS_FSControl 51 (see page 3-128). This is so that another machine isn't misled into believing that an altered disc is unchanged, and – for instance – using invalid cached data. It is the responsibility of all image filing systems to flush new disc cycle ids to media by calling OS_Args 255 (see page 3-55), and to inform their host filing system whenever a disc cycle id has changed **for whatever reason** using OS_Args 8 (see page 3-52).

If there is a change to the disc cycle id and the host filing system is not informed, then it will refuse to match that disc with its internal record, resulting in continuous 'Please insert disc *discname*' messages whenever the user tries to access files on the disc. This is clearly undesirable. So, to summarise:

For a host filing system

- Store away the disc name and disc cycle id to rematch 'new' discs against old ones.
- Respond to the FSEntry_Func 31 and FSEntry_Args 10 entry points to keep the disc name and disc cycle id up to date.
- Call OS_FSControl 51 when a disc might have been removed from the drive since it was last accessed.

For an image filing system

- Call OS_Args 8 whenever you update a disc cycle id.
- Respond to the ImageEntry_Func 32 entry point to keep the disc cycle id up to date.

Storing disc cycle ids

Depending on an image filing system's disc format, there may or may not be room to fit in an explicit disc cycle id somewhere on the disc. For discs where there is room the disc cycle id should simply be incremented with each update. For discs where there isn't room, a disc cycle id may be some derivative of the structures on the disc, such as a checksum of the free space map. Clearly there's not much that can be done in this situation to update the disc cycle id when requested to, but since it is likely to change anyway with each update, this should not be a problem.

The formatter

The formatter is best explained by following through the process. In this example, the host filing system is FileCore/ADFS; other host filing systems should use exactly the same method.

Selecting a format

There are two ways of selecting a format in RISC OS:

1 Specifying the format from the command line.

2 Choosing it from an icon bar menu.

Specifying the format from the command line

Clearly it would be useful for the user to know which formats are available. If the user types `*Help Format`, ADFS displays help on its own formats, and then issues the service call Service_DisplayFormatHelp (see page 3-266). This is passed round all image filing systems, each of which adds its own help text to that already displayed.

To format a disc from the command line, the user calls ADFS's *Format command:

```
*Format drive [format [disc_name]] [Y]
```

ADFS then issues the service call Service_IdentifyFormat (see page 3-265), which passes the *format* around image filing systems. If an image filing system recognises the format, it claims the call. It also returns four values:

- The number of a SWI it provides that will specify the physical format of the disc. For DOSFS, this SWI is DOSFS_DiscFormat; other image filing systems should use the same naming scheme.
- A parameter to pass to that SWI, typically used to identify the format.
- The number of a SWI it provides that will layout the logical structure of an empty disc onto an image file (which may be an entire disc). For DOSFS, this SWI is DOSFS_LayoutStructure; other image filing systems should use the same naming scheme.
- A parameter to pass to that SWI, typically used to specify the image file.

Choosing the format from an icon bar menu

To format a disc from the desktop, the user chooses a format from the **Format** submenu of the ADFSFiler's floppy disc icon bar menu. The ADFSFiler issues the service call Service_EnumerateFormats (see page 3-470). This is passed round all image filing systems, each of which adds its available formats to a linked list of blocks. Each block specifies a single format, and contains its menu text, its help text, and some flags. These entries are used to display the menu, and to provide help on it. But each block also contains the same four values as are returned by Service_IdentifyFormat, thus once a format has been chosen, ADFSFiler can then make them available to ADFS for the next stage of the process.

Whichever way the format has been selected, the rest of the process is identical. **We shall assume that a DOSFS format has been selected**, but the process ought to be identical for other image filing systems.

Negotiating a physical format

Once a DOSFS format has been selected, ADFS calls DOSFS_DiscFormat (see page 3-316), the number of which was obtained from Service_IdentifyFormat, or from Service_EnumerateFormats. In doing so, it passes DOSFS two values:

- The number of a SWI it provides that will vet the disc format for achievability with the available hardware. For ADFS, this SWI is ADFS_VetFormat; other handlers of discs should use the same naming scheme.
- A parameter to pass to that SWI, typically used to identify the drive.

DOSFS fills in a disc format structure with the 'perfect' parameters for the specified format, taking no account of the abilities of the available hardware that will have to perform the format. Once filled in, DOSFS calls ADFS_VetFormat (see page 3-275) to check the format structure for achievability on the available hardware. ADFS may generate an error if the format differs widely from what can be achieved; alternatively it may alter the format structure to the closest match that can be achieved. ADFS_VetFormat then returns to DOSFS, which checks whether the format block – as updated – is still an adequate match for the desired format. If it is, DOSFS_DiscFormat finally returns to ADFS; otherwise it generates an error.

We recommend that image filing systems and handlers of discs only go through one cycle of vetting, as otherwise an infinite loop may ensue.

Formatting the disc

ADFS now has a disc format structure that contains parameters that are both achievable, and satisfactory to DOSFS.

ADFS physically formats and verifies the disc, either by using the *Format command, or by !Format. Both methods use ADFS_DiscOp (see page 3-267) to write and verify tracks. A bad block list is constructed.

The disc then gets opened as a FileSwitch file by whatever is organising the format (*Format or !Format).

Laying out the logical structure

Finally, ADFS calls DOSFS_LayoutStructure to layout the logical structure of an empty disc onto the image file opened by FileSwitch – which is, in fact, the whole disc.

Notes

You can also use DOSFS_LayoutStructure to layout a partition in an image file that is only part of a disc.

Much of the information supplied and managed by one module and used by another is quite long. Because of this, an RMTidy operation is very likely to break the formatting subsystem.

SWI numbers in the formatting subsystem may be passed in either X or non-X form, and the receiver should make no assumption about which form it has been given.

Summary of responsibilities

FileSwitch is responsible for:

- noticing when an image file needs to be opened
- opening it and redirecting the user's request to the relevant image filing system.

FileCore is responsible for:

- organising the identification of a disc whose logical structure is, as yet, unidentified
- faking the entire contents of that disc to be a file of the required type – if an image filing system recognises it – and storing the name of that disc against it
- identifying its own discs and managing the logical structure of them.

ADFS is responsible for:

- identifying the physical format of a disc
- laying down a physical format on a disc
- reading and writing to a disc
- verifying a disc
- organising the formatting and verifying of a disc from the command line.

An image filing system (eg DOSFS) is responsible for:

- managing the logical structure of an image file given its file handle
- identifying a particular disc as being one of its own when requested to do so
- specifying lists of its own formats for the ADFSFiler menu
- identifying a command line format identifier as one of its own
- constructing a physical format description record for one of its own formats
- laying down a logical structure into a file for one of its own formats.

ADFSFiler is responsible for:

- organising the menu selection of a disc format and organising a format to that specification
- organising the verification of a disc to a given specification.

SWI Calls

DOSFS_DiscFormat (SWI &41AC0)

Fills in a disc format structure with parameters for the specified format

On entry

R0 = pointer to disc format structure to be filled in
R1 = SWI number to call to vet disc format (eg ADFS_VetFormat)
R2 = parameter in R1 to use when calling vetting SWI
R3 = format specifier

On exit

R0 - R3 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call fills in the disc format structure pointed to by R0 with the 'perfect' parameters for the specified format, taking no account of the abilities of the available hardware that will have to perform the format. Once filled in, this SWI calls the vetting SWI to check the format structure for achievability on the available hardware. The vetting SWI may generate an error if the format differs widely from what can be achieved; alternatively it may alter the format structure to the closest match that can be achieved. The vetting SWI then returns to this SWI, which checks whether the format block – as updated by the vetting SWI – is still an adequate match for the desired format. If it is, this SWI returns to its caller; otherwise it generates an error.

The following format specifiers are recognised:

Value	Meaning			
0	Q	1.44M	MS-DOS 3.20	double sided
1	M	720K	MS-DOS 3.20	double sided
2	H	1.2M	MS-DOS 3	double sided
3	N	360K	MS-DOS 2, 3	double sided
4	P	180K	MS-DOS 2, 3	single sided
5	T	320K	MS-DOS 1, 2, 3	double sided
6	U	160K	MS-DOS 1, 2, 3	single sided
7	A	720K	Atari	double sided
8	B	360K	Atari	single sided

The returned disc format structure contains the following information:

Offset	Length	Meaning		
0	4	Sector size in bytes (which will be a multiple of 128)		
4	4	Gap1 side 0		
8	4	Gap1 side 1		
12	4	Gap3		
16	1	Sectors per track		
17	1	Density:		
		1	single density (125Kbps FM)	
		2	double density (250Kbps FM)	
		3	double+ density (300Kbps FM)	
				(ie higher rotation speed double density)
		4	quad density (500Kbps FM)	
		8	octal density (1000Kbps FM)	
18	1	Options:		
		bit 0	1	index mark required
		bit 1	1	double step
		bits 2-3	0	interleave sides
			1	format side 1 only
			2	format side 2 only
			3	sequence sides
		bits 4-7		reserved – must be 0
19	1	Start sector number on a track		
20	1	Sector interleave		
21	1	Side/side sector skew (signed)		
22	1	Track/track sector skew (signed)		
23	1	Sector fill value		
24	4	Number of tracks to format (ie cylinders/drive: normally 80)		
28	36	Reserved – must be zero		

This structure tells you how to format a disc. Note that it differs from that used in FileCore_DiscOp to actually format a track (see page 3-213). The differences are because the DiscOp structure only specifies the format of a single track.

This call is not available under RISC OS 2.

Related SWIs

ADFS_VetFormat (SWI &40426), FileCore_DiscFormat (SWI &40547)

Related vectors

None

DOSFS_LayoutStructure (SWI &41AC1)

On entry

R0 = structure specifier
R1 = pointer to list of bad blocks
R2 = pointer to disc name (null terminated)
R3 = file handle of image

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call lays out in the specified image all necessary structures to have a valid, empty, disc. It can be used:

- to layout a structure on a blank, formatted disc (in which case the specified image should be the whole disc image)
- to layout a partition in a file on a disc that has already been formatted (for example for the PC emulator).

The following format specifiers are recognised:

Value	Meaning			
0	Q	1.44M	MS-DOS 3.20	double sided
1	M	720K	MS-DOS 3.20	double sided
2	H	1.2M	MS-DOS 3	double sided
3	N	360K	MS-DOS 2, 3	double sided
4	P	180K	MS-DOS 2, 3	single sided
5	T	320K	MS-DOS 1, 2, 3	double sided
6	U	160K	MS-DOS 1, 2, 3	single sided
7	A	720K	Atari	double sided
8	B	360K	Atari	single sided

If the given image format has no option to store a disc name then this parameter should be ignored.

The bad block list should be presented as an array of bad block addresses. Each address is four bytes long. The array is terminated by a –1 entry.

It is assumed that R0 gives enough information for the format - it may be that R0 contains many bit fields or points to a block of information - the choice is up to the image filing system module.

The value in R0 is used to pass enough information to specify the disc structure DOSFS uses … for this; other image filing systems may pass different information (using a pointer if necessary) for their LayoutStructure SWI.

Related SWIs

None

Related vectors

None

* Commands

*CopyBoot

Copies the boot block from one MS-DOS floppy disc over the boot block of another

Syntax

`*CopyBoot source_drive dest_drive`

Parameters

`source_drive`	the number of the source floppy drive (0 to 3)
`dest_drive`	the number of the destination floppy drive (0 to 3)

Use

*CopyBoot copies the boot block from one MS-DOS floppy disc over the boot block of another.

Example

`*CopyBoot 0 0`	*Copies the boot block from one MS-DOS floppy disc to another, using only drive 0. You will be prompted to change discs when necessary.*

Related commands

None

Related SWIs

None

Related vectors

None

*DOSMap

Specifies a mapping between an MS-DOS extension and a RISC OS file type

Syntax

```
*DOSMap [MS-DOS_extension [file_type]]
```

Parameters

`MS-DOS_extension` An MS-DOS file extension of up to three characters

`file_type` a number (in hexadecimal by default) or text description of the file type to be mapped. The command `*Show File$Type*` displays a list of valid file types.

Use

*DOSMap specifies a mapping between an MS-DOS extension and a RISC OS file type. Any MS-DOS file with the given extension will be treated by RISC OS as having the given file type, rather than being of type 'MSDOS'.

If the only parameter given is an MS-DOS extension, then the mapping (if any) for that extension is cancelled. If no parameter is given, then all current mappings are listed.

Example

`*DOSMap TXT Text` *Treat all files with an MS-DOS 'TXT' extension as RISC OS Text files. For example, they will have Text file icons, and load into a text editor when double-clicked on.*

Related commands

None

Related SWIs

None

Related vectors

None

32 NetFS

Introduction

The NetFS is a filing system that allows you to access and use remote file server machines, using Acorn's Econet network. In common with other filing systems it uses the FileSwitch module, and so when you are using the NetFS you can use any of the commands that FileSwitch provides.

The NetFS module takes the commands that you give to it, either directly or via FileSwitch, and converts them to file server commands. These commands are then sent to the file server using the standard protocol of Econet. The file server then acts on the files or directories that it stores.

Much of the above is transparent to the user, and in general to use file servers you do not need to know file server protocols, or how data is sent over the Econet. For advanced work, you can communicate directly with file servers. If you do need to know more about file server and Econet protocols, you should see:

- the chapter entitled *Econet*
- the *Econet Advanced User Guide*, available from your Acorn supplier.

Overview

The NetFS software provides a filing system for RISC OS. To do this it communicates via the Econet with a file server; the file server stores the files and keeps track of them in its directories, as well as providing authenticated access. The NetFS software translates the user's requests that emerge from FileSwitch into one or more file server commands. These commands are then sent to the file server where they act on the files or directories stored there.

The NetFS software is designed to hold information about each file server that it is logged on to and to use this information when communicating with the file server. There are also some extra commands provided by the NetFS software that communicate directly with the file server.

All communication with the file server is done using the interfaces provided by Econet. Basic communication with a file server involves you transmitting a command to it, and then receiving a reply. Either or both of these may contain your data: for instance when you create a directory the name you supply is sent to the file server, where as when you read the name of the current disc that name is sent back to you. Most commands however send things in both directions. The NetFS software knows all the formats and requirements of the file server and presents these to the user, via FileSwitch.

The other commands (those that do not involve files or directories directly) are accessed via star commands. These commands are only available when NetFS is the current filing system.

There are three commands related to access control: *Logon, *Pass, and *Bye. Two commands are to do with selecting file servers: *FS, and *ListFS. The *Free command provides information about the amount of free space remaining on each of the discs of a file server. The two commands *Mount and *SDisc are identical; the former is provided for compatibility with ADFS, the latter for compatibility with existing network software (ANFS and NFS).

