Part 4 – Using 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 (eg Acorn FileStore, SJ MDFS, Acorn Level 4 Fileserver)
- RAM Filing System (RamFS), for making memory appear to be a disc
- NetPrint, for printing using Econet printer servers (eg !Spooler).

Other modules provide extra filing systems:

- the DOSFS filing system provides access to MS-DOS format discs
- the ResourceFS filing system contains resource files needed by the Window manager and ROM-resident Desktop utilities
- the SystemDevices module and the device filing systems provide various system devices.

FileSwitch

A module called FileSwitch is at the centre of all filing system operation in RISC OS.

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 2-11.

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 1-201; 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 2-531.

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 2-197.

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 2-597.

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 may be added to a future release.

Finding out more…

For full details of DeviceFS, see the chapter entitled *DeviceFS* on page 2-429.

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 2-607.

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 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 *DOSFS* on page 2-323.

Adding image filing systems

You can add image filing systems to the system. For full details, see the chapter entitled *Writing a filing system* on page 2-531.

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 2-499.

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 and FilerSWIs* on page 2-513.

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:

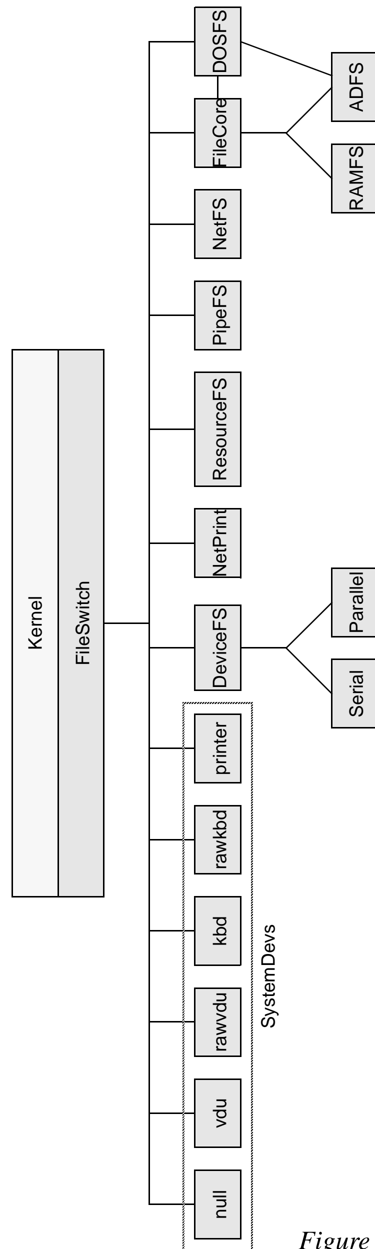


Figure 26.1 Structure of RISC OS 3 printing system

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 2-3.

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 1-201; 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 2-531.

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)

There is a subtle difference in wildcard matching between RISC OS 2 and later versions. Under RISC OS 2, commands acting only on files try to match wildcarded specifications against files only. However, under later versions these commands try to match against all objects; the first match found may be a directory, hence causing an error. (Similarly, a wildcarded specification passed to a command acting only on directories may get matched to a file.)

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::disc1.mike
    #MJHardy::disc1.mike
-net#MJHardy-:disc1.mike
   -#MJHardy-:disc1.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.

The system variable FileSwitch$SpecialField is also used to store the special field.

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.)

System variables

Some of these values are available in system variables under RISC OS 3. These are:

Variable	**Meaning**
FileSwitch$CurrentFilingSystem	current filing system
FileSwitch$TemporaryFilingSystem	temporary filing system
FileSwitch$*fs*$CSD	CSD for filing system *fs*
FileSwitch$*fs*$PSD	PSD for filing system *fs*
FileSwitch$*fs*$Lib	library for filing system *fs*

FileSwitch$*fs*$URD	URD for filing system *fs*
FileSwitch$SpecialField	special field, evaluated as path is processed

See also the section entitled *Using FileSwitch$SpecialField with path variables* on page 2-20.

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 (see the section entitled *File attributes* on page 2-347). 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	Owner execute only (BBC ADFS only), or Private (SJ Research file servers only)
3	Object is locked against deletion by you
4	Object has read access for others
5	Object has write access for others
6	Undefined
7	Object is locked against deletion for others

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 *xxx* is *LOADed or *RUN, FileSwitch looks for the variables Alias$@LoadType_*xxx* or Alias$@RunType_*xxx* 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 p1 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 p1 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" p1 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 Econet 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.

Using FileSwitch$SpecialField with path variables

FileSwitch$SpecialField is often used as part of a macro to define a path variable. For example, the default definition of Serial$Path is this macro:

```
devices#<FileSwitch$SpecialField>:$.Serial.
```

You could change this to set up default values for the serial port as follows:

```
devices#baud=9600,bits=8,<FileSwitch$SpecialField>:$.Serial.
```

Any settings passed to FileSwitch as a special field would then override the defaults in the definition of Serial$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 2-393 and *System devices* on page 2-495.

Filing system numbers

These are the currently allocated filing system numbers:

File system	Number
None	0
RomFS	3
NetFS	5
ADFS	8
NetPrint	12
Null	13
Printer	14
Serial	15
Vdu	17
RawVdu	18
Kbd	19
RawKbd	20
DeskFS	21
Computer Concepts RomFS	22
RamFS	23
RISC*i*XFS	24
Streamer	25
SCSIFS	26
Digitiser	27
Scanner	28
MultiFS	29
NFS	33
CDFS	37
DOSFS	43
ResourceFS	46
PipeFS	47
DeviceFS	53
Parallel	54

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 medium is missing.
3. An UpCall handler prints a message asking the user to supply the medium.
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 1-528 and page 1-908 respectively for details.

Support of calls

Some filing systems do not support all the commands that are detailed in this chapter, and you should be aware of this when writing code. In general, filing systems for handling mass-storage media will provide full support, whereas more esoteric filing systems may have omissions, mostly because a particular function is meaningless to that filing system. If you call an unsupported command, an error will be returned, and you should program to handle this.

Service Calls

Service_StartUpFS
(Service Call &12)

Start up filing system

On entry

R1 = &12 (reason code)
R2 = filing system number (see page 2-21)

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 2-96). 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 R2 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 call 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 2-75) 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 2-56), OS_Find (page 2-75)

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 2-178.

Related SWIs

OS_FSControl 10 (page 2-94)

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. It uses OS_FSControl 10 to do this, which you should use in preference to this obsolete call. 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 2-94), OS_FSControl 15 (page 2-99)

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	Page
0	Saves a block of memory as a file	2-35
1	Writes catalogue information for a named object	2-36
2	Writes load address only for a named object	2-36
3	Writes execution address only for a named object	2-36
4	Writes attributes only for a named object	2-36

R0	Action	Page
5	Reads catalogue information for a named object, using File$Path	2-38
6	Deletes a named object	2-40
7	Creates an empty file	2-41
8	Creates a directory	2-42
9	Writes date/time stamp of a named file	2-36
10	Saves a block of memory as a file, and date/time stamps it	2-35
11	Creates an empty file, and date/time stamps it	2-41
12	Loads a named file, using specified path string	2-43
13	Reads catalogue information for a named object, using specified path string	2-38
14	Loads a named file, using specified path variable	2-43
15	Reads catalogue information for a named object, using specified path variable	2-38
16	Loads a named file, using no path	2-43
17	Reads catalogue information for a named object, using no path	2-38
18	Sets file type of a named file	2-36
19	Generates an error message	2-45
20	Reads catalogue information for a named object, using File$Path	2-46
21	Reads catalogue information for a named object, using specified path string	2-46
22	Reads catalogue information for a named object, using specified path variable	2-46
23	Reads catalogue information for a named object, using no path	2-46
24	Reads the natural block size of a file	2-48
255	Loads a named file, using File$Path	2-43

For details of each of these reason codes, see below. Reason codes 20 - 24 inclusive are not supported by RISC OS 2.

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 load 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 2-41); 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; NetFS may generates an error if you try to write a directory's attributes, depending on the server you are using.

Under RISC OS 2 FileCore based filing systems do not generate an error if the object doesn't exist, whereas NetFS does so. Later versions of RISC OS always generate an error in these circumstances.

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	File$Path 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 2-18.

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
(R2 - R5 corrupted if object not found)

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.

The version of NetFS supplied in RISC OS 2 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.

The version supplied in RISC OS 3 does not behave like this.

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 2-35); 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	File$Path 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 2-18.

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 (0, 1, 2 or &100)

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 or 3*
`"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
```

R2 may only have the values given above; for other values, the call returns with all registers preserved and V set (ie no error is generated). RISC OS 3.00 does not support R2 = 3, although it can return an object type of 3 (an image file); you should be cautious in passing results from other calls directly to this call.

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 (bits 0 - 11)

Special	values:
–1	untyped (R2, R3 are load and execution address), or not found (R0 is 0)
&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 2-18.

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.

These calls are 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 is the same as the granularity of file allocation. For an example see the section entitled *Allocation bytes* on page 2-205, which gives a description of the granularity of FileCore based filing systems.

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 (see page 2-21), 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:

R0	Action	Page
0	Read pointer/FS number	2-51
1	Write pointer	2-52
2	Read extent	2-53
3	Write Extent	2-54
4	Read allocated size	2-55
5	Read EOF status	2-56

R0	Action	Page
6	Reserve space	2-57
7	Get canonicalised name	2-58
8	Inform of changed image stamp	2-59
254	Read information on file handle	2-60
255	Ensure file/files	2-62

Reason codes 7 and 8 are not available under RISC OS 2.

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). For a list of filing system numbers, see page 2-21.

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 an 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 1; 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
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 (ie prompting for input is appropriate)

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 10 shows when the stream is unbuffered; an unbuffered stream is one for which FileSwitch provides no buffering.

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 2-532.

This call indirects through ArgsV.

OS_Args 255 (SWI &09)

Ensure data has been written to 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 2-65), OS_GBPB (page 2-66)

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 2-63), OS_GBPB (page 2-66)

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	Page
1	Writes bytes to an open file using a specified file pointer	2-68
2	Writes bytes to an open file using the current file pointer	2-68
3	Reads bytes from an open file using a specified file pointer	2-69
4	Reads bytes from an open file using the current file pointer	2-69
5	Reads name and boot (*Opt 4) option of disc	2-70
6	Reads current directory name and privilege byte	2-70

R0	Action	Page
7	Reads library directory name and privilege byte	2-70
8	Reads entries from the current directory	2-71
9	Reads entries from a specified directory	2-73
10	Reads entries and file information from a directory	2-73
11	Reads entries and full file information from a directory	2-73
12	Reads entries and file type information from a directory	2-73

Reason code 12 is not available under RISC OS 2.

All OS_GBPB calls either complete successfully, or return an error; they do not partially transfer the group of bytes.

Related SWIs

OS_BGet (page 2-63), OS_BPut (page 2-65)

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 read. 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; under RISC OS 3 it always contains 0.
- 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>

The version of NetFS supplied with RISC OS 2 (5.46) pads disc and directory names to the right with spaces; other filing systems do not, including the version of NetFS supplied with RISC OS 3 (5.69 or later). 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 asked for but 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; see OS_Find 0 on page 2-77.

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 2-49), OS_BGet (page 2-63), OS_BPut (page 2-65) and OS_GBPB (page 2-66), and to eventually close the file using OS_Find 0. For full details of the reason codes to open files, see OS_Find 64 to 255 on page 2-78.

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

Bit 1	Bit 0	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 2-18.

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 2-49), OS_BGet (page 2-63), OS_BPut (page 2-65) and OS_GBPB (page 2-66), and to eventually close the file using OS_Find 0 (page 2-77).

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 FSCV.

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

R0	Action	Page
0	Set the current directory	2-83
1	Set the library directory	2-84
2	Inform of start of new application	2-85
3	Reserved for internal use	—
4	Run a file	2-87
5	Catalogue a directory	2-89

R0	Action	Page
6	Examine the current directory	2-90
7	Catalogue the library directory	2-91
8	Examine the library directory	2-92
9	Examine objects	2-93
10	Set filing system options	2-94
11	Set the temporary filing system from a named prefix	2-95
12	Add a filing system	2-96
13	Check for the presence of a filing system	2-97
14	Filing system selection	2-98
15	Boot from a filing system	2-99
16	Filing system removal	2-100
17	Add a secondary module	2-101
18	Decode file type into text	2-102
19	Restore the current filing system	2-103
20	Read location of temporary filing system	2-104
21	Return a filing system file handle	2-105
22	Close all open files	2-106
23	Shutdown filing systems	2-107
24	Set the attributes of objects	2-108
25	Rename objects	2-109
26	Copy objects	2-110
27	Wipe objects	2-113
28	Count objects	2-114
29	Reserved for internal use	—
30	Read location of secondary module for temporary filing system	2-115
31	Convert a string giving a file type to a number	2-116
32	Output a list of object names and information	2-117
33	Convert a file system number to a file system name	2-118
34	Reserved for future expansion	—
35	Add an image filing system	2-119
36	Image filing system removal	2-120
37	Convert a pathname to a canonicalised name	2-121
38	Convert file information to an object's file type	2-123
39	Set the User Root Directory (URD)	2-124
40	Exchange current and previous directories	2-125
41	Return the defect list for an image	2-126

R0	Action	Page
42	Map out a defect from an image	2-127
43	Unset the current directory	2-128
44	Unset the User Root Directory (URD)	2-129
45	Unset the library directory (Lib)	2-130
46	Return an image file's used space map	2-131
47	Read the boot option of the disc or image file that holds a specified object	2-132
48	Write the boot option of the disc or image file that holds a specified object	2-133
49	Read the free space on the disc or image file that holds a specified object	2-134
50	Name the disc or image file that holds a specified object	2-135
51	Request that an image stamp be updated	2-136
52	Find the name and type of object that uses a particular offset	2-137
53	Set a specified directory to a given path without verification	2-138
54	Read the path of a specified directory	2-140

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

Reason codes 35 upwards are not available under RISC OS 2.

Related SWIs

None

Related vectors

FSCV

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 filing 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 1-197), 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 1-266), 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 1-45); 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 1-326).

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 1-301).

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 1-301).
- 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 1-301).
- 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 1-301). 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 1-287 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 2-178) 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 1-961) 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 2-103).

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

R0 - R3 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 2-531.

OS_FSControl 13 (SWI &29)

Check for the presence of a filing system

On entry

R0 = 13
R1 = filing system number (see page 2-21), or pointer to filing system name
R2 = R1 dependent

On exit

R0 preserved
R1 = filing system number (see page 2-21), or preserved if not found
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 (see page 2-21); 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 (see page 2-21), 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 (see page 2-21). 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 2-149), *Configure NoBoot (page 2-152), and the *Opt commands (page 2-178).

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 2-96).

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 OS_ServiceCall &42 (see page 1-267). 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 2-116) 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 1-961) uses OS_FSControl 11 (see page 2-95)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 primary module for 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 primary module for the temporary filing system, and its private word. (For example, if ADFS were the temporary filing system, this call would return FileCore's address, whereas OS_FSControl 30 would return the address of ADFS – the secondary module. However, if NetFS were the temporary filing system, this call would return its address.)

If no temporary filing system is set, then this call reads the values for the current filing system instead. If there is no current filing system 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 2-532.

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
- dismounts 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 objects. 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 2-110).

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 objects. 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 2-110).

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 30 (SWI &29)

Read location of secondary module for temporary filing system

On entry

R0 = 30

On exit

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

Use

This call reads the location of the secondary module for the temporary filing system, and its private word. (For example, if ADFS were the temporary filing system, this call would return its address, whereas OS_FSControl 20 would return the address of FileCore – the primary module.)

If no temporary filing system is set, then this call reads the values for the current filing system instead. If there is no current filing system, or it does not have a secondary module, then R1 will be zero on exit, and R2 undefined.

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_333333`	Base 4 version of Text file type number

OS_FSControl 18 (see page 2-102) 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 2-168).

OS_FSControl 33 (SWI &29)

Converts a filing system number to a filing system name

On entry

R0 = 33
R1 = filing system number (see page 2-21)
R2 = pointer to buffer
R3 = length of buffer

On exit

Registers preserved

Use

This call converts the filing system number passed in R1 (see page 2-21) 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 2-531.

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
R3 = pointer to name of a path variable that contains a control-character terminated comma separated path string, or 0 if none
R4 = pointer to control-character terminated comma separated path string to use if variable not specified or non-existent, or 0 if none
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 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 1; 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 (entry R2 and R3 were 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 (URD)

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 (Lib)

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 ones.

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, and returns incorrect information for NetFS.

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 2-531 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 2-138). 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	Lock object against deletion by any user
W	Write permission for you
R	Read permission for you
/	Separator between your permissions and the public's
W	Write permission for the public (on NetFS)
R	Read permission for the public (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 read and write 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 on the current filing system. 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 is the only built-in filing system to use this)

Use

*CDir creates a directory with the specified pathname. On the NetFS, and on some third-party filing systems, 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, as a daughter to the current RAMFS directory*

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, no matter which filing system is the current one.

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

*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, *Type

*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 (see page 2-21)

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 1-361). 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 Copy$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 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 leafname of the destination must either be a specific filename, or the character '' in which case the destination will have the same leafname as the source. For example:

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

will copy all the files in the current directory with names beginning data to Dir2, preserving their leafnames.

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

  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

*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.

In RISC OS 2 this command is implemented by FileCore.

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

*Delete

Erases a single file or empty directory

Syntax

`*Delete` *`object_spec`*

Parameters

`object_spec`	a valid (wildcarded) pathname specifying a file or an empty directory

Use

*Delete erases the single named file or empty directory. An error message is given if the object does not exist, or is a directory containing files.

You may not use wildcards in the last component of the pathname.

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 by default, 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 by default) – 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_addr` + *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 1-908)

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 the contents of a file, numbering each line

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 the named file using the configured DumpFormat. Control F might be displayed as '|F', for instance.

Each line is numbered. For a similar display without line numbers added, use *Type.

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 by default).

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.

  If the corresponding Alias$@LoadType variable does not exist then a suitable error is generated.

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). This is shown as an ‘&’ in pathnames.

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.

Under RISC OS 3 this command may not work correctly, depending on the operation (loading is generally worse) and filing system (NetFS is poor).

*Opt 4

*Opt 4 sets the filing system boot option

Syntax

`*Opt 4 [[,]n]`

Parameters

n	0 to 3

Use

*Opt 4 sets the boot option for the current filing system. On filing systems with several media (eg ADFS using several discs) the boot option 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 usually named !Boot, although some filing systems may use different names; for example NetFS calls the file !ArmBoot (to avoid clashes with existing !Boot files that may contain code specific to BBC and Master series computers).

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 option to *Run for the current filing system*

Related commands

*Configure Boot, *Configure NoBoot

*Print

Displays the contents of a file as raw text on the screen

Syntax

`*Print` *`filename`*

Parameters

`filename`	a valid pathname specifying a file

Use

*Print displays the contents of the named 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

Erases a single file or empty directory

Syntax

`*Remove` *`object_spec`*

Parameters

`object_spec`	a valid (wildcarded) pathname specifying a file or an empty directory

Use

*Remove erases the single named file or empty directory. No error message is given if the object does not exist; this allows a program to remove a file without having to trap that error. However, an error message is given if the object is a directory containing files.

You may not use wildcards in the last component of the pathname.

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 1-287 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.

  If the corresponding Alias$@RunType variable does not exist then a suitable error is generated.

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 4-565, 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 specified 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 1-520), OS_Byte 199 (page 1-528), OS_File (page 2-32), OS_BPut (page 2-65)

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 1-520), OS_Byte 199 (page 1-528), OS_File (page 2-32), OS_BPut (page 2-65)

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 named file using the configured DumpFormat. Control F might be displayed as '|F', for instance.

For a similar display with line numbers added, use *List.

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 0 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 (URD)

Syntax

`*URD [directory]`

Parameters

directory	a valid pathname specifying a directory

Use

*URD sets the User Root Directory (URD). 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 (wildcarded) 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 2-3.

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 2-597.

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 2-202, the section entitled *New maps* on page 2-203, the section entitled *Directories* on page 2-211, and the section entitled *Boot blocks* on page 2-215.

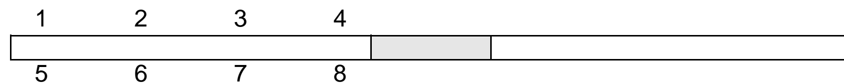
Physical layout

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

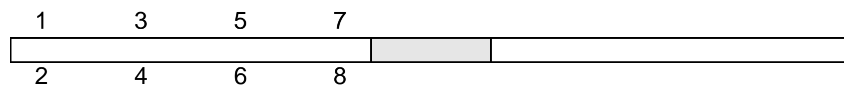
Format	Density	Sectors/track	Bytes/sector	Storage	Heads
L	Double	16	256	640K	1
D	Double	5	1024	800K	2
E	Double	5	1024	800K	2
F	Quad	10	1024	1.6M	2
Hard	—	—	—	≤512M	—

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

- On a sequenced disc the logical order of tracks is those on one side of the disc, followed by those on the other side. For example, with 8 tracks:

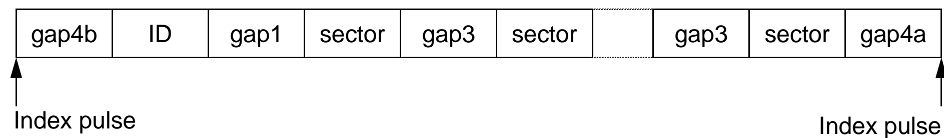


- On an interleaved disc the logical order of tracks alternates between sides of the disc. For example, with 8 tracks:



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
- *gap4a* 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:

sector ID	gap2	sector data

- *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 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 2-211.

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 2-208.

FreeLink is a fragment block giving the offset to the first free space fragment block in the allocation bytes; see page 2-206.

CrossChecks are combined to check that the whole map is self-consistent; see the section entitled *Calculating CrossCheck* on page 2-208.

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, disc is 40 track
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 FileCore_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.

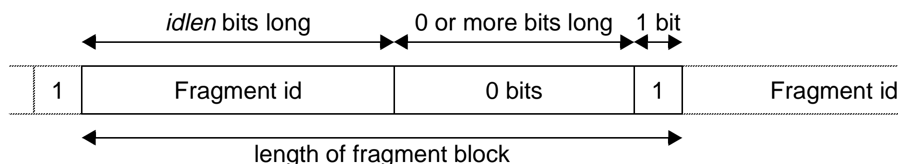
Three disc objects are special, and contain:

- 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 disc object. For a description of how disc objects can contain more than one object, see the section entitled *Internal disc addresses* on page 2-211 and the section entitled *Directories* on page 2-211.

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:



(*idlen* is defined in the disc record.)

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, and the 1 bit to terminate the fragment block.

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:

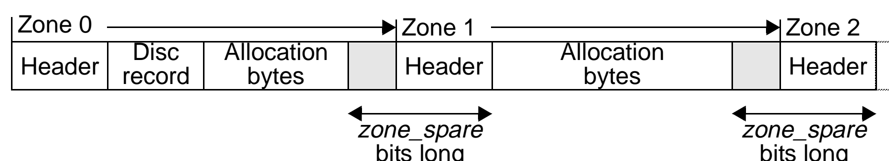
  log2secsize + 3   *ie* log_2 *(sector size in bits)*

  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 id*s 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 a copy of the disc record, 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 on 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:

address = ((t × H + h) × S + s – x) × 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 2-208, 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:

`ddd00000 0fffffff 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 *s*th 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’

BBC and Master series computers always use ‘Hugo’ for L-format discs; for compatibility, we suggest you do the same. For other formats you can use either.

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	

The *NewDirAtts* are as follows:

Bit	Meaning when set
0	Object has owner read access
1	Object has owner write access
2	Object is locked
3	Object is a directory
4	Object is executable †
5	Object has public read access
6	Object has public write access
7	Reserved (must be zero)

† Bit 4 is treated as a second owner read bit; if either this bit or bit 0 are set, the object is treated as having owner read access.

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 *xxx*DirLastMark 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 the fields they occupy (in which case there is no terminator).

The indirect disc address of an object on an old map disc is the most significant 3 bytes of its physical disc address. 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 2-210.

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	Defect list
&1BF downwards	Hardware-dependent information
&1C0 - &1FB	Disc record (see page 2-204)
&1FC - &1FE	Non-ADFS partition descriptor
&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 2-204.

The non-ADFS partition descriptor

These 3 bytes are used to describe any non-ADFS partition on the disc. Such a partition must come at the end of the disc, and is excluded from all descriptions of the ADFS partition. Currently it is only used to describe a RISC *i*X partition:

Offset	Contents	
&1FC	format identifier and flags:	
	bits 0 - 3	partition format identifier (1 ⇒ RISC *i*X)
	bits 4 - 7	flags (reserved – must be zero)
&1FD	low byte of start cylinder	
&1FE	high byte of start cylinder	

You can calculate the disc address of the start of the non-ADFS partition as follows:

start cylinder × heads on drive × sectors per track × bytes per sector

Calculating the boot block's checksum byte

The last byte of the boot block is a checksum 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 2-261). 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 1-183 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 `*Mount`ing 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.

See also *Service_DiscDismounted (Service Call &7D)* on page 2-506.

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 disc format

On entry

R1 = &69 (reason code)
R2 = pointer to buffer
R3 = length of buffer
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 2-204) 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 2-240); 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 2-223) 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.

The buffer pointed to by R2 should be filled in with a short description of the disc's format suitable for use in the **Current format** menu entry. You should ensure this does not overflow the length of the buffer (given in R3).

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
- Fill in buffer with description for **Current format** menu entry
- 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	

† These reason codes are only valid with the 1772 disc controller. They are not supported on 710/711 based machines (such as the A5000) and should be avoided for future compatibility.

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 bits 8 - 31 of R1 shifted left 6 bits (so they form bits 2 - 25 of the pointer).

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 timeout 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 9, 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 non-zero on entry, this reason code is used to write a track. This usage is specific to the 1772 disc controller.

If R3 is zero on entry, this reason code is instead used to format a track; R4 then points to a disc format structure. This usage is available with all controllers, but is not available under RISC OS 2.

The disc format structure pointed to by R4 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

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.

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 will 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 2-235).

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 2-597.

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, with drive 4 in the low byte, and drive 7 in the high byte. 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 (must be ≥ 4K long)
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 2-204.

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 2-227).

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 2-226). 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 (page 2-291), DOSFS_DiscFormat (page 2-335)

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 (terminated by –1)
R2 = pointer to null-terminated disc name
R3 = image 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	Page
0	Mount	2-240
1	Poll changed	2-240
2	Lock drive	2-240
3	Unlock drive	2-240
4	Poll period	2-240
5	Eject disc	2-240

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 2-204).

- 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 2-597), 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 = pointer to disc name (may not be terminated if maximum length)
R8 = pointer to FileCore instance private word

On exit

R5 = minimum polling period (in centiseconds), 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 UpCalls MediaNotPresent and MediaNotKnown. The value applies to all drives rather than a particular drive.

FileCore_MiscOp 5 (SWI &40549)

Power-ejects the disc in the specified drive

On entry

R0 = 5
R1 = drive
R8 = pointer to FileCore instance private word

On exit

—

Use

This call power-ejects the disc in the specified drive, provided that the hardware is capable of it.

This reason code was introduced in RISC OS 3 (version 3.10).

* 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.

In doing so, it closes all files on the disc.

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 or F 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

*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, and ideally you should not use it.

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 after RISC OS 2, 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

**Verify*

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 (on page 2-11 and page 2-197 respectively) 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* on page 2-197 for details.

It provides:

- a * Command to select itself (*ADFS)
- a * Command to format discs (*Format)
- various configure options, accessed using *Configure
- SWIs that give access to corresponding FileCore SWIs
- 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 2-199 onwards of the chapter entitled *FileCore*.

Formatting discs

If you are running a site with a mixture of 1772-equipped ‘old’ machines and 710/711-equipped ‘newer’ machines, we recommend that you format all discs on the latter.

On old machines, D and E format discs have the sectors offset between sides for speed optimisation. The 710/711 cannot format discs in this manner, and may run slow when accessing such discs. By formatting discs on newer machines, they will run at the same speed on every machine, albeit some 5% slower than discs with offset sectors can run on older machines.

Likewise, we recommend that any software you ship uses discs that do not offset sectors between sides (ie the discs are formatted on a newer machine).

Software protection schemes

If you wish to vary the format of a disc to provide software protection, you should follow the guidelines below. This will ensure that your discs are reliably readable and quick to load on all RISC OS machines, current or planned.

Disc formats should conform to the specifications in the chapter entitled *FileCore* on page 2-197, with some exceptions. You may:

- use different size sectors within any one track
- arbitrarily vary the ID held in the sector ID, within the limits imposed by the 1772 disc controller (but you must then use the altered ID to access that sector – see below).

You may not:

- directly access hardware
- vary the data rate or encoding method within a single track
- rely on the contents or operation of system data areas (eg 0 - &8000) or FIQ routines
- access sectors specifying a different ID to that physically held in the sector ID.

The last point prohibits such common practices as reading a 1k sector with a 2k read (to recover inter-sector data), or reading a track with a different head number to that in the sector ID (which works with a 1772, but fails with the 710/711 used on machines such as the A5000).

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 specification 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 format

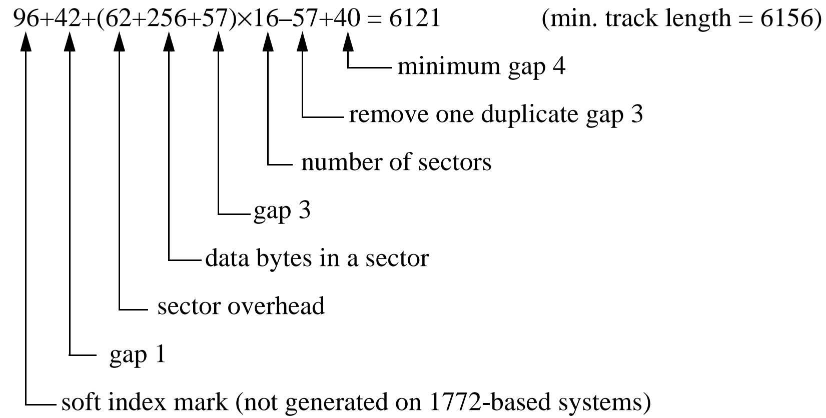


Figure 29.1 Byte usage for a track: L format

D and E formats

1772-based system without index mark:

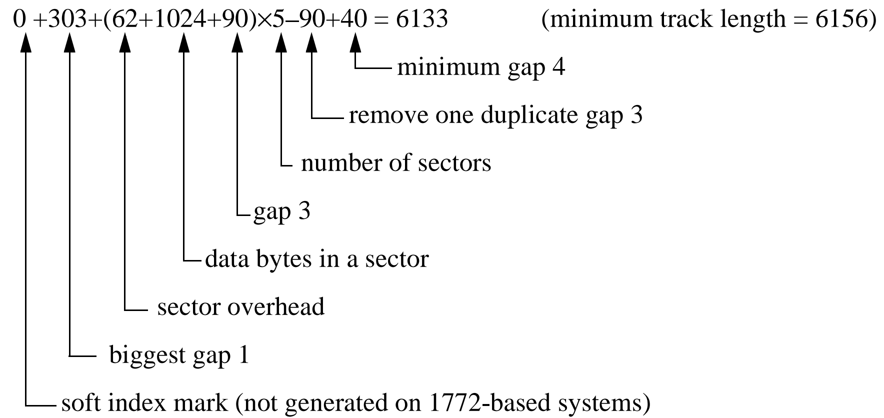


Figure 29.2 Byte usage for a track: D and E formats (no index mark)

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)

- 40: minimum gap 4
- 90: remove one duplicate gap 3
- 5: number of sectors
- 90: gap 3
- 1024: data bytes in a sector
- 62: sector overhead
- 50: biggest gap 1
- 96: soft index mark (not generated on 1772-based systems)

Figure 29.3 Byte usage for a track: D and E formats (index mark)

F format

96+50+(62+1024+90)×10–90+30 = 11846 (min. track length = 12312)

- 30: minimum gap 4
- 90: remove one duplicate gap 3
- 10: number of sectors
- 90: gap 3
- 1024: data bytes in a sector
- 62: sector overhead
- 50: biggest gap 1
- 96: soft index mark (not generated on 1772-based systems)

Figure 29.4 Byte usage for a track: F format

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)

- 9: data size (256+40) × 3%
- 40: write-to-read time

Figure 29.5 Minimum Gap3 size: L format

D and E formats

Min. gap 3 = 32 + 40 = 72 (actually 90)

- 32: data size (1024+40) × 3%
- 40: write-to-read time

Figure 29.6 Minimum Gap3 size: D and E formats

F format

Min. gap 3 = 32 + 43 = 75 (actually 90)

- 32: data size (1024+40) × 3%
- 43: write-to-read time

Figure 29.7 Minimum Gap3 size: F format

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

Figure 29.8 Worst write to read time: L format

D and E formats

Worst write-read-time = (90–32)×32 = 1856 µS

Figure 29.9 Worst write to read time: D and E formats

F format

Worst write-read-time = (90–32)×16 = 928 µS

Figure 29.10 Worst write to read time: F format

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	≥10	≥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 $3^1/_2$"/$5^1/_4$" 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 $3^1/2$" systems or 150Ω in older $5^1/4$" 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 $5^1/4$" 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

Using filing systems

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 are:

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 disc format name

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 2-313).

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 &44B00)* on page 2-335. Similarly, for an example of a call used to lay down a disc structure, see *DOSFS_LayoutStructure (SWI &44B01)* on page 2-338.

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 density 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 (page 2-223)

On exit

See FileCore_DiscOp (page 2-223)

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 (page 2-223), after first setting R8 to point to the FileCore instantiation private word for ADFS.

This call is functionally identical to FileCore_DiscOp.

Related SWIs

FileCore_DiscOp (page 2-223)

Related vectors

None

ADFS_HDC (SWI &40241)

Sets the address of an alternative ST506 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 ST5056 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 controller must be an HD63463 (or compatible).

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 (page 2-230)

On exit

See FileCore_Drives (page 2-230)

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 (page 2-230), after first setting R8 to point to the FileCore instantiation private word for ADFS.

This call is functionally identical to FileCore_Drives.

Related SWIs

FileCore_Drives (page 2-230)

Related vectors

None

ADFS_FreeSpace (SWI &40243)

Calls FileCore_FreeSpace

On entry

See FileCore_FreeSpace (page 2-231)

On exit

See FileCore_FreeSpace (page 2-231)

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 (page 2-231), after first setting R8 to point to the FileCore instantiation private word for ADFS.

This call is functionally identical to FileCore_FreeSpace.

Related SWIs

FileCore_FreeSpace (page 2-231)

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 (page 2-234)

On exit

See FileCore_DescribeDisc (page 2-234)

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 (page 2-234), after first setting R8 to point to the FileCore instantiation private word for ADFS.

This call is functionally identical to FileCore_DescribeDisc.

Related SWIs

FileCore_DescribeDisc (page 2-234)

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 *ImageFS*_DiscFormat (ie drive 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 *ImageFS*, in response to ADFS calling FileCore_DiscFormat (page 2-236) or *ImageFS*_DiscFormat (eg *DOSFS_DiscFormat (SWI &44B00)* on page 2-335) 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 drive 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 drive

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 drive.

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, R3 preserved
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_MiscOp (SWI &4024C)

Calls FileCore_MiscOp

On entry

See FileCore_MiscOp (page 2-240)

On exit

See FileCore_MiscOp (page 2-240)

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_MiscOp (page 2-240), after first setting R8 to point to the FileCore instantiation private word for ADFS.

This call is functionally identical to FileCore_MiscOp.

Related SWIs

FileCore_MiscOp (page 2-240)

Related vectors

None

Using filing systems

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 early models of RISC OS computers (eg the Archimedes 300, 400 and 500 series, and the A3000). More recent models (eg the A5000) 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 (eg the A5000). 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
```