Technical Details

Naming

As well as supplying a filing system name as part of a file name (such as 'Net:&.Fred), you can supply as part of the filing system name the name or number of a file server: for example 'Net#253:&.Fred' or 'Net#Maths:Program'. This will cause the file to be found (or saved, or whatever) on the given file server. If a name is quoted, you must currently be logged on to that file server. If a number is given then you must be logged on to the resulting file server; if only part of the number is given then it will be defaulted against the current file server number.

Timeouts

The dynamics of communication are controlled by several timeouts.

The values used by NetFS for the TransmitCount, TransmitDelay, and ReceiveDelay are more fully explained in the chapter entitled *Econet*. These are the values used for all normal communication with the file server.

Before attempting to log on to a file server, NetFS tries the immediate operation MachinePeek to the file server. This operation uses a second set of values: the MachinePeekCount and the MachinePeekDelay. If this operation fails, the error 'Station not present' is generated. The reason for this is that stations must respond to MachinePeek. You can therefore determine quite quickly if the destination machine is actually present on the network, without having to wait the long time required for a normal transmission to timeout and report 'Station not listening'.

The last value used is called the BroadcastDelay; this is the amount of time for which NetFS will wait for a file server to respond to the broadcast for names of file servers. If the named file server has not responded within that time the error 'Station name not found' will be returned.

Direct access to file servers

To provide access to those functions not provided as part of the FileSwitch interface, or as one of the command interfaces provided directly by NetFS, there are a pair of SWI calls.

The first of these (SWI NetFS_DoFSOp) provides communication with the current file server, and the second (SWI NetFS_DoFSOpToGivenFS) to any file server to which the NetFS software is logged on.

- The function (in R0) is an indication to the file server what it should do. You will find documentation of the file server functions in the *Econet Advanced User Guide* (part number 412,019).
- The buffer contains the data to be sent to the file server. Econet's five byte header (Reply port, Function, URD, CSD, CSL) is prepended to the buffer during transmission. When a reception occurs Econet's two byte header is stripped off before the returned data is placed in the buffer.

Service Calls

Service_NetFS (Service Call &55)

Either a *Logon or *Bye has occurred

On entry

R1 = &55 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This call is issued by NetFS to indicate to the NetFiler that things may have changed. For example, a user logged may have on to a server, while temporarily outside the Wimp.

Service_NetFSDying (Service Call &5F)

NetFS is dying

On entry

R1 = &5F (reason code)

On exit

R1 preserved

Use

Issued by NetFS before closedown to allow Broadcast Loader to unhook.

SWI calls

NetFS_ReadFSNumber (SWI &40040)

Returns the full station number of your current file server

On entry

—

On exit

R0 = station number
R1 = net number

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call returns the full station number of your current file server.

Related SWIs

NetFS_SetFSNumber (SWI &40041), NetFS_ReadFSName (SWI &40042)

Related vectors

None

NetFS_SetFSNumber (SWI &40041)

Sets the full station number used as the current file server

On entry

R0 = station number
R1 = net number

On exit

—

Interrupts

Interrupts may be enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the full station number used by NetFS as the current file server.

Related SWIs

NetFS_ReadFSNumber (SWI &40040), NetFS_SetFSName (SWI &40043)

Related vectors

None

NetFS_ReadFSName (SWI &40042)

Reads the name of the your current file server

On entry

R1 = pointer to buffer
R2 = size of buffer in bytes

On exit

R0 = pointer to buffer
R1 = pointer to the terminating null of the string in the buffer
R2 = amount of buffer left, in bytes

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call reads the name of your current file server.

Related SWIs

NetFS_ReadFSNumber (SWI &40040), NetFS_SetFSName (SWI &40043)

Related vectors

None

NetFS_SetFSName (SWI &40043)

Sets by name the file server used as your current one

On entry

R0 = pointer to buffer

On exit

—

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets by name the file server used as your current one.

Related SWIs

NetFS_SetFSNumber (SWI &40041), NetFS_ReadFSName (SWI &40042)

Related vectors

None

NetFS_ReadCurrentContext (SWI &40044)

Unimplemented

On entry

—

On exit

R0 - R2 corrupted

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call is unimplemented, and returns immediately to the caller.

Related SWIs

NetFS_SetCurrentContext (SWI &40045)

Related vectors

None

NetFS_SetCurrentContext (SWI &40045)

Unimplemented

On entry

—

On exit

All registers preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call is unimplemented, and returns immediately to the caller, with all registers preserved.

Related SWIs

NetFS_ReadCurrentContext (SWI &40044)

Related vectors

None

NetFS_ReadFSTimeouts (SWI &40046)

Reads the current values for timeouts used by NetFS

On entry

—

On exit

R0 = transmit count
R1 = transmit delay in centiseconds
R2 = machine peek count
R3 = machine peek delay in centiseconds
R4 = receive delay in centiseconds
R5 = broadcast delay in centiseconds

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call reads the current values for timeouts used by NetFS when communicating with the file server.

Related SWIs

NetFS_SetFSTimeouts (SWI &40047)

Related vectors

None

NetFS_SetFSTimeouts (SWI &40047)

Sets the current values for timeouts used by NetFS

On entry

R0 = transmit count
R1 = transmit delay in centiseconds
R2 = machine peek count
R3 = machine peek delay in centiseconds
R4 = receive delay in centiseconds
R5 = broadcast delay in centiseconds

On exit

—

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call sets the current values for timeouts used by NetFS when communicating with the file server.

Related SWIs

NetFS_ReadFSTimeouts (SWI &40046)

Related vectors

None

NetFS_DoFSOp (SWI &40048)

Commands the current file server to perform an operation

On entry

R0 = file server function
R1 = pointer to buffer
R2 = number of bytes to send to file server from buffer
R3 = size of buffer in bytes

On exit

R0 = return condition given by file server
R3 = number of bytes placed in buffer by file server

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call commands the file server to perform an operation, as specified by the file server function passed in R0. For further details of these functions, the data they need to be passed in the buffer, and the data they return in the buffer, you should see the *Econet Advanced User Guide* or the documentation for your file server.

The buffer must be large enough to hold the data that the file server returns.

Errors returned by the file server are copied into NetFS's workspace and adjusted to be like a normal RISC OS error – R0 points to the error, and the V bit is set. Any further use of NetFS may overwrite this error, so you should copy it into your own workspace before you call NetFS again, either directly or indirectly. (For example, character input or output may call NetFS, as you may be using an exec or spool file.)

Related SWIs

NetFS_DoFSOpToGivenFS (SWI &4004C)

Related vectors

None

NetFS_EnumerateFSList (SWI &40049)

Lists all file servers to which the NetFS software is currently logged on

On entry

R0 = offset of first item to read in file server list
R1 = pointer to buffer
R2 = size of buffer in bytes
R3 = number of file server names to read from list

On exit

R0 = offset of next item to read (–1 if finished)
R3 = number of file server names read

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call lists all the entries in the list of file servers to which the NetFS software is currently logged on. This is essentially the same as the list you would get by using the `*FS` command with no parameters, except that the user IDs are not returned.

The entries are returned as 20 byte blocks in the buffer:

Offset	Contents
0	Station number
1	Network number
2	Zero
3	Disc name
19	Zero

The order of the list is not significant, save that if you are logged on to your current file server it will be returned last.

Related SWIs

NetFS_EnumerateFS (SWI &4004A), NetFS_EnumerateFSContexts (SWI &4004E)

Related vectors

None

NetFS_EnumerateFS (SWI &4004A)

Lists all file servers of which the NetFS software currently knows

On entry

R0 = offset of first item to read in file server list
R1 = pointer to buffer
R2 = size of buffer in bytes
R3 = number of file server names to read from list

On exit

R0 = offset of next item to read (–1 if finished)
R3 = number of file server names read

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call lists all the entries in a list of file servers which the NetFS software holds internally. This list is used by the NetFS software to resolve file server names, and is the same as the list you would get by using the `*ListFS` command.

The entries are returned as 20 byte blocks in the buffer:

Offset	Contents
0	Station number
1	Network number
2	Drive number
3	Disc name
19	Zero

They are returned in alphabetical order.

Related SWIs

NetFS_EnumerateFSList (SWI &40049)

Related vectors

None

NetFS_ConvertDate (SWI &4004B)

Converts a file server time and date to a RISC OS time and date

On entry

R0 = pointer to file server format time and date (5 bytes)
R1 = pointer to 5 byte buffer

On exit

R1 is preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call converts a file server format time and date to a time and date in the internal format used by RISC OS (centiseconds since 00:00:00 on 1/1/1900).

The file server format is:

Byte	Bits	Meaning
0	0 - 4	Day of month (1 - 31)
	5 - 7	High bits of year (offset from 1980, 0 - 127)
1	0 - 3	Month of year (1 - 12)
	4 - 7	Low bits of year (offset from 1980, 0 - 127)
2	0 - 4	Hours (0 - 23)
	5 - 7	Unused
3	0 - 5	Minutes (0 - 59)
	6, 7	Unused
4	0 - 5	Seconds (0 - 59)
	6, 7	Unused

Related SWIs

OS_ConvertStandardDateAndTime (SWI &C0),
OS_ConvertDateAndTime (SWI &C1)

Related vectors

None

NetFS_DoFSOpToGivenFS (SWI &4004C)

Commands a given file server to perform an operation

On entry

R0 = file server function
R1 = pointer to buffer
R2 = number of bytes to send to file server from buffer
R3 = size of buffer in bytes
R4 = station number
R5 = network number

On exit

R0 = return condition given by file server
R3 = number of bytes placed in buffer by file server

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call commands the given file server to perform an operation, as specified by the file server function passed in R0. For further details of these functions, the data they need to be passed in the buffer, and the data they return in the buffer, you should see the *Econet Advanced User Guide* or the documentation for your file server.

The buffer must be large enough to hold the data that the file server returns.

Errors returned by the file server are copied into NetFS's workspace and adjusted to be like a normal RISC OS error – R0 points to the error, and the V bit is set. Any further use of NetFS may overwrite this error, so you should copy it into your own workspace before you call NetFS again, either directly or indirectly. (For example, character input or output may call NetFS, as you may be using an exec or spool file.)

Related SWIs

NetFS_DoFSOp (SWI &40048)

Related vectors

None

NetFS_UpdateFSList (SWI &4004D)

Adds names of discs to the list of names held by NetFS

On entry

R0 = Station number
R1 = Network number

On exit

R0 is corrupted
R1 is corrupted

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call will fetch the names of the discs of the given file server, and add these names to the list of names held internally to NetFS. This call allows software that uses the NetFS_EnumerateFS call to be sure that information on a particular file server is up-to-date (as the NetFiler does when it offers a menu of disc names to choose when opening '$').

If both R0 and R1 are zero then the entire list will be updated.

Related SWIs

NetFS_EnumerateFS, NetFS_EnableCache

Related vectors

None

NetFS_EnumerateFSContexts (SWI &4004E

Lists all the entries in the list of file servers to which NetFS is currently logged on

On entry

R0 = entry point to enumerate from
R1 = pointer to buffer
R2 = number of bytes in the buffer
R3 = number of entries to enumerate

On exit

R0 = entry point to use next time (–1 indicates no more left)
R2 = space remaining in buffer
R3 = number of entries enumerated

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call lists all the entries in the list of file servers to which NetFS is currently logged on, and includes the user id that NetFS logged on with. This is the same as the list you would get by using the *FS command with no parameters.

Entries are returned as 44 byte blocks in the buffer:

Offset	Contents
0	Station number
1	Network number
2	Reserved – must be zero
3	Disc name padded to 16 characters with spaces
19	Zero
20	User name padded to 21 characters with spaces
41	Zero
42	Reserved – must be zero
43	Reserved – must be zero

Related SWIs

NetFS_EnumerateFSList (SWI &40049), NetFS_EnumerateFS (SWI &4004A)

Related vectors

None

NetFS_ReadUserId (SWI &4004F)

Returns the current userid if logged on to the current file server

On entry

R1 = pointer to buffer
R2 = number of bytes in the buffer

On exit

R2 = space remaining in buffer

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call returns the current user id if logged on to the current file server. If not logged on, a null name is written to the buffer (ie a single zero).

Related SWIs

NetFS_ReadFSNumber (SWI &40040), NetFS_ReadFSName (SWI &40042)

Related vectors

None

NetFS_GetObjectUID (SWI &40050)

On entry

R1 = pointer to the object name
R6 = pointer to the special field, or zero if no special field

On exit

R0 = object type
R1 preserved
R2 = object's load address
R3 = object's exec address
R4 = object's length
R5 = object's attributes
R6 = least significant word of UID
R7 = most significant word of UID

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is very similar to FSEntry_File 5 (Read catalogue information) except that R6 and R7 form a 64 bit unique identifier (UID) for the object. This UID is guaranteed to be unique across all file servers on all networks. The UID is composed of information like the file server's network address, the file server's disc on which the object is held, and the location of the object on that disc. By using this call, stations on an Econet can compare UIDs to see if they are accessing the same object.

Related SWIs

OS_File (SWI &08)

Related vectors

None

NetFS_EnableCache (SWI &40051)

Enables a suspended event task

On entry

—

On exit

—

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The list of names and numbers of file servers that NetFS keeps internally to resolve file server names is added to by an event process. These events are caused by reception packets from file servers of the names of discs. During the enumeration of the list this event task is effectively suspended so that the list does not change during the enumeration. Any call to NetFS_EnumerateFS will cause this suspension to take place. To ensure that the list is being updated it is essential that after a complete enumeration this call is made to re-enable the suspended event task.

Related SWIs

NetFS_EnumerateFS (SWI &4004A), NetFS_UpdateFSList (SWI &4004D)

Related vectors

None

* Commands

*Bye

Logs the user off a file server

Syntax

```
*Bye [[:]file_server]
```

Parameters

file_server	the file server name or number – defaults to the current file server

Use

*Bye terminates the use of a file server, closing all open files and directories. If no file server is given, you are logged off the current file server.

Example

```
*Bye 49.254
```

```
*Bye :fs
```

Related commands

*Logon, *Shut, *Shutdown

*Configure FS

Sets the configured default file server for NetFS

Syntax

```
*Configure FS file_server
```

Parameters

file_server	the file server name or number

Use

*Configure FS sets the configured default file server for NetFS, used where none is specified. It is preferable to use the station name, as this is less likely to change. The default value is 0.254.

Example

```
*Configure FS Server1
```

Related commands

*Configure FileSystem, *Configure PS, *I Am, *Logon

*Configure Lib

Sets the configured library selected by NetFS after logon

Syntax

```
*Configure Lib [0 | 1]
```

Parameters

0 or 1

Use

*Configure Lib sets the configured library selected by NetFS after logon.