Using filing systems

*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
	DOS/Q	1.44M	MS-DOS 3.20	double sided HD 3 1/2" disc
	DOS/M	720K	MS-DOS 3.20	double sided 3 1/2" disc
	DOS/H	1.2M	MS-DOS 3	double sided HD 5 1/4" disc
	DOS/N	360K	MS-DOS 2, 3	double sided 3 1/2", 5 1/4" disc
	DOS/P	180K	MS-DOS 2, 3	single sided 5 1/4" disc
	DOS/T	320K	MS-DOS 1, 2, 3	double sided 5 1/4" disc
	DOS/U	160K	MS-DOS 1, 2, 3	single sided 5 1/4" disc
	Atari/M	720K	Atari ST	double sided 3 1/2" disc
	Atari/N	360K	Atari ST	single sided 3 1/2" disc
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 DOS/H, DOS/Q and F formats. RISC OS 2 only supports L, D and E formats. Newer models of RISC OS 3 computers (eg the A5000) can use all the above 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).

Examples

`*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 (on page 2-11 and page 2-197 respectively) 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* on page 2-197 for details.

It provides:

- a * Command to select itself (*RamFS)
- 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 (page 2-223)

On exit

See FileCore_DiscOp (page 2-223)

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 (page 2-223), after first setting R8 to point to the FileCore instantiation private word for RamFS.

This call is functionally identical to FileCore_DiscOp.

Related SWIs

FileCore_DiscOp (page 2-223)

Related vectors

None

RamFS_Drives (SWI &40782)

Calls FileCore_Drives

On entry

See FileCore_Drives (page 2-230)

On exit

See FileCore_Drives (page 2-230)

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 (page 2-230), after first setting R8 to point to the FileCore instantiation private word for RamFS.

This call is functionally identical to FileCore_Drives.

Related SWIs

FileCore_Drives (page 2-230)

Related vectors

None

RamFS_FreeSpace (SWI &40783)

Calls FileCore_FreeSpace

On entry

See FileCore_FreeSpace (page 2-231)

On exit

See FileCore_FreeSpace (page 2-231)

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 (page 2-231), after first setting R8 to point to the FileCore instantiation private word for RamFS.

This call is functionally identical to FileCore_FreeSpace.