When NetFS logs on to a file server, the file server searches for $.Library on drives 0 - ***maxdrive*** of the file server, in that order. It passes the first match back to NetFS as the library to be used. If it does not match this directory then it instead passes back $ on the lowest numbered physical disc.

- If 0 is used as the parameter, then NetFS uses the library directory returned by the file server.
- If 1 is used as the parameter, then NetFS searches for $.ArthurLib on drives 0 - ***maxdrive*** of the file server, in that order. The first match is used by NetFS as the library. If it does not find a match, then it uses the library directory returned by the file server.

Example

```
*Configure Lib 0
```

*Free

Displays file server free space

Syntax

```
*Free [:file_server] [user_name]
```

Parameters

file_server	the file server name or number – defaults to the current file server
user_name	as issued by the network manager

Use

*Free displays a user's total free space, as well as the total free space for the disc.

If no file server is given, the current file server is used.

If a user name is given, the free space belonging to that user is displayed. If no user is given, then the current user's free space is displayed.

Example

```
*Free :Business William
Disc name        Drive  Bytes free
                        Bytes used

Business           0     3 438 592
                        30 967 808
----------------------------------
User free space            185 007
```

*FS

Restores the file server's previous context.

Syntax

```
*FS [[:]file_server]
```

Parameters

file_server	the file server name or number – defaults to the current file server

Use

*FS selects the current file server, restoring that file server's context (for example, its current directory). If no argument is supplied, your current file server number, file server name and user name are printed out, followed by the same information for any non-current servers.

Example

```
*FS 49.254
```

```
*FS :myFS
```

```
*FS
 13.224 :Server1 guest
    254 :Server2 mhardy
```

Related commands

*ListFS

*I am

Selects NetFS and logs you on to a file server

Syntax

```
*I am [[:]file_server_number|:file_server_name] user_name [[:Return]password]
```

Parameters

file_server_number	the file server number to log on to
file_server_name	the file server name to log on to
user_name	as issued by the network manager
password	as set by the user

Use

*I am selects NetFS and logs you on to a file server. Your user name and password are checked by the file server against the password file before allowing you access. If you give neither a file server number nor name, then this command logs you on to the current file server.

The file server first searches drives 0 – *maxdrive* for a password file containing a password/user name pair that match those given; if none is found, access to the file server is denied.

The file server then searches for a directory matching the given user name. It starts with the drive where the password match was found, followed by drives 0 – *maxdrive*. It passes the first matching directory back to NetFS. If it does not match the user name then it instead passes back $ on the lowest numbered physical disc. NetFS sets the User Root Directory to the returned directory, and sets the current directory to the User Root Directory.

NetFS also sets the library directory, as described in *Configure Lib.

This command is identical to a *Net command (which selects NetFS as the current filing system) followed by *Logon (see below).

Example

```
*I am :fs guest
```

Related commands

*Logon, *Net

*ListFS

Lists available file servers

Syntax

```
*ListFS
```

Use

*ListFS displays a list of the file servers which NetFS is able to recognise.

Example

```
*ListFS
 1.254  :0  Finance1
 1.254  :1  Finance2
 6.246  :0  Production
```

Related commands

*FS

*Logon

Logs you on to a file server

Syntax

```
*Logon [[:]file_server_number|:file_server_name] user_name [[:Return]password]
```

Parameters

file_server_number	the file server number to log on to
file_server_name	the file server name to log on to
user_name	as issued by the network manager
password	as set by the user

Use

*Logon logs you on to a file server. Your user name and password are checked by the file server against the password file before allowing you access. If you give neither a file server number nor name, then this command logs you on to the current file server.

The file server first searches drives 0 – *maxdrive* for a password file containing a password/user name pair that match those given; if none is found, access to the file server is denied.

The file server then searches for a directory matching the given user name. It starts with the drive where the password match was found, followed by drives 0 – *maxdrive*. It passes the first matching directory back to NetFS. If it does not match the user name then it instead passes back $ on the lowest numbered physical disc. NetFS sets the User Root Directory to the returned directory, and sets the current directory to the User Root Directory.

NetFS also sets the library directory, as described in *Configure Lib.

You must select NetFS before typing *Logon (this is not necessary with the *I am command).

Example

```
*Logon :fs guest
```

Related commands

*I am

*Mount

Selects a disc from the file server

Syntax

```
*Mount [:]disc_spec
```

Parameters

disc_spec	the name of the disc to be mounted

Use

*Mount selects a disc from the file server by setting the current directory, the library directory and the User Root Directory.

The file server searches the drive for a directory matching the given user name. It passes the first matching directory back to NetFS. If it does not match the user name then it instead passes back $. NetFS then sets the User Root Directory to the returned directory of the selected disc, and sets the current directory to the User Root Directory.

NetFS also sets the library directory, as described in *Configure Lib.

You cannot dismount a file server's disc.

*SDisc is a synonym for *Mount.

Example

```
*Mount fs
```

Related commands

*SDisc

*Net

Selects the Network Filing System as the current filing system

Syntax

```
*Net
```

Parameters

None

Use

*Net selects the Network Filing System as the filing system for subsequent operations. Remember that it is not necessary to switch filing systems if you use the full pathnames of objects. For example, you can refer to ADFS objects when NetFS is the current filing system.

Example

```
*Net
```

Related commands

*ADFS, *RAM, *ResourceFS

*Pass

Changes your password on your current fileserver

Syntax

```
*Pass [old_password [new_password]]
```

Parameters

old_password	your existing password (if any)
new_password	the new password (if any) that you wish to assign

Use

*Pass changes your password on your current fileserver, knowledge of which allows unrestricted access to your network files on that server. If you enter the command without parameters, the computer will prompt you to enter your old and new passwords, reflecting each character you type as a hyphen. If you do not have one, or wish to remove the one you have without substituting a new one, press Return at the relevant prompt. A password may not be more than six characters long.

Examples

```
*Pass
```

`Old password: ----`	*User types* **`pail`** *(existing password)*
`New password: ------`	*User types* **`bucket`**
`*Pass bucket`	*User enters command again, this time giving existing password as parameter*
`New password:`	*User presses* **`Return`**, *leaving themself with no password*

*SDisc

Selects a disc from the current file server

Syntax

```
*SDisc [:]disc_spec
```

Parameters

disc_spec	the name of the disc to be mounted

Use

*SDisc selects a disc from the current file server by setting the current directory, the library directory and the User Root Directory.

The file server searches the drive for a directory matching the given user name. It passes the first matching directory back to NetFS. If it does not match the user name then it instead passes back $. NetFS then sets the User Root Directory to the returned directory of the selected disc, and sets the current directory to the User Root Directory.

NetFS also sets the library directory, as described in *Configure Lib.

You cannot dismount a file server's disc.

*Mount is a synonym for *SDisc.

Example

```
*SDisc fs
```

Related commands

*Mount

Example program

The following program fragments are examples of how you might use file server operations by calling NetFS_DoFSOp:

```
ReadFileServerVersion
          MOV      r0, #25                   ; Command
          ADR      r1, Buffer
          MOV      r2, 0                     ; Nothing to send
          MOV      r3, #(?Buffer - 1)        ; Lots to receive
          SWI      XNetFS_DoFSOp
          BVS      Error
          MOV      r0, #0                    ; Terminate string returned
          STRB     r0, [ r1, r3 ]            ; One byte past the return size
          MOV      r0, r1
          SWI      XOS_Write0                ; Print it
          BVS      Error

PrintStationNumberOfUser                     ; User name pointed to by R0
          ADR      r1, Buffer
          MOV      r2, #0                    ; Initial value of index
Loop      LDRB     r3, [ r0 ], #1
          CMP      r3, #" "                  ; Check for termination
          MOVLT    r3, #13                   ; Translate to what the FS wants
          STRB     r3, [ r1, r2 ]            ; Copy into transmit buffer
          ADD      r2, r2, #1                ; Update index, and size to send
          BGT      Loop
          MOV      r0, #24                   ; Command
          MOV      r3, #?Buffer
          SWI      XNetFS_DoFSOp
          BVS      Error
          LDRB     r3, [ r1, #1 ]            ; Pickup station number
          LDRB     r4, [ r1, #2 ]            ; Pickup network number
          STMFD    r13!, { r3, r4}           ; Deposit in stack frame
          MOV      r0, r13                   ; Pointer to value for conversion
          MOV      r2, #?Buffer              ; Destination size
          SWI      XOS_ConvertNetStation
          ADD      r13, r13, #8              ; Dispose stack frame
          SWIVC    XOS_Write0                ; Display output
          SWIVC    XOS_NewLine
          BVS      Error
```

33 NetPrint

Introduction and Overview

NetPrint is a filing system that allows you to access and use remote printer server machines, using Acorn's Econet network. In common with other filing systems it uses the FileSwitch module. When you are using NetPrint you can use many of the commands that FileSwitch provides. Obviously there are some operations (such as those that read stored data) that are not applicable to network printer servers.

The NetPrint module takes the commands that you give to it, either directly or via FileSwitch, and converts them to printer server commands. These commands are then sent to the printer server using the standard protocol of Econet. The printer server then acts on the commands and files that it is sent. It handles their spooling, and manages its (locally) connected printer.

Much of the above is transparent to the user, and in general to use printer servers you do not need to know printer server protocols, or how data is sent over the Econet. If you do need to know more about printer server and Econet protocols, you should see:

- the chapter entitled *Econet*
- the *Econet Advanced User Guide*, available from your Acorn supplier

Technical Details

Naming

The network printing system is actually a filing system, and as such you can use it by giving its name as part of a file name. For example:

```
*Save NetPrint:Fred A000 +14C3
```

However, with current implementations the file name is ignored, and the 'NetPrint:' part is used to send the data to the network printer. As well as save operations, the NetPrint filing system can also open files and take data. This means that the operating system can spool to `NetPrint:`. This is discussed in more detail in the chapter entitled *System devices*.

The current printer server

Whenever you open or save a file onto NetPrint: the current printer server is used. This printer server has a default value which is stored in CMOS RAM, and you can set the current value using a star command. You can also override the current value by supplying the printer server number as part of the file name. For example:

```
NetPrint#234:
```

This example would send the print to the printer server at station 234. As usual you can specify a full network number. For example:

```
Netprint#2.235:
```

Also, since printer servers can be named, you can supply the printer name rather than the number. For example:

```
NetPrint#Epson:
NetPrint#Daisy:
```

Operations supported

The NetPrint filing system supports the OS_File Save operation and the OS_Find OpenOut operation, as well as OS_BPut and OS_GBPB writes (but not backwards).

Linking NetPrint to *FX 5 4 and VDU 2

There are system variables that connect the VDU print streams to files; an example of this is the default value set up by NetPrint upon its initialisation. This is `PrinterType$4`, and its value is `NetPrint:`. You could change this value to indicate a particular printer:

```
NetPrint#Epson:
```

and set up another variable to contain a different value:

```
PrinterType$3 = NetPrint#2.235
```

so that you can swap between printers with a *FX command. For example:

```
*FX 5 4
*FX 5 3
```

Timeouts

The dynamics of communication are controlled by several timeouts.

The values used by NetPrint for the TransmitCount, TransmitDelay, and ReceiveDelay are more fully explained in the chapter entitled *Econet*. These are the values used for all normal communication with the printer server.

Before attempting to connect to a printer server, NetPrint tries the immediate operation MachinePeek to the printer server. This operation uses a second set of values: the MachinePeekCount and the MachinePeekDelay. If this operation fails, the error 'Station not present' is generated. The reason for this is that stations must respond to MachinePeek. You can therefore determine quite quickly if the destination machine is actually present on the network, without having to wait the long time required for a normal transmission to timeout and report 'Station not listening'.

The last value used is called the BroadcastDelay; this is the amount of time for which NetPrint will wait for a printer server to respond to the broadcast with the name of the printer server. If the named printer server has not responded within that time the error 'No free printer server of this type' will be returned.

SWI calls

NetPrint_ReadPSNumber (SWI &40200)

Returns the full station number of your current printer server

On entry

—

On exit

R0 = station number
R1 = net number

Interrupts

Interrupts status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call returns the full station number of your current printer server.

Related SWIs

NetPrint_SetPSNumber (SWI &40201),
NetPrint_ReadPSName (SW &40202)

Related vectors

None

NetPrint_SetPSNumber (SWI &40201)

Sets the full station number used as the current printer server

On entry

R0 = station number
R1 = net number

On exit

—

Interrupts

Interrupts may be enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the full station number used by NetPrint as your current printer server.

Related SWIs

NetPrint_ReadPSNumber (SWI &40200),
NetPrint_SetPSName (SWI &40203)

Related vectors

None

NetPrint_ReadPSName (SWI &40202)

Reads the name of your current printer server

On entry

R1 = pointer to buffer
R2 = size of buffer in bytes

On exit

R0 = pointer to buffer
R1 = pointer to the terminating null of the string in the buffer
R2 = amount of buffer left, in bytes

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call reads the name of your current printer server.

Versions of the NetPrint module before 5.26 return R1 one greater than it should be, and hence R2 one less than it should be.

Related SWIs

NetPrint_ReadPSNumber (SWI &40200),
NetPrint_SetPSName (SWI &40203)

Related vectors

None

NetPrint_SetPSName (SWI &40203)

Sets by name the printer server used as your current one

On entry

R0 = pointer to buffer

On exit

—

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets by name the printer server used as your current one.

Related SWIs

NetPrint_SetPSNumber (SWI &40201),
NetPrint_ReadPSName (SWI &40202)

Related vectors

None

NetPrint_ReadPSTimeouts (SWI &40204)

Reads the current values for timeouts used by NetPrint

On entry

—

On exit

R0 = transmit count
R1 = transmit delay in centiseconds
R2 = machine peek count
R3 = machine peek delay in centiseconds
R4 = receive delay in centiseconds
R5 = broadcast delay in centiseconds

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call reads the current values for timeouts used by NetPrint when communicating with the printer server.

Related SWIs

NetPrint_SetPSTimeouts (SWI &40205)

Related vectors

None

NetPrint_SetPSTimeouts (SWI &40205)

Sets the current values for timeouts used by NetPrint

On entry

R0 = transmit count
R1 = transmit delay in centiseconds
R2 = machine peek count
R3 = machine peek delay in centiseconds
R4 = receive delay in centiseconds
R5 = broadcast delay in centiseconds

On exit

—

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call sets the current values for timeouts used by NetPrint when communicating with the printer server.