Related SWIs

FileCore_FreeSpace (page 2-231)

Related vectors

None

RamFS_DescribeDisc (SWI &40785)

Calls FileCore_DescribeDisc

On entry

See FileCore_DescribeDisc (page 2-234)

On exit

See FileCore_DescribeDisc (page 2-231)

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 (page 2-231), after first setting R8 to point to the FileCore instantiation private word for RamFS.

This call is functionally identical to FileCore_DescribeDisc.

Related SWIs

FileCore_DescribeDisc (page 2-231)

Related vectors

None

* Commands

*Configure RamFsSize

Sets the configured amount of memory reserved for the RAM filing system

Syntax

```
*Configure RamFSSize mK|n
```

Parameters

*m*K	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_ChangeDynamicArea (page 1-384), OS_ReadRAMFsLimits (page 1-390)

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.

DOSFS is not available in RISC OS 2.

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:

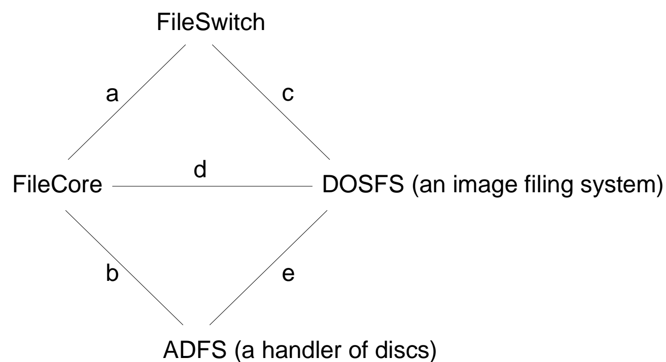


Figure 31.1 Interactions of DOSFS with other filing systems

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:

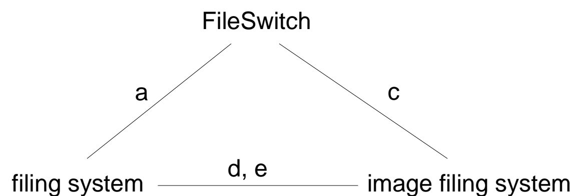


Figure 31.2 Interactions of complete image filing system with other filing systems

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 the chapter entitled *Writing a filing system* on page 2-531.

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 2-531.

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 a 'Disc not understood – has it been formatted?' 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 2-220) 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 2-135), 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 2-136). 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 2-62), and to inform their host filing system whenever a disc cycle id has changed **for whatever reason** using OS_Args 8 (see page 2-59).

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 2-282). 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 2-281), 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, used to specify 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, used to specify the structure.

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 2-504). 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 2-335), 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 2-291) 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 the desktop formatter. Both methods use ADFS_DiscOp (see page 2-283) 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 the desktop formatter).

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.

Filename mapping

Filenames are mapped between RISC OS and DOS filenames as follows:

From RISC OS to DOS

The RISC OS filename is truncated to 8 characters. Some characters having special meaning are changed:

RISC OS	DOS
#	?
?	#
+	&
=	@
;	%
<	$
>	^

Note that the first mapping shown above is unlikely to occur in practice, since ‘#’ is a wild card in RISC OS, and ‘?’ a wildcard in DOS. In practice, we recommend that you use alphanumeric filenames where possible.

Filename extensions

DOSFS provides the *DOSMap command (page 2-342) with which you can set up mappings between RISC OS filetypes and DOS filename extensions.

When transferring a file to DOS, the RISC OS filetype is checked against any that have been registered using *DOSMap; if there is a match the DOS file is given the corresponding filename extension.

From DOS to RISC OS

If the DOS filename has an extension, the separator is changed from ‘.’ to ‘/’. Characters having special meaning are changed, as above. RISC OS is then passed the filename, concatenated with the (changed) separator and extension. This may be up to 12 characters in total; the *Configure Truncate command (page 2-153) controls how RISC OS copes with this. By default, filing systems will typically handle this from the command line or in program interfaces, but their desktop filers will truncate the names.

Setting file types

When transferring a file to RISC OS, the DOS filename extension is checked against any that have been registered using *DOSMap; if there is a match the RISC OS file is given the corresponding file type. Otherwise the file type is set to ‘DOS’ (&FE4).

SWI numbering

Under RISC OS 3 (version 3.00) DOSFS had a SWI chunk base number of &41AC0; all subsequent versions have a SWI chunk base number of &44B00. If you are writing software that calls DOSFS_DiscFormat or DOSFS_LayoutStructure (the only two SWIs present in 3.00), and wish it to work under RISC OS 3 (version 3.00), you must either call the SWIs by name, using OS_SWINumberFromString (page 1-474) to convert the name at run time; or you must call the SWIs by different numbers depending on which version of RISC OS you are running under.

An alternative is to refuse to run under RISC OS 3 (version 3.00), giving a suitable error.

Warning: possible data corruption

There is a bug in DOSFS which can cause data corruption under the following circumstances:

- You write < 256 bytes to the start of a *cluster*.

  (A *cluster* is a technical term used in MS-DOS for a group of sectors, the number of which is format dependent. Note that files always start at the start of a cluster.)
- The last write before closing a file is later in the same cluster.

There are two possible workrounds:

1. If writing < 256 bytes that may be at the start of a cluster, use OS_Args 255 (page 2-62) to flush the data to disc before any subsequent writes.
2. Always build data in structures of > 256 bytes before writing it.

SWI Calls

DOSFS_DiscFormat (SWI &44B00)

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	DOS/Q	1.44M	MS-DOS 3.20	double sided
1	DOS/M	720K	MS-DOS 3.20	double sided
2	DOS/H	1.2M	MS-DOS 3	double sided
3	DOS/N	360K	MS-DOS 2, 3	double sided
4	DOS/P	180K	MS-DOS 2, 3	single sided
5	DOS/T	320K	MS-DOS 1, 2, 3	double sided
6	DOS/U	160K	MS-DOS 1, 2, 3	single sided
7	Atari/M	720K	Atari	double sided
8	Atari/N	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 2-226). The differences are because the DiscOp structure only specifies the format of a single track.

Under RISC OS 3 (version 3.00) this SWI had the number &41AC0.

Related SWIs

ADFS_VetFormat (page 2-291), FileCore_DiscFormat (page 2-236)

Related vectors

None

DOSFS_LayoutStructure (SWI &44B01)

Lays out into the specified image a set of structures for its format

On entry

R0 = structure specifier
R1 = pointer to list of bad blocks (terminated by –1)
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 structure specifiers are recognised:

Value	Meaning			
0	DOS/Q	1.44M	MS-DOS 3.20	double sided
1	DOS/M	720K	MS-DOS 3.20	double sided
2	DOS/H	1.2M	MS-DOS 3	double sided
3	DOS/N	360K	MS-DOS 2, 3	double sided
4	DOS/P	180K	MS-DOS 2, 3	single sided
5	DOS/T	320K	MS-DOS 1, 2, 3	double sided
6	DOS/U	160K	MS-DOS 1, 2, 3	single sided
7	Atari/M	720K	Atari	double sided
8	Atari/N	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 a simple table index for this; other image filing systems may pass different information (using a pointer if necessary) for their LayoutStructure SWI.

Under RISC OS 3 (version 3.00) this SWI had the number &41AC1.

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.

DOSFS currently writes an MS-DOS 3.30 boot-block onto discs that it formats. If you wish to use a different boot block you need a floppy disc containing that boot block (from another system). You can then use this command to overwrite the MS-DOS 3.30 boot-block with your other boot block.

DOSFS does not place the system files on a disc, so it cannot be used to boot-strap an MS-DOS system or the PC-Emulator. To make a DOSFS disc bootable you need to use this command to copy a bootable boot block to the disc, and also need to copy a suitable set of system files to the disc.

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 ‘DOS’.

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.

The mappings are only retained until the next reset.

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 Econet and file server protocols, you should see:

- the chapter entitled *Econet* on page 2-619
- the chapter entitled *File server protocol interface* on page 2-705.

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. Three commands are to do with selecting file servers: *AddFS, *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.

File server name binding

NetFS allows you to refer to file servers by name(s); these are the names of the discs on that particular file server. Inside NetFS a name is always reduced to a station and net number pair (since this is what the Econet interfaces require). To help NetFS make this translation (or binding) between names and numbers it keeps a list (or cache) of the names of the discs on various file servers.

NetFS uses this list when the file server argument for a *Logon command is a name rather than a number. It is also the list you see when you type *ListFS.

The list is generated by broadcasting a request to all file servers to send back the names of all their discs. When a name is looked up (or bound) the list is searched; if the name is present the number is returned, if not a broadcast is issued and the list is searched again. NetFS expects that every disc on every file server will have a different name; this is important, because NetFS needs a one-to-one mapping from names to station and net numbers.

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 chapter entitled *File server protocol interface* on page 2-705.
- 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.

Differences from FileCore based filing systems

Because NetFS does not use FileCore, there are a number of subtle differences between it and FileCore based filing systems. For example, because of the file server protocols it uses (see the chapter entitled *File server protocol interface* on page 2-705) NetFS can only update a file’s datestamp if it is passed a filename rather than a file handle.

You must not assume that the behaviour of all filing systems will be identical to ones that use FileCore.

File attributes

NetFS uses the top 24 bits of to store a file's creation/modification date in the following format:

Bits	Meaning
8 - 12	Day of month (1 - 31)
13 - 15	High bits of year (offset from 1980, 0 - 127)
16 - 19	Month of year (1 - 12)
20 - 23	Low bits of year (offset from 1980, 0 - 127)

With the addition of three zero bytes, these are in the correct format to use as input to the SWI NetFS_ConvertDate (page 2-364).

Service Calls

Service_NetFS
(Service Call &55)

Either a *Logon, a *Bye or a *SDisc/*Mount 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 may have logged 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. If your current file server is not set then this call returns zero for both the net and station number.

Related SWIs

NetFS_SetFSNumber (page 2-351), NetFS_ReadFSName (page 2-352)

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

R0, R1 corrupted

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, restoring any context (for example its current directory).

This is the same as *FS *net.station*

Related SWIs

NetFS_ReadFSNumber (page 2-350), NetFS_SetFSName (page 2-353)

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 if you are logged on, and otherwise returns a null string.

Related SWIs

NetFS_ReadFSNumber (page 2-350), NetFS_SetFSName (page 2-353)

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, restoring any context such as your current directory. You must be logged on to the file server; if you are not, an error is generated.

Related SWIs

NetFS_SetFSNumber (page 2-351), NetFS_ReadFSName (page 2-352)

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. It will be removed from future versions of NetFS, and you must not use it.

Related SWIs

NetFS_SetCurrentContext (page 2-355)

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. It will be removed from future versions of NetFS, and you must not use it.

Related SWIs

NetFS_ReadCurrentContext (page 2-354)

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 (page 2-357)

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

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 sets the current values for timeouts used by NetFS when communicating with the file server.

Related SWIs

NetFS_ReadFSTimeouts (page 2-356)

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 chapter entitled *File server protocol interface* on page 2-705, 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 (page 2-366)

Related vectors

None

NetFS_EnumerateFSList (SWI &40049)

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	Net number
2	Drive number
3	Disc name, padded with spaces
19	Zero

They are returned in alphabetical order.

This call disables the event process that updates the list, so that it does not change during enumeration. After you have completed the enumeration you **must** restart the event process by calling NetFS_EnableCache (page 2-375).

Related SWIs

NetFS_EnumerateFS (page 2-362), NetFS_EnableCache (page 2-375)

Related vectors

None

NetFS_EnumerateFS (SWI &4004A)

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	Net number
2	Zero
3	Disc name, padded with spaces
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_EnumerateFSList (page 2-360), NetFS_EnumerateFSContexts (page 2-370)

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 (page 1-447),
OS_ConvertDateAndTime (page 1-449)

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 = net 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 chapter entitled *File server protocol interface* on page 2-705, 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 (page 2-358)

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 = net 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.

This call is not available in RISC OS 2.

Related SWIs

NetFS_EnumerateFS (page 2-362), NetFS_EnableCache (page 2-375)

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	Net number
2	Reserved
3	Disc name padded to 16 characters with spaces
19	Zero
20	User name padded to 21 characters with spaces
41	Zero
42	Reserved
43	Reserved

This call is not available in RISC OS 2.

Related SWIs

NetFS_EnumerateFSList (page 2-360), NetFS_EnumerateFS (page 2-362)

Related vectors

None

NetFS_ReadUserId (SWI &4004F)

Returns the current user ID if logged on to the current file server

On entry

R1 = pointer to buffer
R2 = number of bytes in the buffer

On exit

R0 corrupted
R1 = pointer to terminating zero
R2 = space remaining in buffer (including 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 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).

This call is not available in RISC OS 2.

Related SWIs

NetFS_ReadFSNumber (page 2-350), NetFS_ReadFSName (page 2-352)

Related vectors

None

NetFS_GetObjectUID (SWI &40050)

Gets a unique identifier for an object

On entry

R1 = pointer to a canonical object name
R6 = pointer to a canonical 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: see page 2-558) 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.