Related SWIs

NetPrint_ReadPSTimeouts (SWI &40204)

Related vectors

None

NetPrint_BindPSName (SWI &40206)

On entry

On exit

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call

Related SWIs

NetPrint_ (SWI &4020)

Related vectors

None

NetPrint_ListServers (SWI &40207)

Returns the names of all printer servers

On entry

R0 = format code:

0	names and numbers
1	names only, sorted, no duplicates
2	names, numbers and status

R1 = pointer to buffer
R2 = length of buffer in bytes
R3 = time to take before returning, in centiseconds

On exit

R0 = number of entries returned
R1, R2 preserved
R3 = return code:

0	timed out
1	buffer full

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call returns the names of all printer servers. The format and contents of the returned buffer are determined by the format code passed in R0

R0 = 0: Names and numbers

Offset	Contents
0	station number
1	network number
2	server name, zero terminated

R0 = 1: Names only, sorted by name (case insensitive), no duplicates.

Offset	Contents
0	server name, zero terminated

R0 = 2: Names, numbers and status

Offset	Contents
0	station number
1	network number
2	status
3	station number for status (optional)
4	network number for status (optional)
5	server name, zero terminated

Status values are as follows:

Value	Name	English message(s)
0	Status_Ready	'ready'
1	Status_Busy	'busy with nnn.sss' 'busy'
2	Status_Jammed	'jammed'
6	Status_Offline	'offline'
7	Status_AlreadyOpen	'already in use'

For Status_Busy, the former message is used when the printer server is busy with a single known station: its number follows. The latter message is used when the printer server is busy with an unknown station, or with more than one: in this case the optional status number is set to zero.

Related SWIs

NetPrint_ConvertStatusToString (SWI &40208)

Related vectors

None

NetPrint_ConvertStatusToString (SWI &40208)

Translates a status value returned from NetPrint_ListServers into the local language

On entry

R0 = pointer to a status value byte, followed by two optional bytes containing the station and network number associated with the status
R1 = pointer to buffer to hold message
R2 = length of the buffer in bytes

On exit

R0 = value of R1 on entry
R1= pointer to the terminating zero
R2 = bytes remaining in the buffer after the terminating zero

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call translates a status value returned from NetPrint_ListServers into the local language, copying the resultant message into the specified buffer and terminating it with a zero.

Related SWIs

NetPrint_ListServers (SWI &40207)

Related vectors

None

* Commands

*Configure PS

Sets the configured default network printer server

Syntax

```
*Configure PS printer_server
```

Parameters

printer_server	the name or station number of the printer server

Use

*Configure PS sets the configured default network printer server.

You do not need to be logged on to a file server to use a printer server.

Example

```
*Configure PS Laser1
```

Related commands

*ListPS, *PS, *SetPS

*ListPS

Lists all the currently available printer servers

Syntax

```
*ListPS [-full]
```

Parameters

-full	show status of each printer server

Use

*ListPS lists all the currently available printer servers, optionally showing their status as well. The order in which they are given depends on the order in which the printer servers reply.

Example

```
*ListPS -full
Printer server 'Ebony' (235) is ready

Umber     46.235  ready
Jade      44.235  ready
Mauve     93.235  ready
White     59.235  ready
Coral     32.235  ready
Lime       2.235  ready
```

Related commands

*Configure PS, *PS, *SetPS

*PS

Changes the default printer server

Syntax

```
*PS printer_server
```

Parameters

printer_server	the name or station number of the printer server

Use

*PS changes the default printer server, checking that the new one exists. The new printer server will be used next time you print to the default net printer.

Example

```
*PS 49.254
```

```
*PS myPS
```

Related commands

*Configure PS, *ListPS, *SetPS

*SetPS

Changes the default printer server

Syntax

```
*SetPS printer_server
```

Parameters

printer_server	the name or station number of the printer server.

Use

*SetPS changes the default printer server. This command only changes the stored name or number of the default printer server. No check is made that the printer server exists, or is available, until the next time you print to the default network printer. It is only then that an error might be generated.

Example

```
*SetPS 49.254
```

```
*SetPS myPS
```

Related commands

*Configure PS, *ListPS, *PS

34 PipeFS

Introduction and Overview

PipeFS provides a mechanism for implementing named pipes between tasks, using the *PipeCopy command to move bytes from one pipe to another.

It calls OS_UpCall 6 (see page 1-177) if a pipe being read becomes empty, or if one being written to gets full, and thus cooperates with the Task Window.

It calls OS_UpCall 7 (see page 1-179) if an open pipe is closed or deleted. The Task Window module then traps this and objects (by returning an error) if any of its tasks are currently waiting for the poll word related to that pipe to become non-zero.

This prevents a *Shut command from deleting the workspace which is being accessed by the Task Window, which could potentially cause address exceptions. If the task which called PipeFS is killed by the user, the pipe can be released in a safe manner.

* Commands

*PipeCopy

Copies a file one byte at a time to one or two output files

Syntax

```
*PipeCopy source_file destination_file [destination_file]
```

Parameters

source_file	a valid pathname specifying a file
destination_file	a valid pathname specifying a file

Use

*PipeCopy copies a file one byte at a time to one or two output files. It is provided for use with pipes. You should use *Copy to copy normal files.

(Unlike *Copy, this command does not use the OS_File load and save operations, which are incompatible with the use of pipes.)

Example

```
*PipeCopy Pipe:Input Pipe:Output1 Pipe:Output2
```

Related commands

*Copy

Related SWIs

None

Related vectors

None

35 ResourceFS

Introduction and Overview

This chapter describes the interface to the ResourceFS module, which provides the hooks necessary for modules to include files in the Resources: filing system.

This facility is useful because it allows the resource files associated with a particular module to be included in the same file as the binary image, which helps with release control.

It also has an important application for expansion card modules, since it allows them to `*IconSprites` a sprite file which they put into Resources:. This is important as there is no other way to introduce a sprite into the Wimp's free pool other than from a file.

Another application is for certain resource files to be replaced on a selective basis, which is an additional technique to the path mechanism already in use (e.g. WimpSPath can be set up to reference a resource directory).

Technical Details

Directory structure

In order to avoid possible name clashes, it is important that a well-defined directory structure is adhered to by all concerned. This is:

```
$.Apps.!appname              ; the ROM-resident applications
$.Fonts                      ; the ROM-resident fonts
$.Resources.modulename       ; resources for ADFSFiler etc
$.Resources.appname          ; resources for PrinterDM etc
$.ThirdParty.appname         ; resources for 3rd parties
```

where *appname* is the name of the application concerned, without the '!' on the front (e.g. Draw, Paint, Edit).

The above all indicate directories, which normally contain files called `!Sprites`, `Templates`, `Messages` and so on.

Where third party software is involved, the actual *appname* used must be registered with Acorn, to avoid clashes. See *Appendix H: Registering names* on page 6-473.

Path variables

The Fonts directory contains the ROM-based fonts, and are accessed by the ROMFonts module setting up Font$Path as follows:

```
*SetMacro Font$Path <Font$Prefix>.,Resources:$.Fonts.
```

(It only does this if Font$Path was previously set to '<Font$Prefix>.'.)

All the Desktop modules (ADFSFiler, NetFiler etc) access their resource files (Messages and Templates) via path variables, eg: 'NetFiler:Messages'. On initialisation, they check for the existence of the relevant path variable and set up the appropriate default if it is not defined, eg:

```
*Set NetFiler$Path Resources:$.Resources.NetFiler.
```

You can set up any or all of these path variables to point to your own message files.

Note that the Wimp uses "WindowManager$Path" rather than "Wimp$Path", to allow Wimp$Path to remain separate. Its resources are:

```
Resources:$.Resources.Wimp.Sprites
Resources:$.Resources.Wimp.Sprites23
Resources:$.Resources.Wimp.Templates
```

The Sprites files contain the Wimp's ROM sprite pool, and cannot be redirected (since the Wimp needs direct access to their ROM addresses).

Auto-starting applications

The Apps directory contains the ROM applications, which each have a *!App* directory, and can be started up by '/Resources:$.Apps.!*App*'. The Desktop module will automatically start the applications using such commands, if the corresponding bits in CMOS RAM are set (see the section entitled *Non-volatile memory* (CMOS RAM) on page 1-346), by issuing Wimp_StartTask as appropriate. It does this on *Desktop : after the normal modules have been started, and before any parameters to the *Desktop command have been decoded.

By default, no applications are auto-started.

Note that !Chars and !Magnifier are not auto-started, since they have no iconbar icons of their own; instead they are put onto the iconbar using the *AddTinyDir command.

Note that this auto-starting procedure does not occur if the *Desktop command has a filename parameter, since in this case it is assumed that the Desktop Boot file will start any applications that are required. The configuration options are provided to allow discless operation of the machine.

Storing configuration data

The two applications !Alarm and !PrinterDM both need to store the user's preferred settings somewhere. Normally this is saved inside the application directory, but clearly there are problems if this is in ROM.

The solution is that the actual !Alarm and !PrinterDM applications do not contain the entire application, but simply the !Run files and the initial configuration files. The !Run file then sets up a path variable, consisting of the current value of <Obey$Dir> (ie the application directory itself) and another directory in Resources:$ (eg Resources:$.Resources.Alarm).

If an attempt is made to save configuration data into Resources:, the error message 'Copy !Alarm onto the application disc and run it from there' results. If the application is then copied onto disc and run from there, alarms can be saved in the new application directory on the disc, while the main body of the program is still located in the ROM.

Internationalisation

Because !Alarm uses a path variable to access its resource files, you can put an updated copy of the 'Messages' file into the application directory on disc, and this will take precedence over the version in the ROM directory, which is accessed via the second path element.

Thus the messages file is not normally copied onto the disc when the user copies !Alarm from the Apps directory, but can be included there if required.

Software interface

In order to register a group of files with ResourceFS, a module must have the files included in their image, with appropriate header information, and then call the SWIs ResourceFS_RegisterFiles and ResourceFS_DeregisterFiles to register and deregister this area as appropriate.

Resource file data

The format of the (word-aligned) resource file data is as follows:

Offset	Size	Meaning	
0	4	offset from here to the next file (contiguous), or 0 for end of list (no data follows)	
4	4	load address of file	as returned by OS_File 5
8	4	exec address of file	
12	4	size of file	
16	4	attributes of file	
20	*n*	full filename, excluding 'S.', null terminated	
20+*n*	0 - 3	padded with 0s until word-aligned	
	4	size of file + 4	
	s	file data	
	0 - 3	padded with 0s until word-aligned, followed by more data in the same format	

The resource file data is terminated by a single 0 word.

The resource file data should be contiguous. If this is not possible, then ResourceFS_RegisterFiles must be called once for each of the areas of resource file data to be used (and an equivalent set of ResourceFS_DeRegisterFiles's later on). Note that each area of resource file data must be terminated by a single word containing 0.

There are no directory objects, since the directory structure can be determined from the full filenames supplied.

Note that where name clashes occur, the first occurrence of the filename in the most recently registered area will be used.

Service Calls

Service_ResourceFSStarted (Service Call &59)

Issued by ResourceFS after the file structure inside ResourceFS has changed.

On entry

R0 = &59 (Reason code)

On exit

All registers preserved (do not claim the service)

Use

This service call is issued by ResourceFS to tell any programs relying on ResourceFS files that the structure has changed.

Applications making use of ResourceFS should note that they have to look again to see if things have changed. For example, the Wimp responds to this service call by looking for its default sprite pool again.

Service_ResourceFSDying (Service Call &5A)

ResourceFS is killed

On entry

R0 = &5A (reason code)

On exit

All registers preserved (do not claim the service)

Use

This call is issued by ResourceFS just before it removes itself as a filing system. The expected uses are similar to Service_ResourceFSStarted.

Service_ResourceFSStarting (Service Call &60)

ResourceFS module is reloaded or reinitialised

On entry

R1 = &60 (reason code)
R2 = code address to call
R3 = workspace pointer for ResourceFS module

On exit

All registers preserved (do not claim the service)

Use

When the ResourceFS module is reloaded or reinitialised, it issues this service call so that modules that provide ResourceFS files can put them back into the structure.

Unfortunately the ResourceFS module is not linked into the module chain at this point, so it is not possible to call ResourceFS_RegisterFiles. Instead, the application should execute the following code:

```
STMFD   SP!, {R0, LR}
ADR     R0, ResourceFSfiles          ; R0 -> ResourceFS file structure(page 3-390)
MOV     LR, PC                       ; LR -> return address
MOV     PC, R2                       ; call ResourceFS routine
LDMFD   SP!, (R0, PC)^
```

Note that the value of R3 passed in the service call must be given to the ResourceFS routine intact, so it can find its workspace.

This call is subtly different from SWI ResourceFS_RegisterFiles,in that it will not cause a Service_ResourceFSStarted to be issued. This is because the ResourceFS module waits until all modules have received the Service_ResourceFSStarting before issuing a Service_ResourceFSStarted to let the 'clients' of Resources: know about it.

SWI Calls

ResourceFS_RegisterFiles (SWI &41B40)

On entry

R0 = pointer to resource file data (see page 3-390 for format)

On exit

—

Use

This call should be made by a module adding files to the ResourceFS structure when the module is initialised.

ResourceFS will link the file(s) into its structure, and then issue a Service_ResourceFSStarted (not to be confused with Service_ResourceFSStarting), which tells any programs relying on ResourceFS files that the structure has changed. For example, the Wimp responds to this service call by looking for its default sprite pool again.

ResourceFS_DeregisterFiles (SWI &41B41)

On entry

R0 = pointer to resource file data (see page 3-390 for format)

On exit

—

Use

This call should be made when the area of memory containing the files is about to be deallocated (e.g. when the module containing them is killed).

Note that it is not necessary to call this SWI on receipt of a Service_ResourceFSDying, since the ResourceFS module 'loses' all references to ResourceFS files when it dies anyway.

* Commands

*ResourceFS

Selects the Resource Filing System as the current filing system

Syntax

```
*ResourceFS
```

Parameters

None

Use

*ResourceFS selects the Resource Filing System as the filing system for subsequent operations. Remember that it is not necessary to switch filing systems if you use the full pathnames of objects. For example, you can refer to ADFS objects when ResourceFS is the current filing system.

Example

```
*ResourceFS
```

Related commands

*ADFS, *Net, *RAM

Related SWIs

None

Related vectors

None

36 DeskFS

Introduction

DeskFS is a ROM based filing system that provided system resources for the Desktop in RISC OS 2. It is not available in later versions of RISC OS, and you should not use it.

The Desktop used the system variable Wimp$Path to find these system resources; by default its value was `DeskFS:`. You could change where the Desktop looked for these system resources by changing the value of Wimp$Path.

DeskFS provided a single * Command to select the filing system, described overleaf for reference.