For information on canonical; file names, see FSEntry_Func 23 (page 2-578).

This call is not available in RISC OS 2.

Related SWIs

OS_File (page 2-32)

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.

This call is not available in RISC OS 2.

Related SWIs

NetFS_EnumerateFS (page 2-362), NetFS_UpdateFSList (page 2-368)

Related vectors

None

* Commands

*AddFS

Adds a remote file server's disc to the list of known file servers' discs

Syntax

`*AddFS` *`file_server_number`* `[`*`disc_number`* `[:]`*`disc_name`*`]`

Parameters

file_server_number	the file server number of the file server to add, which must be a full address including both a net number and a station number
disc_number	the disc number of the disc to add
disc_name	the disc name of the disc to add

Use

*AddFS adds a remote file server's disc to the list of file servers' discs that are known to NetFS. If only the file server is specified, then all its discs will be removed from the list.

NetFS updates the list as necessary for file servers to which it can broadcast. This command is useful for you to add file servers to which NetFS cannot normally broadcast: for example ones located over a wide area network link.

This command is not available in RISC OS 2, nor is it in RISC OS 3 (version 3.00).

Example

`*AddFS 201.254 4 :Server`	*Add disc 'Server' on drive 4 of file server 201.254 to list*
`*AddFS 202.254`	*Remove all entries for file server 202.254 from list*

Related commands

None

*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
```

Related commands

None

*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
```

Related commands

None

*FS

Restores the file server's previous context

Syntax

`*FS [[:]file_server]`

Parameters

file_server	the file server name or number

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 implemented as an alias using the system variable Alias$I. It 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 [-force]
```

Parameters

`-force`	force the list to be updated before it is displayed

Use

*ListFS displays a list of the file servers which NetFS is able to recognise. The optional argument forces the list to be updated before it is displayed.

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 file server

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 file server, 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. The maximum password length is file server dependent: on Level 4 file servers it is 22 characters, whereas on earlier file servers it is only 6 characters.

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 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 net 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 Econet protocols, you should see the chapter entitled *Econet* on page 2-619.

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*.

Selecting a printer server

Whenever you open or save a file with NetPrint the software needs to know which printer server to send your data to. When you have only a single printer server on the network you should use *Configure PS to set its station number as the default. Then when you use the filename `NetPrint:` your printout will be sent to the correct station.

Some printer servers and spoolers support a naming protocol which allows you to refer to a particular printer by name rather than by number. Names can be up to six characters in length and are usually alphanumeric: for example Epson, Art, CDT, Laser1, Gerald, Draft, and PScrpt. It is sensible to choose a consistent set of names, based on either location, type, brand or class. Before NetPrint can use a named printer server, it must resolve the name to a station number; this process is called name binding. Put simply, the name binder broadcasts the name, and returns the number of the first server that says it is ready to accept a connection. If no suitable reply occurs within a specified time an error is returned.

NetPrint has the notion of the ‘current printer server’. This is usually set by the *Configure PS *printer_server* command, or with the *PS *printer_server* command. If the *printer_server* is given by name, then name binding occurs; it is the returned number that is retained as the current setting. Using *PS *printer_server* will cause the binding to occur immediately and the result to be known. Once the number is selected, it will be used whenever you open or save a file with the name `NetPrint:`.

When your network has more than one printer server (or a spooler that is more than one server) you may wish to choose which server to use. The easiest is to set the name of the server as the configured default using *Configure PS *printer_server*.

It is always possible to override the current setting by supplying the name or number of the server you wish to use as part of the filename. For example you might specify a server by number thus:

`NetPrint#233:`	*station number only (on current net)*
`NetPrint#2.253:`	*full net.station address*

or you might specify it by name (which would then be bound) thus:

```
NetPrint#Daisy:
NetPrint#Epson:
```

When selecting a particular printer server by this method the ‘current printer server’ remains unaffected.

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 within that time no printer server with that name has responded, or all those that did were busy, 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. If the current printer server is only stored as a name (eg after *SetPS *printer_server_name*) then zero is returned for both the net and station numbers.

Related SWIs

NetPrint_SetPSNumber (page 2-398), NetPrint_ReadPSName (page 2-399)

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

R0, R1 preserved

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 (page 2-397), NetPrint_SetPSName (page 2-401)

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. If the current printer server is only stored as a number (eg after *SetPS *printer_server_number*) then a null name is returned.

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 (page 2-397), NetPrint_SetPSName (page 2-401)

Related vectors

None

NetPrint_SetPSName (SWI &40203)

Sets by name the printer server used as your current one

On entry

R0 = pointer to buffer containing null-terminated printer server name

On exit

R0 preserved

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 (page 2-398), NetPrint_ReadPSName (page 2-399)

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 (page 2-403)

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 (page 2-402)

Related vectors

None

NetPrint_BindPSName (SWI &40206)

Converts a printer server's name to its address, providing it is free

On entry

R0 = pointer to buffer containing null-terminated printer server name

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 not re-entrant

Use

This call attempts to find a printer server with the specified name that is not busy. If one is found its address is returned in R0 and R1; otherwise an error is returned.

This call is not available in RISC OS 2.

Related SWIs

None

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 not 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	net 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	net number
2	status
3	station number for status (optional)
4	net 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 open'

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 station and net numbers (at offsets 3 and 4) are set to zero.

This call is not available in RISC OS 2.

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 net 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.

This call is not available in RISC OS 2.

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.

The stored name or station number is only bound when the default printer is first used (ie when NetPrint: is first used, without any special fields to specify a server other than the default).

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.

This command is identical to the command:

```
*Cat NetPrint:
```

or, with the `-full` parameter, to the command:

```
*Ex NetPrint:
```

Example

```
*ListPS -full
Umber    46.235  ready
Jade     44.235  ready
Mauve    93.235  ready
White    59.235  ready
Coral    32.235  jammed
Lime      2.235  ready
```

Related commands

*Configure PS, *PS, *SetPS

*PS

Changes the current printer server

Syntax

```
*PS [printer_server]
```

Parameters

printer_server	the name or station number of the printer server

Use

*PS changes the current printer server. The new printer server will be used next time you print to the default net printer. If the server is specified by name, it is immediately bound, and the current server set to the returned number; if the server is specified by number, the current server is set to that number.

If you don't specify a printer server, then this command returns the current printer server and its status.

Example

```
*PS 49.254
```

```
*PS myPS
```

```
*PS
Printer server myPS (9.235) is ready
```

Related commands

*Configure PS, *ListPS, *SetPS

*SetPS

Changes the current printer server

Syntax

`*SetPS [`*`printer_server`*`]`

Parameters

printer_server the name or station number of the printer server.

Use

*SetPS changes the current 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.

If you don't specify a printer server, then this command sets the current printer server to be the default printer server (as set by *Configure PS).

Example

```
*SetPS 49.254
```

```
*SetPS myPS
```

```
*SetPS
```

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 1-191) 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 1-192) 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.

Before attempting to read data from a pipe you must first ensure that it contains data. The recommended way to do this is to call OS_GBPB 10 (page 2-73).

* Commands

*PipeCopy

Copies a file one byte at a time to one or two output files

Syntax

`*PipeCopy` *`source_file destination_file1`* `[`*`destination_file2`*`]`

Parameters

source_file	a valid pathname specifying a source file
destination_file1	a valid pathname specifying a first destination file
destination_file2	a valid pathname specifying a second (optional) destination file

Use

*PipeCopy copies a file one byte at a time to one or two output files.

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 sprite 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. Wimp$Path can be set up to reference a resource directory).

ResourceFS is not available in RISC OS 2.

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
`$.Resources.`*`modulename`* ; resources for system modules
`$.Resources.`*`appname`* ; resources for applications

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 4-551.

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 Filer 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:$.Resources.Wimp.Messages
Resources:$.Resources.Wimp.Sprites
Resources:$.Resources.Wimp.Templates
Resources:$.Resources.Wimp.Tools
```

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 1-361), by issuing *Filer_Run commands 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 is not auto-started, since it has no iconbar icon of its own; instead it is 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.

Internationalisation

The ROM applications do not contain the entire application, but simply the !Boot, !Help and !Run 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).

Because each application uses a path variable to access its resource files, you can copy it to disc and add an updated copy of the ‘Messages’ file to the application directory. This will take precedence over the version in the ROM directory, which is accessed via the second path element.

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 (as returned by OS_File 5)
12	4	size of file (as returned by OS_File 5)
16	4	attributes of file (as returned by OS_File 5)
20	*n*	full filename, excluding '$.', 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)

The file structure inside ResourceFS has changed

On entry

R1 = &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

R1 = &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 2-418)
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 ResourceFS know about it.

SWI Calls

ResourceFS_RegisterFiles (SWI &41B40)

Add file(s) to the ResourceFS structure

On entry

R0 = pointer to resource file data (see page 2-418 for format)

On exit

R0 corrupted

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

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 issue Service_ResourceFSStarted (not to be confused with Service_ResourceFSStarting), which tells any programs relying on ResourceFS files that the structure has changed.

Related SWIs

ResourceFS_DeregisterFiles (page 2-423)

Related vectors

None

ResourceFS_DeregisterFiles (SWI &41B41)

Remove file(s) from the ResourceFS structure

On entry

R0 = pointer to resource file data (see page 2-418 for format)

On exit

R0 corrupted

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

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).

ResourceFS will unlink the file(s) from its structure, and issue Service_ResourceFSStarted (not to be confused with Service_ResourceFSStarting), which tells any programs relying on ResourceFS files that the structure has changed.

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.

Related SWIs

ResourceFS_RegisterFiles (page 2-422)

Related vectors

None

* 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 after RISC OS 2, 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.

DeviceFS is not available in RISC OS 2.

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_seq[/escape_seq]…`

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 describes 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

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 1 word per escape character, set to &DEADDEAD if the corresponding keyword is not present in the special field string.

Numbers are simply stored into the word; they are decoded using OS_ReadUnsigned and stored away. Switches store the state of the keywords placed, ie:

mike,dennis/S

This yields 0 if ‘mike’ is present within the string, 1 if ‘dennis’ is present within the string.

The order of commands within the validation string and the special field string need 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 call is issued when the module wants the device drivers to re-register with DeviceFS; it 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
R3 = pointer to device name (if an individual device is being deregistered),
    or 0 (if the device driver as a whole is being deregistered)

On exit

All registers must be preserved

Use

This is issued to inform a device driver that a specified device 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.

If a device driver is being deregistered, this service call is issued once for each device using that driver (with R3 pointing to the device name), and is then issued a last time with R3 set to 0.

Service_DeviceFSCloseRequest (Service Call &81)

Opening a device which already has the maximum number of streams open

On entry

R1 = &81 (reason code)
R2 = file handle of an open stream on that device

On exit

R1 = 0 if the file was closed; otherwise all registers preserved

Use

This service call is issued whenever an attempt is made to open a device for input or output and one of the following applies:

- the device already has the maximum number of streams open for input or output respectively; one such stream must be closed before the new one can be opened
- the device is not full duplex, and already has one or more streams open for output or input respectively; all such streams must be closed before the new one can be opened.

The service call is offered in the hope that one or more 'blocking' streams need no longer be kept open and can be closed, allowing the new stream to be opened.

If your application opened the stream specified by R2, and you no longer need to keep it open, you should close it and then claim the service call to inform DeviceFS that you have done so. Otherwise you should pass on the service call with all registers preserved.

The kernel responds to this service call, because it implicitly opens streams such as the printer and the serial device, which need only be open when actually sending data.

DeviceFS issues this service call for each blocking stream, stopping if sufficient blocking streams have been closed for it to open the new stream, or if this is clearly impossible (eg the service call is not claimed for an output stream that is blocking input to a half-duplex device).

Using filing systems

SWI Calls

DeviceFS_Register (SWI &42740)

Registers a device driver and its associated devices with DeviceFS

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 a 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_RegisterObjects 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
24	reserved (must be zero)

Device names should be registered with Acorn; see *Appendix H: Registering names* on page 4-551. 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 4-88.

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 2-607.

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 2-430.

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.

Related SWIs

DeviceFS_Deregister (page 2-438), DeviceFS_RegisterObjects (page 2-439)

Related vectors

None

DeviceFS_Deregister (SWI &42741)

Deregisters all devices and their device driver from DeviceFS

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.

Related SWIs

DeviceFS_Register (page 2-435), DeviceFS_DeregisterObjects (page 2-440)

Related vectors

None

DeviceFS_RegisterObjects (SWI &42742)

Registers a list of additional devices with a device driver

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 2-435).

Related SWIs

DeviceFS_DeregisterObjects (page 2-440)

Related vectors

None

DeviceFS_DeregisterObjects (SWI &42743)

Deregisters a device related to a particular device driver

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.