* Commands

*DeskFS

Selects the desktop filing system as the current filing system

Syntax

`*DeskFS`

Parameters

None

Use

*DeskFS selects the desktop filing system as the filing system for subsequent operations. This is a ROM based filing system used to store system resources for the Desktop module, including some useful window template files used by system utilities.

DeskFS files can be catalogued, loaded and opened for input. They are usually accessed through the `DeskFS:` file system prefix. The system variable `Wimp$Path` defaults to `DeskFS:`

This command is not available in versions of RISC OS after 2.0, and you should no longer use it.

Example

`*DeskFS`

Related commands

*Ram, *ADFS, *Net

Related SWIs

Wimp_OpenTemplate, Wimp_LoadTemplate, Wimp_CloseTemplate

Related vectors

None

37 DeviceFS

Introduction and Overview

DeviceFS provides a standardised interface to device drivers within the RISC OS environment. Devices are declared within the system, and are seen as objects within the 'devices:' filing system.

Streams can be opened for input or output (as supported) onto these objects within the directory structure. A device is given the device file type of &FCC. A device adopts the access rights relevant to its input or output capabilities.

A device driver is simply a set of routines that handle the input or output of data. The device can specify if it would like to be buffered, but it will never know if this is the case. Devices have access to the special field passed on opening a stream, this can be used to pass extra information about opening streams and configuration required, for instance a serial device may contain its setup within the special field string.

DeviceFS provides a way of calling devices (DeviceFS_CallDevice) with a reason code and control registers. All devices have to support a set of system specific calls, but have a range of reason codes available for their own use. This could, for example, be used for controlling a scanner.

DeviceFS currently only supports character devices; block devices have yet to be implemented.

Most filing system operations can be performed on objects: for example data transfer operations. However, it is not possible to create objects within the directory structure which are not devices, nor is it possible to delete objects.

Technical Details

Special fields

Special fields within DeviceFS are commonly used to specify parameters to the device, ie what buffers to be used, if the device should be flushing when a stream is closed and so on.

The device can specify a validation string which is used to parse the special field when the stream is being opened. If this is present then DeviceFS will parse the string and return a block of data relating to the strings contents. This data will remain intact until the stream is closed. If no validation string is specified then it is up to the device to take and manage a copy, also to filter out any unwanted information.

The syntax for validation strings is very simple:

```
    <keyword>[,<keyword>]/<escape>|/<escape>]...
```

Keywords are used to associate each command with an escape sequence, there can be more than one keyword associated with a particular escape field, this is provided for two reasons, the first is when a different word has the same meaning, eg. Colour or Color. And secondly when defining the various states for a switch.

The escape sequence describès how the preceding data should be treated and also that to do with the rest of the special field string (up to the next separator).

The following characters are valid in escape sequences:

\N	number
\S	switch
\G	GSTrans (NYI)

Within the special field string each parameter is separated by a comma or a character which is out of place, ie a non-numeric in a numerical field. Each keyword within the special field string is separated by a semi-colon.

The buffer passed to the device contains the following:

<word> per escape character
<data> for GSTrans'd fields.

The words are reset to &DEADDEAD on entry to indicate that they are currently of no use, as each command sequence is decoded then its contents is placed into the return buffer.

Numbers are simply stored into the word, they are decoded using OS_ReadUnsigned and stored away. Switches, these store the state of the keywords placed, ie:

```
    foo,poo/S
```

This yields 0 if 'foo' is present within the string, 1 if 'poo' is present within the string.

The order of commands within the validation string and the special field string do not match, the commands within the validation string control how the values are returned back to the caller.

Service Calls

Service_DeviceFSStarting (Service Call &70)

DeviceFS is starting

On entry

R1 = &70 (reason code)

On exit

All registers preserved

Use

This code is issued when the module wants the device drivers to re-register with DeviceFS, the call is issued during the module initialisation. In this case it is actually issued on a callback to ensure that the module has been correctly linked into the module chain.

Service_DeviceFSDying (Service Call &71)

DeviceFS is dying

On entry

R0 = 0
R1 = &71 (reason code)

On exit

All registers preserved

Use

This is issued when DeviceFS is about to be killed, the device driver will already have had all of its streams closed and will have received the DeviceFS_DeviceDead service.

Service_DeviceDead (Service Call &79)

Device has been killed by DeviceFS

On entry

R0 = 0
R1 = &79 (reason code)
R2 = handle of device driver, or 0 for all
R3 = device name, or 0 if device driver dead, or undefined if R2 ≤ 0

On exit

All registers must be preserved.

Use

This is issued to inform the specified device that it has been killed. This is usually caused by another device of the same name being registered, when the original one is therefore killed to stop duplicates.

Service_DeviceFSCloseRequest (Service Call &81)

Opening a device which already has the maximum number of streams open

On entry

On exit

Use

SWI Calls

DeviceFS_Register (SWI &42740)

On entry

R0 = global flags for devices:
bit 0 clear ⇒ character device, set ⇒ block device
bit 1 clear ⇒ device is not full duplex, set ⇒ device is full duplex
all other bits reserved (must be zero)
R1 = pointer to list of devices to be installed
R2 = pointer to device driver entry point
R3 = private word
R4 = workspace pointer
R5 = pointer to validation string for special fields (0 ⇒ none)
R6 = maximum number of RX devices (0 ⇒ none, −1 ⇒ unlimited)
R7 = maximum number of TX devices (0 ⇒ none, −1 ⇒ unlimited)

On exit

R0 = device driver's handle

Use

This call registers device driver and its associated devices with DeviceFS. The device driver is the actual interfacing code with the hardware, and the device acts as a port into the driver. A device driver may have many devices within it; for instance you may have devices to support both buffered and unbuffered transfer.

Flags word

R0 contains a global flags word which describes all the driver's devices. It contains the following bit fields:

- Bit 0 is used to indicate if the devices are character or block devices.

  An example of a block device is a floppy disc drive, where data is transferred in blocks (sectors) to the caller. Examples of character devices are a parallel port or serial port.

  Block devices are not supported under the RISC OS 3 implementation of DeviceFS.

- Bit 1 is used to indicate if the device is full duplex or not

  A full duplex device can handle both input and output streams at the same time.

List of devices

R1 contains a pointer to a list of devices to be associated with this device driver. The list is terminated by a null word, and can be empty as you can use the SWI DeviceFS_RegisterObject to register devices later. The format of each entry in the list is as follows:

Offset	Meaning
0	offset to device name
4	flags: bit 0 set ⇒ device is buffered bit 1 set ⇒ create path variable for use as pseudo filing system
8	default flags for the device's RX buffer
12	default size of RX buffers
16	default flags for the device's TX buffer
20	default size of TX buffers

Device names should be registered with Acorn; see the chapter entitled *Appendix H: Registering names* on page 6-473. They are used for several things:

- The device name is used in the DeviceFS directory structure.
- The device name is used to create an option variable (initially null) named `DeviceFS$Device$Options`, so long as one does not already exist.

  This is used for storing device setup options, and is concatenated with the special field strings when streams are opened.

  If the options variable already exists, it is preserved, thus preserving the last setup used for the same device.
- The device name is used to create – if specified in the flags word – a path variable named `Device$Path` which points to the driver's entry point. The device can then be accessed via the pseudo filing system `device:`.

The device's buffers are not created until a stream is opened onto it. The flags are passed to the buffer manager; see page 5-410.

If for any reason a device in the list should fail to register than all devices specified will be removed.

Device driver entry point

R2 contains the pointer to the device driver entry point, which is called with various reason codes to access the routines available in the driver. See the chapter entitled *Writing a device driver* on page 4-71.

Parameters passed to driver: private word, and workspace pointer

R3 and R4 contain parameters which are passed to the device driver whenever its entry point is called. The parameter in R3 is passed to the device driver in R8, and might be used as a private word to indicate which hardware platform is being used; the parameter in R4 is used as a workspace pointer and is passed to the device in R12.

Validation string

On entry R5 contains the pointer to a validation string used to decode special fields within the device. For a full explanation, see the section entitled *Special fields* on page 3-402.

This value can be 0 which means that the string will be passed to the device unparsed; in these cases any unknown keywords should be ignored, as some keywords used by DeviceFS will still be present.

Number of output streams

R6 and R7 contain the maximum number of input and output streams on a device. If a register is zero then the device does not support that operation; if a register is –1 then the device has unlimited support for that type of transfer, and will be called to open streams.

DeviceFS uses these values to range check the number of streams being opened, so the device driver need not worry about this.

Device driver's handle

You will need to use the returned handle of the device driver to refer to it in any further SWI calls you make to DeviceFS.

DeviceFS_Deregister (SWI &42741)

On entry

R0 = device driver's handle

On exit

R0 preserved

Use

This call deregisters all devices and their device driver from DeviceFS. This causes all streams to be closed and any system variables set up for the device to be unset. The exception to this is the `DeviceFS$`*`Device`*`$Options` variable, which is left intact so that when the device is reloaded it can assume its original setup.

DeviceFS_RegisterObject (SWI &42742)

On entry

R0 = device driver's handle
R1 = pointer to list of devices to be registered with device driver

On exit

—

Use

This call registers a list of additional devices with a device driver. This is an extension to the DeviceFS_Register SWI which itself allows devices to be registered at the same time as their device driver.

The list of devices pointed to by R1 has the same format as that used in DeviceFS_Register (see page 3-408).

DeviceFS_DeregisterObject (SWI &42743)

On entry

R0 = device driver's handle
R1 = pointer to device name of the device to remove

On exit

—

Use

This call deregisters a device related to a particular device driver, tidying up as required.

DeviceFS_CallDevice (SWI &42744)

On entry

R0 = reason code
R1 = pointer to device driver handle, or pointer to path,
or 0 to broadcast to all devices
R2 - R7 = parameters passed to device driver
R12 = pointer to workspace

On exit

Register values returned by device (ie device/call-dependent)

Use

This call is used to make a call to a device with the specified register set. You can direct the call at a specific device or at all devices. When directing a call at a specific device you can specify this either by its device driver's handle, or by its filename within the directory structure (which can include '$').

DeviceFS_Threshold (SWI &42745)

On entry

R1 = DeviceFS stream handle, as passed to device driver on initialisation
R2 = threshold value to be used, or –1 to read

On exit

R1, R2 preserved

Use

This call is made by a device driver to set the threshold value used on buffered devices. DeviceFS will call the device drivers 'Halt' and 'Resume' entry points appropriately when the buffer levels get close to the specified threshold.

An error is generated if the device is not buffered.

DeviceFS_ReceivedCharacter (SWI &42746)

On entry

R0 = byte received
R1 = DeviceFS stream handle, as passed to device driver on initialisation

On exit

C set ⇒ byte not transferred, else C clear

Use

This call is made by a device driver when it receives a character. DeviceFS then attempts to process the character as required, unblocking any streams that may be waiting for the character or simply inserting it into a buffer.

For speed, DeviceFS_TransmitCharacter and DeviceFS_ReceivedCharacter do not validate the external handle passed; be warned that some strange effects can occur by passing in bad handles.

The C flag is cleared if the transfer was successful.

DeviceFS_TransmitCharacter (SWI &42747)

On entry

R1 = DeviceFS stream handle, as passed to device driver on initialisation

On exit

R0 = character to transmit (8 bits) if C clear

C set ⇒ unable to read character to be transmitted

Use

This call is made by a device driver when it wants to transmit a character. DeviceFS then attempts to obtain the character to be sent, either by extracting from a buffer or reading it from a waiting stream.

For speed, DeviceFS_TransmitCharacter and DeviceFS_ReceivedCharacter do not validate the external handle passed; be warned that some strange effects can occur by passing in bad handles.

The C flag is cleared if the transfer was successful.

38 Serial device

The serial device is provided as a DeviceFS (*Device Filing System*) device. For full details, see the chapter entitled *DeviceFS* on page 3-401.

OS_SerialOp

For your convenience, we've also documented the kernel's OS_SerialOp SWI here, even though it properly belongs in *Part 2 – The kernel*. This SWI provides routines to access the serial device driver directly. It is like OS_Byte in that it contains a number of operations, determined by the reason code passed in R0. The advantages of using this approach are the speed of not going through several routines in the stream system and no possibility of confusion about where the data is going.

OS_Byte calls

There are also a number of OS_Byte commands for controlling the serial port, that are in RISC OS mainly for compatibility with earlier Acorn operating systems. Again, we've documented them here rather than with the kernel documentation. **We strongly recommended that you use the OS_SerialOp commands** in preference to the OS_Bytes because they are more complete and consistent.

Note that the serial device's input and output sides may be controlled independently. For example, you can transmit at a different baud rate from the one which is being used to receive, although hardware restrictions mean that this is not possible on machines fitted with the 82C710 or 82C711 controller.

Technical details

Serial Device

The serial device driver provides facilities to send and receive a byte, control the handshake lines and alter the protocol of the data. RISC OS provides a number of SWIs that allow access to these facilities.

Summary of commands

OS_SerialOp

Here is a summary of the OS_SerialOp commands:

- OS_SerialOp 0 reads and writes the handshaking status.
- OS_SerialOp 1 reads and writes the data format.
- OS_SerialOp 2 sends a break.
- OS_SerialOp 3 sends a byte.
- OS_SerialOp 4 gets a byte.
- OS_SerialOp 5 reads and writes the receive baud rate.
- OS_SerialOp 6 reads and writes the transmit baud rate.

OS_Byte

Below is a summary of the OS_Byte commands in this chapter:

- OS_Byte 7 sets the receive baud rate.
- OS_Byte 8 sets the transmit baud rate.
- OS_Byte 156 reads/writes various state information from/to a bit mask.
- OS_Byte 181 makes the data that comes in from a serial port appear to RISC OS as if it had been typed at the keyboard.
- OS_Byte 191 reads and writes the busy flag (an obsolete BBC usage)
- OS_Byte 192 reads from the above state byte.
- OS_Byte 203 reads/writes the serial input buffer minimum space.
- OS_Byte 204 stops any incoming data being buffered by the serial driver. The port is still active, and serial errors can still occur, but the data is discarded.
- OS_Byte 242 reads both baud rates.

Remember, where possible you should use OS_SerialOp calls in preference to OS_Byte calls.

Streams

RISC OS uses streams for character input, character output, and printer output. There are OS_Byte calls to set the source and destination(s) of these streams. As an innate part of the character input/output system, they are described in full in the chapters entitled *Character Input* and *Character Output*, but we summarise them below.

Of course, you can also use OS_SerialOp calls to independently send and receive characters via the serial port; generally this is preferable.

OS_Byte 2

This call selects the device from which all subsequent input is taken by OS_ReadC. This is determined by the value of R1 passed as follows:

Value of R1	Source of input
0	Keyboard, with serial input buffer disabled
1	Serial port
2	Keyboard, with serial input buffer enabled

The difference between the 0 and 2 values is that the latter allows characters to be received into the serial input buffer under interrupts at the same time as the keyboard is being used as the primary input. If the input stream is subsequently switched to the serial device, then those characters can then be read.

For full details of OS_Byte 2, see page 2-359.

OS_Byte 3

This call selects which output stream(s) are active, and will hence receive all subsequent output from OS_WriteC and its derivatives. A bit mask in R1 determines this:

Bit	Effect if set
0	Enables serial driver
1	Disables VDU drivers
2	Disables VDU printer stream
3	Enables printer (independently of the VDU)
4	Disables spooled output
5	Calls VDUXV instead of VDU drivers (see the chapter on VDU)
6	Disables printer, apart from VDU 1,n
7	Not used

Thus to send characters to the serial output stream, call OS_Byte 3 with bit 0 of R1 set. Such characters sent are inserted into the serial output buffer (buffer number 2), where they remain until removed by the interrupt routine dealing with serial transmission.

For full details of OS_Byte 3, see page 2-18.

OS_Byte 5

This call sets which printer driver type (and hence printer port) is used for subsequent printer output. The value of R1 on entry determines this:

Value of R1	Printer driver type
0	Printer sink
1	Parallel (Centronics) printer driver
2	Serial output
3 - 255	Files in system variables PrinterType$*n* (eg the NetPrint module sets up PrinterType$4)

Thus to send printer output to the serial port, call OS_Byte 5 with R1 = 1.

Note that if the serial port is selected as the printer by OS_Byte 5, and the serial port is enabled with OS_Byte 3, then the character is inserted into both buffers. This means that eventually the character is printed twice, first from the serial output buffer and then from the printer buffer. To solve this problem, make the printer another device type, such as the printer sink, which allows data sent to the printer to be ignored.

For full details of OS_Byte 5, see page 2-20.

Serial buffers

Input buffer

The serial driver will attempt to stop the sender transmitting when the amount of free space in the serial input buffer falls below a set level. The idea is that this space gives enough time for the sender to recognise the command and stop without overflowing the buffer. OS_Byte 203 can change the setting of this level.

Output buffer

If the output buffer is already full and there is nothing communicating with the serial port, when you insert another character the machine temporarily halts while it waits for a character to be removed to make space for the new character. An escape condition abandons this wait.

Handshaking and protocol

When trying to get communications working with an external device using the serial device, there are several important factors to remember:

- The receiver must be electrically compatible with RS423 or RS232.

- The handshaking lines must be connected between the sender and receiver in exactly the right way.
- The sender must match baud rates with the receiver.
- They must also match the transmission protocol. Each byte sent is packaged up in some variation of the following sequence:
  1. A start bit synchronises the receiver with the sender.
  2. The number of bits of actual data sent is variable from 5 to 8.
  3. There can be an optional parity bit, which is used to check that no errors have taken place during transmission.
  4. It ends with a stop bit, either 1, 1.5 or 2 bits long.

Note that the default setup of the serial protocol (configured in CMOS RAM) is different from some earlier Acorn machines. For example, the setup for RISC OS machines is the same as the Master series (8 data bits, no parity, 2 stop bits), but different from the original BBC series (8 data bits, no parity, 1 stop bit).

Serial line names

Coming out of the serial connector are many lines. This is a list of their names and common abbreviations:

- data receive (RxD)
- data transmit (TxD)
- ground (0V)
- request to send (RTS)
- confirm to send (CTS)
- data carrier detect (DCD)
- data terminal ready (DTR)
- data set ready (DSR)
- ring indicator (RI)

Refer to the documentation accompanying your particular communications device for information on how to wire these lines correctly with the serial port. For further information, contact Acorn Customer Support.

SWI Calls

OS_Byte 7 (SWI &06)

Write serial port receive rate

On entry

R0 = 7
R1 = baud rate code

On exit

R0 preserved
R1, R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets the serial port baud rate for receiving data as follows:

Value	Baud rate
0	9600
1	75
2	150
3	300
4	1200
5	2400
6	4800
7	9600
8	19200
9	50
10	110
11	134.5
12	600
13	1800
14	3600
15	7200

The settings from 0 to 8 are in an order compatible with earlier operating systems. The other speeds from 9 to 15 provide all the other standard baud rates.

The default rate is that set by *Configure Baud.

Related SWIs

OS_Byte 8 (SWI &06)

Related vectors

ByteV

OS_Byte 8 (SWI &06)

Sets the transmit baud rate for the serial port

On entry

R0 = 8 (reason code)
R1 = baud rate code

On exit

R0 preserved
R1, R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets the transmit baud rate for the serial port. It is provided for compatibility with older operating systems, and you should use OS_SerialOp 6 instead: see page 3-452.

(This call uses the same baud rate codes as OS_SerialOp 6.)

Related SWIs

OS_Byte 7 (page 3-424)

Related vectors

ByteV

OS_Byte 156 (SWI &06)

Reads/writes serial port state

On entry

R0 = 156 (reason code)
R1 = 0 or new value
R2 = 255 or 0

On exit

R0 preserved
R1 = value before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The value stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((value AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call accesses the control byte of the serial port. In addition to updating the status byte in RAM, it also updates the hardware register which controls the serial port characteristics.

The call enables the current settings of the transmitter, receiver, interrupts and the serial handshake line Request To Send (RTS) to be read or altered.

When writing, the effect depends on the bits in R1:

Bit 1	Bit 0	Effect
0	0	No effect
0	1	No effect
1	0	No effect
1	1	Reset transmit, receive and control registers

Bit 4	Bit 3	Bit 2	Word length	Parity	Stop bits
0	0	0	7	even	2
0	0	1	7	odd	2
0	1	0	7	even	1
0	1	1	7	odd	1
1	0	0	8	none	2
1	0	1	8	none	1
1	1	0	8	even	1
1	1	1	8	odd	1

Bit 6	Bit 5	Transmission control
0	0	RTS low, transmit interrupt disabled
0	1	RTS low, transmit interrupt enabled
1	0	RTS high, transmit interrupt disabled
1	1	RTS low, transmit break level on transmit data, transmit interrupt disabled

The above bits should not be modified as they are controlled by the OS. Use the OS_SerialOp SWIs instead to control transmission.

Bit 7	Receive interrupt
0	Disabled
1	Enabled

The default setting for bits 2 - 4 comes from the *Configure Data value, shifted left by two bits. The current value of this byte may be read (but not set) using OS_Byte 192 (page 3-433).

OS_SerialOps 0 and 1 provide all of these facilities and more, with the exception of the interrupt control bit. The receive interrupt/control bit can be set/cleared via OS_Byte 2 (page 2-359). You should not change the RTS/transmit IRQ bits; RISC OS handles this function.

This call is provided for compatibility only and should not be used. In all cases you should use OS_SerialOp (page 3-440) to provide these functions.

Related SWIs

OS_Byte 192 (page 3-433), OS_SerialOp (page 3-440)

Related vectors

ByteV

OS_Byte 181 (SWI &06)

Read/write serial input interpretation status

On entry

R0 = 181
R1 = 0 to read or new state to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = state before being overwritten
R2 = NoIgnore state (see OS_*Byte* 182 (SWI &06) on page 2-23)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The state stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((state AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

Usually, top-bit-set characters read from the serial input buffer are not treated specially. For example, if the remote device sends the code &85, when this is read, using OS_ReadC for example, that ASCII code will be returned to the caller immediately. It is sometimes useful to be able to treat serial input characters in exactly the same way as keyboard characters. OS_Byte 181 allows this.

The state value passed to this call has two values:

0 In this state the keyboard interpretation is placed on characters read from the serial input buffer.

1 This is the default state in which no keyboard interpretation is done This means that:

- the current escape character is ignored
- the function key codes are not expanded
- events are not generated.

Related SWIs

None

Related vectors

ByteV

OS_Byte 191 (SWI &06)

Read/write serial busy flag

On entry

R0 = 191 (reason code)
R1 = 0 or new value
R2 = 255 or 0

On exit

R0 = preserved
R1 = state before being overwritten
R2 = value of serial port control byte (see OS_Byte 192 – page 3-433)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call is provided for compatibility reasons only; the cassette interface and RS423 serial port shared the same hardware on the BBC/Master 128 machines. It performs no useful function under RISC OS.

Related SWIs

None

Related vectors

ByteV

OS_Byte 192 (SWI &06)

Reads the serial port state

On entry

R0 = 192 (reason code)
R1 = 0
R2 = 255

On exit

R0 = preserved
R1 = value of communications state
R2 = value of flash counter (see OS_Byte 193 (SWI &06) on page 2-165)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call reads the control byte of the serial port. It is equivalent to a read operation with OS_Byte 156.

This call should not be used to write the value back, as to do so would make the RISC OS copy of the register inconsistent with the actual register in the serial hardware.

Related SWIs

OS_Byte 156 (page 3-427), OS_SerialOp (page 3-440)

Related vectors

ByteV

OS_Byte 203 (SWI &06)

Read/write serial input buffer minimum space

On entry

R0 = 203
R1 = 0 to read or new value to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 = serial ignore flag (see OS_Byte 204)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The value stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((value AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

The serial input routine attempts to halt input when the amount of free space left in the input buffer falls below a certain level. This call allows the value at which input is halted to be read or changed.

OS_SerialOp 0 can be used to examine or change the handshaking method.

The default value is 9 characters.

Related SWIs

None

Related vectors

ByteV

OS_Byte 204 (SWI &06)

Read/write serial ignore flag

On entry

R0 = 204
R1 = 0 to read or new flag to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The flag stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((flag AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call is used to read or change the flag which indicates whether serial input is to be buffered or not. Although this call can stop data being placed in the serial input buffer, data is still received by the serial driver. Errors will still generate events unless they have been disabled by OS_Byte 13.

If the flag is zero, then serial input buffering is enabled. Any non-zero value disables it.

Related SWIs

OS_Byte 13 (SWI &06)

Related vectors

ByteV

OS_Byte 242 (SWI &06)

Read serial baud rates

On entry

R0 = 242 (&F2) (reason code)
R1 = 0
R2 = 255

On exit

R0 = preserved
R1 = baud rates
R2 = timer switch state (see OS_Byte 243 (SWI &06) on page 1-399)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

R1 returns an encoded value which gives the baud rate for serial receive and transmit. Originally, in the BBC/Master operating systems, only eight baud rates were available. These could be encoded in three bits each for receive and transmit. Under RISC OS, 15 are available, which requires four bits to encode. For compatibility with this earlier format, the layout of this byte looks unusual:

Bit	Meaning
0	Transmit bit 0
1	Transmit bit 1
2	Transmit bit 2
3	Receive bit 0
4	Receive bit 1
5	Receive bit 2
6	Receive bit 3
7	Transmit bit 3

These four bit groups are encoded with baud rates. Note that this order is not the same as the order used by any other baud rate setting SWI. This order is based on the original hardware:

Value	Baud Rate
0	19200
1	1200
2	4800
3	150
4	9600
5	300
6	2400
7	75
8	7200
9	134.5
10	1800
11	50
12	3600
13	110
14	600
15	undefined

The value stored must not be changed by making R1 and R2 other than the values stated above.

This call is provided for backwards compatibility with the BBC and Master operating systems. You should in preference use OS_SerialOps 5 and 6 to read and write baud rates.

Related SWIs

OS_Byte 7 (page 3-424), OS_Byte 8 (page 3-426), OS_SerialOp (page 3-440)

Related vectors

ByteV

OS_SerialOp (SWI &57)

Low level serial operations

On entry

R0 = reason code
other input registers as determined by reason code

On exit

R0 preserved
other registers may return values, as determined by the reason code passed.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is like OS_Byte in that it is a single call with many operations within it. The operation required, or reason code, is passed in R0. It can have the following meanings:

R0	Meaning
0	Read/write serial states
1	Read/write data format
2	Send break
3	Send byte
4	Get byte
5	Read/write receive baud rate
6	Read/write transmit baud rate

On the following pages is a detailed explanation of each of these reason codes in turn.

Related SWIs

None

Related vectors

None

OS_SerialOp 0 (SWI &57)

Read/write serial status

On entry

R0 = 0 (reason code)
R1 = EOR mask
R2 = AND mask

On exit

R0 preserved
R1 = old value of state
R2 = new value of state

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The structure of this call is very similar to that of OS_Bytes between SWI &A6 and SWI &FF. The new state is determined by:

New state = (Old state AND R2) EOR R1

This call is used to read and write various states of the serial system. These states are presented as a 32-bit word. The bits in this word represent the following states:

Bit	Read/Write or Read Only	Value	Meaning
0	R/W	0	No software control. Use RTS handshaking if bit 5 is clear.
		1	Use XON/XOFF protocol. Bit 5 is ignored. The hardware will still do CTS handshaking (ie if CTS goes low, then transmission will stop), but RTS is not forced to go low.
1	R/W	0	Use the ~DCD bit. If the ~DCD bit in the status register goes high, then cause a serial event. Also, if a character is received when ~DCD is high, then cause a serial event, and do not enter the character into the buffer.
		1	Ignore the ~DCD bit. Note that some serial chips (GTE and CMD) have reception and transmission problems when this bit is high.
2	R/W	0	Use the ~DSR bit. If the ~DSR bit in the status register is high, then do not transmit characters.
		1	Ignore the state of the ~DSR bit.
3	R/W	0	DTR on (normal operation).
		1	DTR off (on 6551 serial chips, cannot use serial port in this state).
4	R/W	0	Use the ~CTS bit. If the ~CTS bit in the status register is high, then do not transmit characters.
		1	Ignore the ~CTS bit (not supported by 6551 serial chips).
5	R/W		This bit is ignored if bit 0 is set. If bit 0 is clear:
		0	Use RTS handshaking.
		1	Do not use RTS handshaking.
6	R/W	0	Input is suppressed.
		1	Input is not suppressed.
7	R/W		Users should only modify this bit if RTS handshaking is not in use:
		0	RTS controlled by handshaking system (low if no RTS handshaking).
		1	RTS high.
8 - 15	RO		These bits are reserved for future expansion; do not modify them.
16	RO	0	XOFF not received.
		1	XOFF has been received. Transmission is stopped by this occurrence.

17	RO	0	The other end is intended to be in XON state.
		1	The other end is intended to be in XOFF state. When this bit is set, then it means that an XOFF character has been sent and it will be cleared when an XON is sent by the buffering software. Note that the fact that this bit is set does not imply that the other end has received an XOFF yet.
18	RO	0	The ~DCD bit is low, ie carrier present.
		1	The ~DCD bit is high, ie no carrier.
19	RO	0	The ~DSR bit is low, ie 'ready' state.
		1	The ~DSR bit is high, ie 'not-ready' state.
20	RO	0	The ring indicator bit in IOC is low.
		1	The ring indicator bit in IOC is high.
21	RO	0	Do not send break
		1	Send break
22	RO	0	User has not manually sent an XOFF.
		1	User has manually sent an XOFF.
23	RO	0	Space in receive buffer above threshold.
		1	Space in receive buffer below threshold.
24 - 31	RO		These bits are reserved for future expansion; do not modify them.