Related SWIs

DeviceFS_RegisterObjects (page 2-439)

Related vectors

None

DeviceFS_CallDevice (SWI &42744)

Makes a call to a device with a specified register set

On entry

R0 = reason code
R1 = device driver's 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 '$').

Related SWIs

None

Related vectors

None

DeviceFS_Threshold (SWI &42745)

Informs DeviceFS of the threshold value to use on buffered devices

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 cross the specified threshold.

An error is generated if the device is not buffered.

Related SWIs

None

Related vectors

None

DeviceFS_ReceivedCharacter (SWI &42746)

Informs DeviceFS that a device driver has received a character

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.

Related SWIs

DeviceFS_TransmitCharacter (page 2-444)

Related vectors

None

DeviceFS_TransmitCharacter (SWI &42747)

Informs DeviceFS that a device driver wants to transmit a character

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.

Related SWIs

DeviceFS_ReceivedCharacter (page 2-443)

Related vectors

None

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 2-429.

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, such as the A5000.

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 control byte.
- 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 control byte.
- OS_Byte 203 reads/writes the serial input buffer threshold.
- 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 1-882.

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 start sending 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 1-520.

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. For RISC OS 2, this works as follows:

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 to NetPrint:)

Whereas for later versions of RISC OS:

Value of R1	Printer driver type
0 - 255	Files in system variables PrinterType$*n* Note that appropriate values are set up for backwards compatibility: eg the serial device driver sets PrinterType$2 to use the serial device.

Thus to send printer output to the serial port, call OS_Byte 5 with R1 = 2.

For full details of OS_Byte 5, see page 1-522.

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 threshold. 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)
- clear 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)

Sets the receive baud rate for the serial port

On entry

R0 = 8
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 receive baud rate for the serial port. It is provided for compatibility with older operating systems, and you should use OS_SerialOp 5 instead; see page 2-479.

(This call uses the same baud rate codes as OS_SerialOp 5.)

Related SWIs

OS_Byte 8 (page 2-453), OS_SerialOp 5 (page 2-479), OS_SerialOp 6 (page 2-481)

Related vectors

ByteV

OS_Byte 8 (SWI &06)

Sets the transmit baud rate for the serial port

On entry

R0 = 8
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 2-481.

(This call uses the same baud rate codes as OS_SerialOp 6.)

Related SWIs

OS_Byte 7 (page 2-451), OS_SerialOp 5 (page 2-479), OS_SerialOp 6 (page 2-481)

Related vectors

ByteV

OS_Byte 156 (SWI &06)

Reads/writes serial port state

On entry

R0 = 156
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 2-460) .

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 1-882). 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 2-468) to provide these functions.

Related SWIs

OS_Byte 192 (page 2-460), OS_SerialOp (page 2-468)

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 on page 1-526)

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
- 'Escape' events and 'character entering input buffer' events are not generated.

Related SWIs

None

Related vectors

ByteV

OS_Byte 191 (SWI &06)

Read/write serial busy flag

On entry

R0 = 191
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 on page 2-460)

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 on page 1-678)

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 2-454), OS_SerialOp (page 2-468)

Related vectors

ByteV

OS_Byte 203 (SWI &06)

Read/write serial input buffer threshold value

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 on page 2-464)

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 (page 1-150)

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 on page 1-417)

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 require 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 2-451), OS_Byte 8 (page 2-453), OS_SerialOp (page 2-468)

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	Page
0	Read/write serial states	2-470
1	Read/write data format	2-473
2	Send break	2-475
3	Send byte	2-476
4	Get byte	2-477
5	Read/write receive baud rate	2-479
6	Read/write transmit baud rate	2-481

For a detailed explanation of each reason code, see the relevant page.

Related SWIs

None

Related vectors

SerialV

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 not suppressed.
		1	Input is 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 is low.
		1	The ring indicator bit is high.
21	RO	0	CTS low (clear to send)
		1	CTS high (not clear to send)
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 2-454)

Related vectors

SerialV

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 2-454)

Related vectors

SerialV

OS_SerialOp 2 (SWI &57)

Send break

On entry

R0 = 2 (reason code)
R1 = length of break in centiseconds

On exit

R0, 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

SerialV

OS_SerialOp 3 (SWI &57)

Send byte

On entry

R0 = 3 (reason code)
R1 = character to be sent

On exit

R0, R1 preserved
C flag clear if character was sent, or set if character was not sent (ie 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

SerialV

OS_SerialOp 4 (SWI &57)

Get a byte from the serial buffer

On entry

R0 = 4

On exit

R0 preserved
R1 = character received (if C flag clear), or preserved (if C flag set – ie no character available in buffer to read)

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 the transmitting device is re-enabled in the way specified by the serial port state (as set by OS_SerialOp 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

SerialV

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

SerialV

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 2-453)

Related vectors

SerialV

*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 2-451), OS_Byte 8 (page 2-453), OS_SerialOp 5 (page 2-479), OS_SerialOp 6 (page 2-481)

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 2-454), OS_SerialOp 1 (page 2-473)

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 PrinterType$1 to point at its DeviceFS object, ie:

PrinterType$1 ⇒ devices#buffer3:$.Parallel

The module supports SWIs to allow the 82C710 or 82C711 chip driving the parallel port to be directly accessed (if present – some machines use other chips).

The ‘parallel:’ device can be opened for output (eg to a printer) or input but not for both. The input stream is only available on machines which use an 710/711 controller.

The output stream uses standard parallel printer handshaking, and can send data to many types of printer. In the absence of any standard parallel input protocol the input stream has been provided mainly as a means of passing data between one machine and another (eg downloading data from a portable to a master machine). The input device driver behaves like a printer, and can therefore accept data from another machine which is ‘printing’ from its parallel port. To enable such data transfer a twisted cable must be made with the following connections:

Pin	Signal	Direction	Pin	Signal	Direction
1	/STROBE	O	10	/ACK	I
2	DATA 0	I/O	2	DATA 0	I/O
3	DATA 1	I/O	3	DATA 1	I/O
4	DATA 2	I/O	4	DATA 2	I/O
5	DATA 3	I/O	5	DATA 3	I/O
6	DATA 4	I/O	6	DATA 4	I/O
7	DATA 5	I/O	7	DATA 5	I/O
8	DATA 6	I/O	8	DATA 6	I/O
9	DATA 7	I/O	9	DATA 7	I/O
10	/ACK	I	1	/STROBE	O
11	BUSY	I	17	/SLCTIN	O
17	/SLCTIN	O	11	BUSY	I

Either end of such a cable can be connected to a sending or receiving machine. Note that sending (ie ‘printing’) machines do not need to be Acorn products, so you can use the parallel input device to transfer data from, for example, a PC.

To send data, the 'parallel:' device should be opened for output as if it were a file. Data can then be written to the open device which should be closed when no more data is to be sent (`*Copy` *`file`* `printer#parallel:` does this). At the receiving end the 'parallel:' device should be opened for input, the bytes should be read, and then the device should be closed.

SWI calls

Parallel_HardwareAddress (SWI &42EC0)

This call is for internal use only. Do not use it; use the SWI Parallel_Op instead.

Parallel_Op (SWI &42EC1)

Provides low level parallel operations

On entry

R0 = reason code
other registers are reason code dependent

On exit

R0 preserved
other registers are reason code dependent

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call provides low level parallel operations, which are only available with newer hardware that uses the 710/711 family of controllers (such as the A5000). The SWI returns an error for (older) machines on which a 710/711 is not present.

The action of the call depends on the reason code passed in R0:

R0	Action	Page
0	read data and status registers	2-492
1	write data register	2-493
2	read/write control register	2-494

This call is not available in RISC OS 2, or in RISC OS 3.00.

The call is provided to allow you to drive the 82C710/82C711 for yourself. If you intend to drive the hardware directly then you should open the `parallel:` device. For example:

```
lock =OPENOUT("parallel:")
... play around with hardware ...
CLOSE#lock
```

This stops any other application altering the values that you have setup, preventing any possible confusion.

Related SWIs

OS_ClaimDeviceVector (page 1-123), OS_ReleaseDeviceVector (page 1-125)

Related vectors

None

Parallel_Op 0 (SWI &42EC1)

Reads the parallel data and status registers

On entry

R0 = 0

On exit

R0 preserved
R1 = contents of the parallel data register
R2 = contents of the parallel status register

Use

This call is used to obtain the current state of the parallel data lines and status register. The bits in the read only parallel status register correspond to the following inputs:

7	6	5	4	3	2	1	0
$\overline{\text{BUSY}}$	$\overline{\text{ACK}}$	PE	SLCT	$\overline{\text{ERROR}}$	rsvd	rsvd	rsvd

Figure 39.1 Parallel status register

Bits 8 to 31 of R2 are undefined. See the 82C710/82C711 data sheet for a description of these bits. If the DIR bit in the parallel control register (see page 2-494) is 0 (ie output) then the contents of the data register will be the same as the last data value written. The data register is read after the status register.

Parallel_Op 1 (SWI &42EC1)

Writes the parallel data register

On entry

R0 = 1
R1 = data

On exit

R0, R1 preserved

Use

This call is used to write a byte to the parallel data lines. This will only have an effect if the DIR bit in the parallel control register (see page 2-494) is 0 (ie output).

Parallel_Op 2 (SWI &42EC1)

Reads/writes the parallel control register

On entry

R0 = 2
R1 = EOR mask
R2 = AND mask

On exit

R0 preserved
R1 = old contents of the parallel control register
R2 = new contents of the parallel control register

Use

This call is used to read or write the current state of the parallel control register. The new state is determined by:

new state = (old state AND R2) EOR R1

The bits in this value correspond to the following outputs:

7	6	5	4	3	2	1	0
rsvd	rsvd	DIR	IRQEN	$\overline{\text{SLCTIN}}$	$\overline{\text{INIT}}$	$\overline{\text{AUTOFD}}$	$\overline{\text{STROBE}}$

Figure 39.2 Parallel control register

Bits 8 to 31 are undefined and must not be modified. See the 82C710/82C711 data sheet for a description of these bits. These lines are output only, but their current state can be read without changing them by setting R1 = 0 and R2 = &FFFFFFFF. The interrupt enable bit, IRQEN, should normally be 1 and interrupt disabling should be done in IOC.

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 currently configured printer
`netprint:`	the currently configured 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 2-445
`parallel:`	parallel port; see the chapter entitled *Parallel device* on page 2-487

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 than 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:`, which refers to printer type 3
- `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 2-445.

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

The Filer is responsible for providing a graphical representation of the filing system structure. It uses standard filing system calls to do its work, and so will work with any filing system.

The filing-system-specific desktop filers – such as ADFSFiler – cooperate with the Filer by issuing the command *Filer_OpenDir when their icon is clicked on, so that the Filer can open the appropriate directory display. They also provide other operations which are not sufficiently generic to be provided by the Filer: for example the Format and Verify operations provided by the ADFSFiler.

See the section entitled *Filer messages* on page 3-231 for full details on messages used by the Filer.

Service Calls

Service_StartFiler
(Service Call &4B)

Request to filing-system-specific desktop filers 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-specific desktop filers 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-specific desktop filer 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 to Wimp_StartTask the * Command returned by the module. 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 `taskHandle` 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-specific desktop filers to set `taskHandle` variable to zero

On entry

R1 = &4C or &27 (reason code)

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-specific desktop filers. 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-specific desktop filers are also closed down, by issuing this service call.

On receipt of this service call, a filing-system-specific desktop filers 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)

Enumerate available disc formats

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 extended list of format specifications suitable for a menu

Use

This service call is issued to get information about the available formats, and to support !Help for those formats.

- This service call should be issued when the information is required, as formats can be dynamically added and removed by soft-loading or removing modules. If this service call is only issued once, 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 client 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 = pointer to description of disc which has been dismounted

On exit

All registers are preserved

Use

This call informs modules that a disc has just been dismounted so they can take appropriate action. For example the Filer might close any open directory displays for that disc.

The value in R2 should be a pointer to a null-terminated string of the following form:

`filing_system::disc`

where *filing_system* is the name of the filing system, and *disc* is the name of the disc. If the disc has no name then the drive number 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 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.

Under RISC OS 2 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 will close all directory displays that match the specified sub-string. Under RISC OS 2 the minimum substring you can pass is the filing system only, whereas under RISC OS 3 you must also specify the drive name.

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 pathname of a directory whose directory display is to be opened
x	the x-coordinate of the top left of the directory display, in OS units
y	the y-coordinate of the top left of the directory display, in OS units
width	the width of the directory display, in OS units
height	the height of the directory display, in OS units
switches	switches to control the display type of the directory display; there are alternatives; the case of the letters is not significant:

`-SmallIcons`	`-si`	display small icons
`-LargeIcons`	`-li`	display large icons
`-FullInfo`	`-fi`	display full information
`-SortByName`	`-sn`	display sorted by name
`-SortByType`	`-st`	display sorted by type
`-SortByDate`	`-sd`	display sorted by date
`-SortBySize`	`-ss`	display sorted by size

Use

*Filer_OpenDir opens a directory display on the Desktop.