Note that if XON/XOFF handshaking is used, then OS_Byte 2,1 or 2,2 must be called beforehand.

RISC OS 2 does not support bits 4-7 and 21-23 inclusive.

Related SWIs

OS_Byte 156 (page 3-427)

Related vectors

None

OS_SerialOp 1 (SWI &57)

Read/write data format

On entry

R0 = 1 (reason code)
R1 = –1 to read or new format value

On exit

R0 = preserved
R1 = old format value.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the encoding of characters when sent and received on the serial line. The bits in this word represent the following formats:

Bit	Read/Write or Read Only	Value	Meaning
0,1	R/W	0	8 bit word.
		1	7 bit word.
		2	6 bit word.
		3	5 bit word.
2	R/W	0	1 stop bit.
		1	2 stop bits in most cases. 1 stop bit if 8 bit word with parity. 1.5 stop bits if 5 bit word without parity.
3	R/W	0	parity disabled.
		1	parity enabled.
4,5	R/W	0	odd parity.
		1	even parity.
		2	parity always 1 on TX and ignored on RX.
		3	parity always 0 on TX and ignored on RX.
6 - 31			reserved – must be set to zero.

Related SWIs

OS_Byte 156 (page 3-427)

Related vectors

None

OS_SerialOp 2 (SWI &57)

Send break

On entry

R0 = 2 (reason code)
R1 = length of break in centiseconds

On exit

R0 = preserved
R1 = preserved.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the ACIA to transmit a break, then waits R1 centiseconds before resetting it to normal. Any character being transmitted at the time the call is made may be garbled. After sending the break the transmit process is either awakened if the buffer is not empty, or made dormant if the buffer is empty.

Related SWIs

None

Related vectors

None

OS_SerialOp 3 (SWI &57)

Send byte

On entry

R0 = 3 (reason code)
R1 = character to be sent

On exit

R0 = preserved
R1 = preserved
if C flag = 0 then character was sent
if C flag = 1 then character was not sent because the buffer was full

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call puts a character in the serial output buffer, and re-enables the transmit interrupt if it had been disabled by RISC OS.

If the serial output buffer is full, the call returns immediately with the C flag set.

Related SWIs

None

Related vectors

None

OS_SerialOp 4 (SWI &57)

Get a byte from the serial buffer

On entry

R0 = 4

On exit

R0 preserved
if C flag = 0 then R1 = character received
if C flag = 1 then R1 preserved.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call removes a character from the serial input buffer if one is present. If removing a character leaves the input buffer with more free spaces than are specified by OS_Byte 203, then transmission is re-enabled in the way specified by the state set by reason code 0.

Note that reception must have been enabled using OS_Byte 2 before this call will have any effect.

Related SWIs

OS_Byte 2 (SWI &06), OS_Byte 203 (SWI &06)

Related vectors

None

OS_SerialOp 5 (SWI &57)

Read/write RX baud rate

On entry

R0 = 5 (reason code)
R1 = –1 to read or 0 - 15 to set to a value

On exit

R0 = preserved
R1 = old receive baud rate

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The baud rate codes are as follows:

Value of R1	Baud rate
0	9600
1	75
2	150
3	300
4	1200
5	2400
6	4800
7	9600
8	19200
9	50
10	110
11	134.5
12	600
13	1800
14	3600
15	7200

The settings from 0 to 8 are in an order compatible with earlier operating systems. The other speeds from 9 to 15 provide all the other standard baud rates.

The default rate is set by *Configure Baud.

This call has the same effect as an OS_Byte 7 for writing.

Related SWIs

OS_Byte 7 (SWI &06)

Related vectors

None

OS_SerialOp 6 (SWI &57)

Read/write TX baud rate

On entry

R0 = 6 (reason code)
R1 = –1 to read or 0 - 15 to set to a value

On exit

R0 = preserved
R1 = old transmit baud rate

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The baud rate codes are as follows:

Value of R1	Baud rate
0	9600
1	75
2	150
3	300
4	1200
5	2400
6	4800
7	9600
8	19200
9	50
10	110
11	134.5
12	600
13	1800
14	3600
15	7200

The settings from 0 to 8 are in an order compatible with earlier operating systems. The other speeds from 9 to 15 provide all the other standard baud rates.

The default rate is set by *Configure Baud.

This call has the same effect as an OS_Byte 8 for writing.

Related SWIs

OS_Byte 8 (page 3-426)

Related vectors

None

*Commands

*Configure Baud

Sets the configured baud rate for the serial port

Syntax

```
*Configure Baud n
```

Parameters

n	0 to 8

Use

*Configure Baud sets the configured receive and transmit baud rates for the serial port. The values of n correspond to the following baud rates:

n	Baud rate
0	9600
1	75
2	150
3	300
4	1200
5	2400
6	4800
7	9600
8	19200

The default value is 4 (1200 baud).

The change takes effect on the next reset.

Example

```
*Configure Baud 7
```
sets the configured baud rate to 9600

Related commands

None

Related SWIs

OS_Byte 7 (page 3-424), OS_Byte 8 (page 3-426), OS_SerialOp 5 (page 3-450), OS_SerialOp 6 (page 3-452)

Related vectors

None

*Configure Data

Sets the configured data word format for the serial port.

Syntax

```
*Configure Data n
```

Parameters

n 0 to 7

Use

*Configure Data sets the configured data word format for the serial port. The values of *n* correspond to the following formats:

n	Word length	Parity	Stop bits
0	7	even	2
1	7	odd	2
2	7	even	1
3	7	odd	1
4	8	none	2
5	8	none	1
6	8	even	1
7	8	odd	1

The default value is 4 (8 bits, no parity, 2 stop bits).

The change takes effect on the next reset.

Example

```
*Configure Data 0
```
(7 bits, even parity, 2 stop bits)

Related commands

None

Related SWIs

OS_Byte 156 (page 3-427), OS_SerialOp 1 (page 3-445)

Related vectors

None

39 Parallel device

Introduction and Overview

This module provides parallel device support. It is not available in RISC OS 2. The module is a client of DeviceFS and can be accessed via that system.

It will setup PrinterTypeS1 to point at its DeviceFS object, ie:

PrinterTypeS1 ⇒ devices#buffer3:S.Parallel

The module supports a single SWI to allow the 82C710 or 82C711 chip driving the parallel port to be directly accessed (if present – some machines use other chips).

SWI calls

Parallel_HardwareAddress (SWI &42EC0)

Allows 82C710/82C711 to be driven directly

On entry

—

On exit

R0 = pointer to base address of parallel 82C710/82C711 in IOEB space (0 if no 710 present)

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is provided to allow external authors to drive the 82C710/82C711 for themselves, the SWI is provided to return the base address only and does not give any other support for the device.

If you intend to drive the hardware directly then you should perform the following:

```
lock =OPENOUT("parallel:")
.... play around with hardware....
CLOSE#lock
```

This stops the device driver altering the values that you have setup preventing any possible confusion.

Note that older RISC OS machines do not use these controller chips.

Related SWIs

None

Related vectors

None

40 System devices

System devices

The SystemDevices module provides a number of system devices, which behave like files in some ways. You can use them anywhere you would normally use a file name as a source of input, or as a destination for output. They include:

System devices suitable for input

`kbd:`	the keyboard, reading a line at a time using OS_ReadLine (this allows editing using Delete, Ctrl-U, and other keys)
`rawkbd:`	the keyboard, reading a character at a time using OS_ReadC
`null:`	the 'null device', which effectively gives no input

System devices suitable for output

`vdu:`	the screen, using GSRead format passed to OS_WriteC
`rawvdu:`	the screen, via the VDU drivers and OS_WriteC
`printer:`	the printer
`netprint:`	the network printer driver (provided by the NetPrint module)
`null:`	the 'null device', which swallows all output

An error is given if the specified system device is not present; for example, if the SystemDevices module is not present.

Other devices

There are also two devices provided as a part of the DeviceFS system:

`serial:`	serial port; see the chapter entitled *Serial device* on page 3-419
`parallel:`	parallel port; see the chapter entitled *Parallel device* on page 3-457

Redirection

These system devices can be useful with commands such as *Copy, and the redirection operators (> and <):

`*Copy myfile printer:`	Send `myfile` to the printer
`*Cat { > printer: }`	List the files in the current directory to the printer

Suppressing output using null:

You can use the system device `null:` to suppress unwanted output from a command script or program:

`*myprogram { > null: }`	Run `myprogram` with no output

Input devices

You can only open one file for input on `kbd:` at once as it has buffered input; normal line editing facilities are available. If you try to open `kbd:` a second time whilst the first file is open, you will get returned a handle of 0, or an error if the appropriate bit is set in the open mode passed to FileSwitch. Ctrl-D in the input line will yield EOF when it is read from the buffer.

You can open `rawkbd:` as many times as you like, even if a file is open on `kbd:`. It uses XOS_ReadC (without echoing to the screen) to read characters. No EOF condition exists on `rawkbd:`; the program reading it must detect an input value/pattern and stop on that.

No files exist on any of these devices. If you call OS_File 5 on the devices it will always return object type 0, so you cannot use them for input to programs that need to load an entire file at once for processing.

netprint:

The netprint: system device is more sophisticated that other ones. As well as using it in place of file names, you can also use it with certain commands that normally use the name of a filing system.

printer:

The `printer:` device allows various special fields, to refer to the different types of printers. These are:

- `printer#sink:` and `printer#null:`, which are synonyms
- `printer#parallel:` and `printer#centronics:`, which are synonyms
- `printer#serial:` and `printer#rs423:`, which are synonyms
- `printer#user:`
- `printer#`*`n`*`:`, which refers to printer type *n*, where *n* is in the range 0 - 255.

You can open multiple files on `printer:`, provided they are on different devices and using different buffers.

Other output devices

You can open as many files as you wish on the other output devices, which are:

`null:`, `vdu:`, and `rawvdu:`

For example:

```
H% = OPENOUT "rawvdu:"
SYS"OS_Byte",199,H%,0
type here…
*Spool
```

When you type everything is sent to the vdu, which outputs it and then uses XOS_BPut to send it to the spool file handle. This in turn sends it (through another mechanism, OS_PrintChar) to the screen again! The `*Spool` at the end clears up.

In addition to byte-oriented operations, you are allowed to perform file save operations on the output devices.

The difference between `vdu:` and `rawvdu:` is that the former is filtered using the configured DumpFormat, whereas the latter sends its characters straight to the VDU drivers.

The RISC OS 2 serial device

RISC OS 2 provided its serial port device as a part of the SystemDevices module. It has since been reimplemented as a device; see the chapter entitled *Serial device* on page 3-419.

The RISC OS 2 serial device (`serial:`) is bidirectional, has no EOF condition, and allows multiple files to be opened.

41 The Filer

Introduction and Overview

...

See the section entitled *Filer messages* on page 4-292 for full details on how the

Service Calls

Service_StartFiler (Service Call &4B)

Request to filing system modules to start up

On entry

R0 = Filer's task handle
R1 = &4B (reason code)

On exit

R1 = 0 to claim call
R0 = pointer to * Command to start module

Use

In order to ensure that filing system modules are not started up without the Filer module, they are started by a different mechanism. Rather than responding to the Service_StartWimp service call, they wait for the Filer module to start them up, using Service_StartFiler. The Filer behaves in a similar way to the Desktop, issuing the Service_StartFiler service call, followed by Wimp_StartTask, if the service call is claimed.

The Filer will try to start up any resident filing system module tasks when it is started (by responding to Service_StartWimp). It does this by issuing a service call Service_StartFiler (&4B).

If this call is claimed, the Filer starts the task by passing the * Command returned by the module to Wimp_StartTask. It then issues the service again, and repeats this until no-one claims it.

A module's service call handler should deal with this reason code as follows:

```
serviceCode
        LDR     R12, [R12]                  ;Load workspace pointer
        STMFD   SP!, {LR}                   ;Save link and make R14 available
        TEQ     R1, #Service_StartFiler     ;Is it service &4B?
        BEQ     startFiler                  ;Yes
        ...                                 ;Otherwise try other services
        LDMFD   SP!, {PC}                   ;Return

startFiler
        LDR     R14, taskHandle             ;Get task handle from workspace
        TEQ     R14, #0                     ;Am I already active?
        MOVEQ   R14, #-1                    ;No, so init handle to -1
        STREQ   R14, taskHandle             ;R12 relative
        ADREQ   R0, myCommand               ;Point R0 at command to start task
        MOVEQ   R1, #0                      ;(see earlier) and claim the service
        LDMFD   SP!, {PC}                   ;Return
```

Note that the `taskHandle` word of the module's workspace must be zero before the task has been started. This word should therefore be cleared in the module's initialisation code. If the task is not already running, the StartFiler code should set the handle to −1, load the address of a command that can be used to start the module, and claim the call. Otherwise (if `taskHandle` is non-zero) it should ignore the call.

The automatic start-up process is made slightly more complex by the necessity to deal elegantly with errors that occur while a module is trying to start up. If the appropriate code is not executed, the Desktop can get into an infinite loop of trying to initialise unsuccessful modules.

This is avoided by the task setting its handle to −1 when it claims the StartFiler service. If the task fails to start, this will still be −1 the next time the Filer issues a Service_StartFiler, and so it will not claim the service.

Note that the Filer passes its own `tas kHandle` to the module in R0 in the service call, to make it easier for the task to send it Message_FilerOpenDir messages later.

Service_StartedFiler (Service Call &4C)

Service_Reset (Service Call &27)

Request to filing system task modules to set `taskHandle` variable to zero

On entry

R1 = Service_StartedFiler (&4C) or Service_Reset (&27)

On exit

Module's `taskHandle` variable set to zero

Use

A task which failed to initialise would have its taskHandle variable stuck at the value –1, which would prevent it from ever starting again (as Service_StartFiler would never be claimed). In order to avoid this, the two service calls should be recognised by the filing system task modules. On either of them, the task handle should be set to zero:

```
serviceCode
...
        TEQ     R1, #Service_StartedFiler           ;Service &4C?
        BNE     tryServiceReset                     ;No
        LDR     R14, taskHandle                     ;taskHandle = -1?
        CMN     R14, #1
        MOVEQ   R14, #0                             ;Yes, so zero it
        STREQ   R14, taskHandle
        LDMFD   SP!, {PC}                           ;Return

tryServiceReset
        TEQ     R1, #Service_Reset                  ;Reset reason code?
        MOVEQ   R14, #0                             ;Yes, so zero handle
        STREQ   R14, taskHandle
        LDMFD   SP!, {PC}                           ;Return
...
```

Service_StartedFiler is issued when the last of the resident filing system task modules has been started, and Service_Reset is issued whenever the computer is soft reset.