Under RISC OS 2, 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.

RISC OS always ensures that the opened directory display is entirely visible on the desktop, and that its size is within the normal limits imposed by the Filer. Invalid parameters are rounded up/down until valid.

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 variants on the keywords; again, the case of the letters is not significant. Valid keywords are:

Keyword	**Alternative**	**Precedes parameter**
`-dir`	`-directory`	*directory*
`-x0`	`-topleftx`	*x*
`-y1`	`-toplefty`	*y*
`-width`	`-w`	*width*
`-height`	`-h`	*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

Performs the equivalent of double-clicking on an object in a directory display

Syntax

`*Filer_Run` *`object`*

Parameters

`object`	a valid pathname specifying an object to be treated as if double-clicked on

Use

*Filer_Run performs the equivalent of double-clicking on an object in a directory display. For example an application would be run, a directory would be opened, and a file might be loaded into the relevant application. 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 and FilerSWIs

Introduction and Overview

The Filer_Action 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 3-233 for details of how the Filer Action window operates.

The FilerSWIs module provides SWIs to help make starting Filer_Action easier.

SWI calls

FilerAction_SendSelectedDirectory (SWI &40F80)

Sends message specifying the selected directory

On entry

R0 = task handle to which 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 3-233).

For a description of how messages within the Wimp environment are generated see *Wimp_SendMessage (SWI &400E7)* on page 3-193.

Related SWIs

FilerAction_SendSelectedFile (page 2-515),
FilerAction_SendStartOperation (page 2-517)

Related vectors

None

FilerAction_SendSelectedFile (SWI &40F81)

Sends message specifying the selected files within a directory

On entry

R0 = task handle to which 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 accumulates the names of selected files that you pass to it. When it has received sufficient filenames to fill a Wimp message block it sends those names using the Wimp message Message_FilerAddSelection (see page 3-234). The same message is used to send any unsent filenames if you subsequently call FilerAction_SendStartOperation.

For a description of how messages within the Wimp environment are generated see *Wimp_SendMessage (SWI &400E7)* on page 3-193.

Related SWIs

FilerAction_SendSelectedDirectory (page 2-514),
FilerAction_SendStartOperation (page 2-517)

Related vectors

None

FilerAction_SendStartOperation (SWI &40F82)

Sends message containing information to start operation

On entry

R0 = task handle to which 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 4

Set type:
R3 pointer to word containing new type in bits 0-11
R4 4

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 3-234). Before doing so, it uses Message_FilerAddSelection to send any filenames passed to Filer_Action using FilerAction_SendSelectedFile that have not already been sent.

For a description of how messages within the Wimp environment are generated see *Wimp_SendMessage (SWI &400E7)* on page 3-193.

Related SWIs

FilerAction_SendSelectedDirectory (page 2-514),
FilerAction_SendSelectedFile (page 2-515)

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 a 'start operation' message 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 (page 3-174) 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 (page 3-174)

Related vectors

None

43 Free

Introduction and Overview

This module enables an interactive free space display from the desktop.

Any filing system that wishes to display an interactive free space display should register with this module. In doing so, the filing system provides the address of a routine that accepts a variety of reason codes, each of which provides support for this module.

When the **Free** entry is selected from the filing system's menu, its desktop filer should issue the command:

`*ShowFree -fs` *`fs_name device`*

This module will then display the free space left.

The Free module is not available in RISC OS 2.

SWI calls

Free_Register (SWI &444C0)

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
R2 = 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 call adds the filing system to the list of filing systems known by the Free module. The Free module automatically deals with the following filing systems: ADFS, RamFS, NetFS, NFS, SCSIFS.

R1 contains the address of the entry point for a set of support routines, which the Free module uses to help it to provide an interactive free space display for the filing system. The entry point is called in User mode, with the Free module's private stack, the top of which contains the return address. You cannot assume the depth of this stack, and should not use it save to pull the return address. Alternatives are:

- Construct a new stack.

- Call the SWI OS_EnterOS to get the SVC mode stack, do the work and then return to User mode before returning.
- Use a SWI to do all the work: for example, you might use OS_FSControl 49 (get free space) for reason code 2.

The routine should exit using the instruction:

```
LDMIA   R13,{PC}
```

The entry point may be called with the following reason codes:

Reason code 0 – NoOp

On entry

—

On exit

—

Details

This entry point is a No Op, and you should just return with all registers preserved.

Reason code 1 – Get 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. You should place the device name in the buffer pointed to by R2, and the length of the name in R0.

Reason code 2 – Get free space for device

On entry

R0 = 2
R1 = filing system number
R2 = pointer to 3 word 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 pointed to by R2 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

Reason code 3 – Compare 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 (eg RAMFS).

Related SWIs

Free_DeRegister (page 2-526)

Related vectors

None

Free_DeRegister (SWI &444C1)

Removes the filing system from the list of filing systems known by the Free module

On entry

R0 = filing system number
R1 = address of routine (as passed to Free_Register)
R2 = R12 value (as passed to Free_Register)

On exit

R0 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 call removes the filing system from the list of filing systems known by the Free module.

Related SWIs

Free_Register (page 2-522)

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. It is used by desktop filers such as ADFSFiler

This command will only work on filing systems registered using the SWI Free_Register.

Example

```
*ShowFree -fs adfs HardDisc4
```

Related commands

None

Related SWIs

Free_Register (page 2-522), Free_DeRegister (page 2-526)

Related vectors

None

Part 5 – Writing filing systems

44 Writing a filing system

Writing your own filing system

You can add filing systems to RISC OS. You must write them as relocatable modules. There are two ways of doing so:

- by adding a module that FileSwitch communicates with directly
- by adding a secondary module to FileCore; FileSwitch communicates with FileCore, which then communicates with your module.

In both cases, the amount of work you have to do is considerably less than if you were to write a filing system from scratch, as the FileSwitch and FileCore modules already provide a core of the functions your filing system must offer. Obviously if you use FileCore as well as FileSwitch, more is already provided for you, and so you have even less work to do. The structure of FileCore is then imposed on your filing system; to the user, it will appear very similar to ADFS, leading to a consistency of design.

Obviously there is no way that FileSwitch can know how to communicate directly with the entire range of hardware that any filing system might use. Your filing system must provide these facilities, and declare the entry points to FileSwitch. When FileSwitch receives a SWI call or * Command, it does its share of the work, and uses these entry points to get the relevant filing system to do the work that is hardware dependent.

What to read next

The relevance of the rest of this chapter depends on how you intend to write your own filing system:

- if you are not using FileCore, then you should read this chapter, which tells you how to add a filing system to FileSwitch
- if you are using FileCore, then you should ignore this chapter and instead read the chapter entitled *Writing a FileCore module* on page 2-597.

In both cases you should also see the chapter entitled *Modules* on page 1-201, for more information on how to write a module.

Filing systems

Declaring a filing system

When your module initialises, it must declare itself to be a filing system, so that FileSwitch knows of its existence. You must call OS_FSControl 12 to do this – see page 2-96 for details. R1 and R2 tell FileSwitch where to find a *filing system information block*. This in turn tells FileSwitch the locations of all the entry points to the filing system's low level routines that interface with the hardware.

Filing system information block

This table shows the offsets from the start of the filing system information block, and the meaning of each word in the block:

Offset	Contains	
&00	Offset of filing system name (null terminated)	
&04	Offset of filing system startup text (null terminated)	
&08	Offset of routine to open files	(FSEntry_Open)
&0C	Offset of routine to get bytes from media	(FSEntry_GetBytes)
&10	Offset of routine to put bytes to media	(FSEntry_PutBytes)
&14	Offset of routine to control open files	(FSEntry_Args)
&18	Offset of routine to close open files	(FSEntry_Close)
&1C	Offset of routine to do whole file operations	(FSEntry_File)
&20	*Filing system information word*	
&24	Offset of routine to do various FS operations	(FSEntry_Func)
&28	Offset of routine to do multi-byte operations	(FSEntry_GBPB)
&2C	*Extra filing system information word* (optional)	

The offsets held in each word are from the base of the filing system module. The GBPB entry (at offset &28 from the start of the information block) is optional if the filing system supports non buffered I/O, and not required otherwise.

The block need not exist for long, as FileSwitch takes a copy of it and converts the entry points to absolute addresses. So you could set up the block as an area in a stack frame, for example.

Filing system information word

The filing system information word (at offset &20) tells FileSwitch various things about the filing system:

Bit	Meaning if set
31	Special fields are supported
30	Streams are interactive (ie prompting for input is appropriate)
29	Filing system supports null length filenames

28	Filing system should be called to open a file whether or not it exists
27	Tell the filing system when flushing by calling FSEntry_Args 255
26	Filing system supports FSEntry_File 9
25	Filing system supports FSEntry_Func 20
24	Reserved – must be zero
23	Filing system supports image filing system extensions
22	Pass & and % in paths when appropriate
21	Need not store directories for this filing system
20	Use Open/GetBytes/Close entry points rather than File 255
19	Use Open/GetBytes/Close entry points rather than File 0
18	Use FSEntry_Func 9 in preference to FSEntry_File entry points
17	Extra filing system information word is present
16	Filing system is read-only
15 - 8	Maximum number of files that may be open (see below)
7 - 0	Filing system number (see below)

Bits 16 - 23 are ignored by RISC OS 2. File systems that were written for RISC OS 2 should have these bits clear, which may cause problems: for example, RISC OS 2 read-only filing systems will incorrectly have bit 16 clear.

Bits 8 - 15 tell FileSwitch the maximum number of files that can be easily opened on the filing system (per server, if appropriate). A value of 0 means that there is no definite limiting factor – DMA failure does not count as such a factor. These bits may be used by system extension modules such as the Font Manager to decide whether a file may be left open or should be opened and closed as needed, to avoid the main application running out of file handles.

Bits 0 - 7 contain the filing system identification number. Currently allocated ones are listed in the chapter entitled *Filing system numbers* on page 2-21. For your own allocation, contact Acorn Computers in writing: see *Appendix H: Registering names* on page 4-551.

Extra filing system information word

The extra filing system information word is present if bit 17 of the filing system information word is set. If absent, it is assumed by FileSwitch to have value zero. The meaning of the bits in the word is as follows:

Bit	Meaning if set
0	Filing system supports FSEntry_Func 34
1	Filing system should be called to do Cat
2	Filing system should be called to do Ex
3 - 31	Reserved – must be zero

You should only set bits 1 and 2 if your filing system provides a non-standard format for Cat and Ex respectively.

Service Call handler

Your filing system must have a Service Call handler. It must respond to Service_FSRedeclare (see page 2-25) by redeclaring the filing system. For some filing systems, it may be appropriate to respond to Service_CloseFile (page 2-26). Disc based filing systems should also support Service_IdentifyDisc (page 2-220), Service_EnumerateFormats (page 2-504), Service_IdentifyFormat (page 2-281), and Service_DisplayFormatHelp (page 2-282).

Selecting your filing system

If your filing system has associated file storage, it must provide a * Command to select itself, such as *ADFS or *Net. This must call OS_FSControl 14 to direct FileSwitch to make the named filing system current, thus:

```
StarFilingSystemCommand
        STMFD   R13!, {R14}      ; In a * Command so R0-R6 may be corrupted
        MOV     R0, #FSControl_SelectFS ; 14
        ADR     R1, FilingSystemName
        SWI     XOS_FSControl
        LDMFD   R13!, {PC}
```

For full details of OS_FSControl 14, see page 2-98.

Other * Commands

There are no other * Commands that your filing system **must** provide, but it obviously **should** provide more than just a way to select itself. Look through the previous chapters in this part of the manual to see what other filing systems offer.

If the list of * Commands you want to provide closely matches those in the chapter entitled *FileCore* on page 2-197, you ought to investigate adding your filing system to FileCore rather than to FileSwitch; this will be less work for you.

Removing your filing system

The finalise entry of your module must call OS_FSControl 16 (for both soft and hard deaths), so that FileSwitch knows that your filing system is being removed:

```
        MOV     R0, #FSControl_RemoveFS ; 16
        ADR     R1, FilingSystemName
        SWI     XOS_FSControl
        CMP     PC, #0                  ; Clears V (also clears N,Z, sets C)
```

For full details of OS_FSControl 16, see page 2-100.

Image filing systems

For a description of image filing systems, and their relationship to other filing systems, see the chapter entitled *DOSFS* on page 2-323. Image filing systems are not supported by RISC OS 2.

Declaring an image filing system

When your module initialises, it must declare itself to be an image filing system, so that FileSwitch knows of its existence. You must call OS_FSControl 35 to do this – see page 2-119 for details. R1 and R2 tell FileSwitch where to find an *image filing system information block*. This in turn tells FileSwitch the locations of all the entry points to the image filing system's low level routines that interface with the hardware.