Service_FilerDying (Service Call &4F)

Notification that the Filer module is about to close down

On entry

R1 = &4F (reason code)

On exit

Module's `taskHandle` variable set to zero

Use

If the Filer module task is closed down (e.g. if the module is *RMKilled, or the Filer task is quitted from the TaskManager window) the Filer module tries to ensure that all the other filing system tasks are also closed down, by issuing this service call.

On receipt of this service call, a filing system task should check to see if it is active and if it is, it should close itself down by calling Wimp_CloseDown as follows:

```
serviceCode
...
        TEQ     R1, #Service_FilerDying
        BNE     try next
        STMFD   SP!, {R0-R1, R14}
        LDR     R0, taskHandle          ;in workspace
        CMP     R0, #0
        MOVNE   R14, #0
        STRNE   R14, taskHandle
        LDRGT   R1, taskid
        SWIGT   XWimp_CloseDown
        LDMFD   SP!, {R0-R1, PC}^       ;can't return errors from service call

trynext
...
taskid  DCB     "TASK"                  ;word-aligned
```

Service_EnumerateFormats (Service Call &6A)

Get **Format** submenu entries

On entry

R1 = &6A (reason code)
R2 = pointer to list of format specifications suitable for a menu (initially 0)

On exit

R1 preserved to pass on (do not claim)
R2 = pointer to list of format specifications suitable for a menu

Use

This service call is issued by desktop Filer modules, both to get sufficient information to construct a **Format** submenu, and to support !Help for that submenu.

- This service should be issued at menu construction time, either when the user has clicked Menu on the Filer's icon, or when the pointer moves over the arrow leading to the **Format** submenu. If the Filer were instead to issue the service call as part of its initialisation, it is likely many formats would not be available (they may finish initialising later, or be soft-loaded later); consequently it is not recommended.

Each image filing system responds by adding entries to a linked list of blocks held in the RMA, each of which describes a format:

Offset	**Meaning**
0	Pointer to next of these blocks, or 0 to indicate end of list
4	Pointer to RMA block containing text suitable for inclusion in the format submenu
8	Pointer to RMA block containing text which is a suitable response for !Help for this entry in the format submenu
12	SWI number to call to obtain raw disc format information
16	Parameter in R3 to use when calling disc format SWI
20	SWI number to call to lay down disc structure
24	Parameter in R0 to use when calling disc structure SWI
28	Flags:

Bit	**Meaning when set**
0	'This is a native (ADFS) format'
1-31	Reserved – must be zero

The image filing system must fill in each block in this order:

1. Allocate a data block in the RMA to link into the linked list
2. Fill in 0 in the fields of the data block holding pointers to text
3. Link the data block to the list by filling in the pointer at offset 0
4. Allocate the RMA block to hold the text for the submenu entry
5. Attach that RMA block to the data block by filling in the pointer at offset 4
6. Allocate the RMA block to hold the help text for the submenu entry
7. Attach that RMA block to the data block by filling in the pointer at offset 8
8. Copy the text for the submenu entry into its RMA block
9. Copy the help text for the submenu entry into its RMA block
10. Fill in the rest of the data block

The image filing system must **not** set the pointers at offsets 4 and 8 to point at text embedded inside its code, but must instead copy the text into individually allocated RMA blocks.

Once it has filled in each block, it must pass on the service call for other image filing systems to attach their own formats.

This sequence of actions has been carefully constructed such that any error can be returned by claiming the service and returning both the error and an intact list. It is then the responsibility of the issuer of the service call to free the list.

The desktop Filer must also free the list when the user has chosen a format, and must then initiate the format using the given parameters.

Service_DiscDismounted (Service Call &7D)

Disc dismounted

On entry

R1 = &7D (reason code)
R2 = disc which has been dismounted

On exit

All registers are preserved

Use

Inform modules that a disc has just been dismounted. A module, such as the Filer, may wish to take action given this activity such as close its viewers.

The value in R2 should be a pointer to a null-terminated string of the following form:

<FS>::<Disc>

Where <FS> is the name of the filing system and <Disc> is the name of the disc. If the disc has no name then the drive should be filled in instead. For example, ADFS would issue the service call with these parameters: R1 = &7D, R2 = 'ADFS::MyFloppy' or, for an unnamed disc: R1 = &7D, R2 = 'ADFS::0'.

This service call should not be claimed.

* Commands

*Filer_Boot

Boots a desktop application

Syntax

```
*Filer_Boot application
```

Parameters

application	a valid pathname specifying an application, the !Boot file of which is to be run

Use

*Filer_Boot boots the specified desktop application by running its !Boot file. This command is most useful in Desktop boot files.

You can only use this command from within the desktop environment, or within a Desktop boot file.

Example

```
*Filer_Boot adfs::mhardy.$.Apps.!PrinterPS
```

Related commands

*Filer_Run

Related SWIs

None

Related vectors

None

*Filer_CloseDir

Closes a directory display on the Desktop

Syntax

```
*Filer_CloseDir directory
```

Parameter

directory — the full pathname of a directory whose directory display is to be closed

Use

*Filer_CloseDir closes a directory display on the Desktop, and any of its sub-directories. The directory display will typically have been opened by an earlier *Filer_OpenDir command, but it could equally well have been opened some other way.

The directory pathname must exactly match a leading sub-string of the title of a directory display for it to be closed. To avoid problems, your directory pathname should always include the filing system, the drive name and a full path from $. The case of letters is not significant, but the Filer uses lower case for filing system names.

This call must be able to close all directory displays that match the specified sub-string.

You can only use this command from within the desktop environment, or within a Desktop boot file.

Example

```
*Filer_CloseDir adfs::applDisc.$.progs.basic
```

Related commands

*Filer_OpenDir

Related SWIs

None

Related vectors

None

*Filer_OpenDir

Opens a directory display on the Desktop

Syntax

```
*Filer_OpenDir directory [x y [width height]] [switches]
```

Parameters

directory — the full pathname of a directory whose directory display is to be opened

x — the x-coordinate of the top left of the directory viewer, in OS units

y — the y-coordinate of the top left of the directory viewer, in OS units

width — the width of the directory viewer, in OS units

height — the height of the directory viewer, in OS units

switches — switches to control the display type of the directory viewer; the case of the letters is not significant:

-SmallIcons	display small icons
-LargeIcons	display large icons
-FullInfo	display full information
-SortByName	display sorted by name
-SortByType	display sorted by type
-SortByDate	display sorted by date
-SortBySize	display sorted by size

Use

*Filer_OpenDir opens a directory display on the Desktop.

If the directory pathname exactly matches the title of a directory display that is already open, it simply stays open; no new display appears. However, if the pathname is even slightly different from a display's title (eg you omit the $. after the drive name), it will be treated as a different directory. This can result in two displays looking at the same directory.

To avoid such problems, your directory pathname should always include the filing system, the drive name and a full path from $. The case of letters is not significant, but the Filer uses lower case for filing system names. This also ensures that applications run correctly, since they use their pathnames to reference files within themselves.

Each parameter – except for the switches – can be preceded by a keyword for the sake of clarity. This is especially useful when writing scripts. There are two variants on some keywords; again, the case of the letters is not significant. Valid keywords are:

Keyword	Alternative	Precedes parameter
`-dir`	`-directory`	*directory*
`-x0`	`-topleftx`	*x*
`-y0`	`-toplefty`	*y*
`-width`		*width*
`-height`		*height*

You can only use this command from within the desktop environment, or within a Desktop boot file.

Example

```
*Filer_OpenDir adfs::applDisc.$.progs.basic
```

Related commands

*Filer_CloseDir

Related SWIs

None

Related vectors

None

*Filer_Run

Runs a desktop application

Syntax

```
*Filer_Run application
```

Parameters

`application` a valid pathname specifying an application, the !Run file of which is to be run

Use

*Filer_Run runs the specified desktop application by running its !Run file. It is equivalent to double clicking on the application's icon. This command is most useful in Desktop boot files.

You can only use this command from within the desktop environment, or within a Desktop boot file.

Example

```
*Filer_Run adfs::mhardy.$.Apps.!PrinterPS
```

Related commands

*Filer_Boot

Related SWIs

None

Related vectors

None

42 Filer_Action

Introduction and Overview

This module performs file manipulation operations for the Filer without the desktop hanging whilst they are under way.

See the section entitled *Filer Action Window* on page 4-292 for details of how the Filer Action window operates.

SWI calls

FilerAction_SendSelectedDirectory (SWI &40F80)

Sends message specifying the selected directory

On entry

R0 = task handle to send the message
R1 = pointer to null terminated directory name

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sends the Wimp message Message_FilerSelectionDirectory (see page 4-293 for details of this message).

See the section entitled *Wimp_SendMessage* (SWI &400E7) on page 4-261 for a description of how messages within the Wimp environment are generated.

Related SWIs

FilerAction_SendSelectedFile (SWI &40F81)

Related vectors

None

FilerAction_SendSelectedFile (SWI &40F81)

Sends message specifying the selected files within a directory

On entry

R0 = task handle to send the message
R1 = pointer to null terminated selection name

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sends the Wimp message Message_FilerAddSelection (see page 4-293 for details of this message).

See the section entitled *Wimp_SendMessage* (SWI &400E7) on page 4-261 for a description of how messages within the Wimp environment are generated.

Related SWIs

FilerAction_SendSelectedDirectory (SWI &40F80)

Related vectors

None

FilerAction_SendStartOperation (SWI &40F82)

Sends message containing information to start operation

On entry

R0 = task handle to send the message
R1 = reason code:

0	Copy
1	Move (rename)
2	Delete
3	Set access
4	Set type
5	Count
6	Move (by copying and deleting afterwards)
7	Copy local (within directory)
8	Stamp files
9	Find file

R2 = option bits:

bit	meaning when set
0	Verbose
1	Confirm
2	Force
3	Newer (as opposed to just Look)
4	Recurse (only applies to access)

R3 = pointer to operation specific data
R4 = length of operation specific data:

Copy:
R3 pointer to name of destination directory (null terminated)
R4 length of name of destination directory (including null terminator)

Move:
R3 pointer to name of destination directory (null terminated)
R4 length of name of destination directory (including null terminator)

Delete:
R3 unused
R4 0

Set access:
R3 pointer to word containing required new access:
bottom two bytes indicate the values to set
top two bytes flag which bits are to be left alone
R4 sizeof (int)

Set type:
R3 pointer to word containing new type in bits 0-11
R4 sizeof (int)

Count:
R3 unused
R4 0

Copy Move:
R3 pointer to name of destination directory (null terminated)
R4 length of name of destination directory (including null terminator)

Copy Local:
R3 pointer to destination name (null terminated)
R4 length of name of destination name (including null terminator)

Stamp:
R3 unused
R4 0

Find:
R3 pointer to name of object to find (null terminated)
R4 length of name of object to find (including null terminator)

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sends the Wimp message Message_FilerAction (see page 4-293 for details of this message).

See the section entitled *Wimp_SendMessage* (SWI &400E7) on page 4-261 for a description of how messages within the Wimp environment are generated.

Related SWIs

None

Related vectors

None

* Commands

*Filer_Action

Used to start a Filer_Action task running under the desktop

Syntax

```
*Filer_Action
```

Parameters

None

Use

*Filer_Action is used to start a Filer_Action task running under the desktop. The task automatically sets its own slot size to an appropriate value. If it does not receive any relevant messages before the next null event, it kills itself.

This command is only useful to programmers writing applications to run under the desktop. To issue the command, you should call Wimp_StartTask (SWI &400DE) with R0 pointing to the string 'Filer_Action'. The reason why this command has to be provided is that it is only possible to start a new Wimp task using a * Command.

If you do try to use this command outside the desktop, the error `Wimp is currently active` is generated.

Related commands

None

Related SWIs

Wimp_StartTask (SWI &400DE)

Related vectors

None

43 Free

Introduction and Overview

This module enables an interactive free space display from the desktop.

Old filing systems can provide this new interactive style free space display by registering with this module.

FS_EntryFree

These are not filing system entry points…

FS_EntryFree 0

NoOp

On entry

—

On exit

—

Details

This entry point is a No Op, and you should just return.

FS_EntryFree 1

Gets device name

On entry

R0 = 1
R1 = filing system number
R2 = pointer to buffer
R3 = pointer to device name / id

On exit

R0 = length of name
R1-R3 preserved

Details

This entry point is called to get the name of a device.

FS_EntryFree 2

Gets free space for device

On entry

R0 = 2
R1 = filing system number
R2 = pointer to buffer
R3 = pointer to device name / id

On exit

R0 - R3 preserved

Details

This entry point is called to get the free space for a device. You should fill in the buffer with the following information:

Offset	Meaning
0	total size of device (0 if unchanged from last time read)
4	free space on device
8	used space on device

FS_EntryFree 3

Compares device

On entry

R0 = 3
R1 = filing system number
R2 = pointer to filename
R3 = pointer to device id
R6 = pointer to special field

On exit

R0 - R3, R6 preserved

Z set if R2 & R6 result in a file on the device pointed to by R3.

Details

This entry point is called to compare a device id with a filename and special field. This call can simply return with Z set if the filing system is a **fast** filing system (e.g. RAMFS).

SWI calls

Free_Register (SWI &

Provides an interactive free space display for a filing system

On entry

R0 = filing system number
R1 = address of routine to call to get free space info (see FS_*EntryFree* on page 3-488)
R1 - R12 on entry to the above routine

On exit

Registers preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This must be called in order for the free space module to provide an interactive free space display for a filing system. It is done automatically by the free space module for the following filing systems: ADFS, RamFS, NetFS, NFS, SCSIFS.

Related SWIs

None

Related vectors

None

Free_DeRegister

Removes the filing system from the list of known filing systems

On entry

R0 = filing system number

On exit

registers preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This removes the filing system from the list of known filing systems.

Related SWIs

None

Related vectors

None

* Commands

*ShowFree

Shows within a desktop window the amount of free space on a device

Syntax

```
*ShowFree -fs fs_name device
```

Parameters

fs_name	name of the filing system used to access the device
device	name of the device for which to show free space

Use

*ShowFree shows within a desktop window the amount of free space on a device.

Note: This command will only work on filing systems registered using Free_Register.

Example

```
*ShowFree -fs adfs HardDisc4
```

Related commands

None

Related SWIs

None

Related vectors

None