Image filing system information block

This table shows the offsets from the start of the image filing system information block, and the meaning of each word in the block:

Offset	Contains	
&00	*Image filing system information word*	
&04	Image filing system file type	
&08	Offset of routine to open files	(ImageEntry_Open)
&0C	Offset of routine to get bytes from media	(ImageEntry_GetBytes)
&10	Offset of routine to put bytes to media	(ImageEntry_PutBytes)
&14	Offset of routine to control open files	(ImageEntry_Args)
&18	Offset of routine to close open files	(ImageEntry_Close)
&1C	Offset of routine to do whole file operations	(ImageEntry_File)
&24	Offset of routine to do various FS operations	(ImageEntry_Func)

The offsets held in each word are from the base of the image filing system module.

The block need not exist for long, as FileSwitch takes a copy of it and converts the entry points to absolute addresses. So you could set up the block as an area in a stack frame, for example.

The image filing system file type gives the numerical file type of files which contain images understood by the image filing system.

Image filing system information word

The image filing system information word (at offset 0) tells FileSwitch various things about the image filing system:

Bit	Meaning if set
27	Tell the image filing system when flushing by calling ImageEntry_Args 255

All other bits are reserved and should be set to zero.

Service Call handler

Your image filing system must have a Service Call handler. It must respond to the same service calls as any other filing system; see the section entitled *Service Call handler* on page 2-534.

* Commands

There are no * Commands that your image filing system **must** provide, but most **should** provide some. See the chapter entitled *DOSFS* on page 2-323 for an example of what other image filing systems offer.

Removing your image filing system

The finalise entry of your module must call OS_FSControl 36 (for both soft and hard deaths), so that FileSwitch knows that your image filing system is being removed:

```
MOV     R0, #FSControl_DeRegisterImageFS        ; 36
ADR     R1, ImageFileType
SWI     XOS_FSControl
CMP     PC, #0                  ; Clears V (also clears N,Z, sets C)
```

For full details of OS_FSControl 36, see page 2-120.

Filing system interfaces: introduction

Calling conventions

The principal part of a filing system (or of an image filing system) is the set of low-level routines that control the filing system's hardware. There are certain conventions that apply to them.

Processor mode

Routines called by FileSwitch are always entered in SVC mode, with both IRQs and FIQs enabled. This means you do not have to change mode to access hardware devices directly, and are able to change to FIQ mode to set up FIQ registers if necessary.

Using the stack

R13 in supervisor mode is used as the system stack pointer. The filing system (or image filing system) may use this full descending stack. When the filing system (or image filing system) is entered you should take care not to push too much onto the stack, as it is only guaranteed to be 1024 bytes deep; however most of the time it is substantially greater. The stack base is on a 1Mbyte boundary. Hence, to determine how much stack space there is left for your use, use the following code:

```
MOV     R0, R13, LSR #20        ; Get Mbyte value of SP
SUB     R0, R13, R0, LSL #20    ; Sub it from actual value
```

You may move the stack pointer downwards by a given amount and use that amount of memory as temporary workspace. However, interrupt processes are allowed to use the supervisor stack so you must leave enough room for these to operate. Similarly, if you call any operating system routines, you must give them enough stack space.

Using file buffers

If a read or write operation occurs that requires a file buffer to be claimed for a file, and this memory claim fails, then FileSwitch will look to steal a file buffer from some other file. Victims are looked for in the order:

1 an unmodified buffer of the same size

2 an unmodified buffer of a larger size

3 a modified buffer of the same size

4 a modified buffer of a larger size.

In the last two cases, FileSwitch obviously calls the filing system (or image filing system) to write out the buffer first, before giving it to the new owner. If an error occurs in writing out the buffer under RISC OS 2, the stream that owned the data in the buffer (not the stream that needed to get the buffer) is marked as having ‘data lost’; any further operations will return the ‘Data lost’ error. FileSwitch is always capable of having one file buffered at any time, although it won’t work very well under such conditions.

Workspace

R12 on entry to the filing system (or image filing system) is set to the value of R3 it passed to FileSwitch when initialising by calling OS_FSControl 12 or 35. Conventionally, this is used as a pointer to your private word. In this case, module entries should contain the following:

```
LDR R12, [R12]
```

to load the actual private word into the register.

Returning errors

The error numbers your filing system returns should take this format:

&0001*nnee*

where *nn* is the filing system number, as passed in the information word (see the section entitled *Filing system information word* on page 2-532); and *ee* is one of the error numbers used by FileCore based filing systems (see the table on page 2-600), or – if none is relevant – a number that does not appear in that table.

Supporting unbuffered streams

Filing systems may support both buffered and unbuffered streams. Unbuffered streams must maintain their own sequential pointers, file extents and allocated sizes. File Switch will maintain the *EOF-error-on-next-read* flag for them.

Image filing system streams are always buffered; consequently they should not support unbuffered streams.

Dealing with access

Generally FileSwitch does not make calls to filing systems (or to image filing systems) unless the access on objects is correct for the requested operation.

Note that if a file is opened for buffered output and has only write access, FileSwitch may still attempt to read from it to perform its file buffering. You must not fault this.

Other conventions

Filing system (or image filing system) routines do not need to preserve any registers other than R13.

If a routine wishes to return an error, it should return to FileSwitch with V set and R0 pointing to a standard format error block.

You may assume that:

- **all names are null terminated**
- all pathnames are non-null, unless the filing system allows them (for example printer:)
- all pathnames have correct syntax.

All pathnames should be treated as read-only If you do need to make changes to a pathname, you must copy it to your local workspace and modify that copy.

Using canonical names

All filing system interfaces, with the exception of FSEntry_Func 23, are always passed names in the canonical form. This canonical form is defined by the designer of a particular filing system and is fixed. Canonical form is used to ensure that dissimilar references to the same object reduce to identical strings, and thus the filing system can easily determine that two object references are to the same object. For example after:

```
*Mount 0 *Dir A.z
```

references to `$.a.b.c` and to `^.b.c` will reduce to the same canonical form:

```
adfs::MyDisc.$.a.b.c
```

The use of canonical form also helps the filing system to run faster. Because all filing system interfaces only receive canonical names, the parsing can be fast and efficient. Remember that canonicalisation happens once for several calls to the filing system itself.

The chosen canonical form should be a subset of the acceptable name styles, and hence the canonical name should be acceptable to the canonicaliser as input. For example the input syntax for the NetFS canonicaliser is:

`Net[#(name|number]:[:discname.]$|&`

The output format is:

`Net::name.$|&`

It is also worthwhile optimising the canonicalisation code so that an already canonical name is processed very fast.

Canonical names are not used by RISC OS 2.

ImageEntry entry points

In the following descriptions a pathname will always be relative to the root directory of the image, and will never have any '^', '$', '@', '%', '\' or '&' characters in it. When a wildcarded pathname is specified, the operation should be applied to all matching leafnames; but earlier wildcarded elements in the path should use the first match. A null pathname indicates the root directory of the image.

Interfaces

These are the interfaces that your filing system (or image filing system) must provide. Their entry points must be declared to FileSwitch by calling OS_FSControl 12 when your filing system module is initialised, or by calling OS_FSControl 35 when your image filing system module is initialised.

FSEntry_Open and ImageEntry_Open

Open a file

On entry (FSEntry_Open)

R0 = reason code
R1 = pointer to filename
R3 = FileSwitch handle for the file
R6 = pointer to special field if present, otherwise 0

On exit (FSEntry_Open)

R0 = file information word (**not** the same as the filing system information word)
R1 = your filing system's handle for the file (0 if not found)
R2 = buffer size for FileSwitch to use (0 if file unbuffered, else must be a power of 2 between 64 and 1024)
R3 = file extent (buffered files only)
R4 = space currently allocated to file (buffered files only: must be a multiple of buffer size)

On entry (ImageEntry_Open)

R1 = pointer to filename
R3 = FileSwitch handle for the file
R6 = image filing system's handle for image that contains file

On exit (ImageEntry_Open)

R0 = image file information word (**not** the same as the image filing system information word)
R1 = your image filing system's handle for the file (0 if not found)
R2 = buffer size for FileSwitch to use (must be a power of 2 between 64 and 1024)
R3 = file extent
R4 = space allocated to file (must be a multiple of buffer size)

Use

FileSwitch calls this entry point to open a file for read or write, and to create it if necessary.

General details

On entry, R3 contains the handle that FileSwitch will use for the file if your filing system successfully opens it. This is a small integer (typically going downwards from 255), but must be treated as a 32-bit word for future compatibility. Your filing system may want to make a note of it when the file is opened, in case it needs to refer to files by their FileSwitch handles (for example, it must close all open files on a *Dismount). It is the FileSwitch handle that the user sees.

On exit, your filing system must return a 32-bit file handle that it uses internally to FileSwitch. FileSwitch will then use this file handle for any further calls to your filing system. You may use any value, apart from a handle of 0 which means that no file is open.

The value returned in R2 is the natural block size of the file; for disc oriented filing systems, this should be the same as the natural sector size. FileSwitch – when calling the filing system – will tend to use multiple of this value, aligned on a boundary which is also a multiple of this value.

If your memory allocation fails, this is not an error, and you should indicate it to FileSwitch by setting R1 to 0 on exit.

Details specific to FSEntry_Open

The reason code given in R0 has the following meaning:

Value	Meaning
0	Open for read
1	Create and open for update (only used by RISC OS 2)
2	Open for update

For both reason codes 0 and 2 FileSwitch will already have checked that the object exists (unless you have overridden this by setting bit 28 of the filing system information word) and, for reason code 2 only, that it is not a directory. These reason codes must not alter a file’s datestamp.

If a directory is opened for reading, then bytes will **not** be requested from it. The use of this is for compatibility with existing programs which use this as a method of testing the existence of an object.

For reason code 1 FileSwitch will already have checked that the leafname is not wildcarded, and that the object is not an existing directory. Your filing system should return an extent of zero. If the file already exists you should return an allocated space the

same as that of the file; otherwise you should return a sensible default that allows space for the file to grow. Your filing system should also give a new file a filetype of &FFD (Data), datestamp it, and give it sensible access attributes (WR/ is recommended).

The file information word returned in R0 uses the following bits:

Bit	**Meaning if set**
31	Write permitted to this file
30	Read permitted from this file
29	Object is a directory
28	Unbuffered OS_GBPB supported (stream-type devices only)
27	Stream is interactive

All other bits are reserved and should be set to 0.

An interactive stream is one on which prompting for input is appropriate, such as kbd:.

Details specific to ImageEntry_Open

FileSwitch will already have checked that the object exists and that it is not a directory. You must not alter a file's datestamp.

The image file information word returned in R0 uses the following bits:

Bit	**Meaning if set**
31	Write permitted to this file
30	Read permitted from this file

All other bits are reserved and should be set to 0.

FSEntry_GetBytes (from a buffered file), and ImageEntry_GetBytes (all cases)

Get bytes from a buffered file

On entry

R1 = file handle used by your filing system/image filing system
R2 = pointer to buffer
R3 = number of bytes to read into buffer
R4 = file offset from which to get data

On exit

—

Details

This entry point is used by FileSwitch to request that you read a number of bytes from an open file, and place them in memory.

The file handle is guaranteed by FileSwitch not to be a directory, **but not necessarily** to have had read access granted at the time of the open – see the last case given below.

The memory address is not guaranteed to be of any particular alignment. You should if possible optimise your filing system's transfers to word-aligned locations in particular, as FileSwitch's and most clients do tend to be word-aligned. The speed of your transfer routine is vital to filing system performance. An optimised example (similar to that used in RISC OS) is given in the section entitled *Example program* on page 2-591.

The number of bytes to read, and the file offset from which to read data are guaranteed to be a multiple of the buffer size for this file. The file offset will be within the file's extent.

This call is made by FileSwitch for several purposes:

- A client has called OS_BGet at a file offset where FileSwitch has no buffered data, and so FileSwitch needs to read the appropriate block of data into one of its buffers, from where data is returned to the client.
- A client has called OS_GBPB to read a whole number of the buffer size at a file offset that is a multiple of the buffer size. FileSwitch requests that the filing system transfer this data directly to the client's memory. This is often the case where language libraries are being used for file access. If FileSwitch has any buffered data in the transfer range that has been modified but not yet flushed out to the filing system, then this data is copied to the client's memory after the GetBytes call to the filing system.

- A client has called OS_GBPB to perform a more general read. FileSwitch will work out an appropriate set of data transfers. You may be called to fill FileSwitch's buffers as needed and/or to transfer data directly to the client's memory. You should make no assumptions about the exact number and sequence of such calls; as far as possible RISC OS tries to keep the calls in ascending order of file address, to increase efficiency by reducing seek times, and so on.
- A client has called OS_GBPB to perform a more general write. FileSwitch will work out an appropriate set of data transfers. You may be called to fill FileSwitch's buffers as needed, so that the data at the start and/or end of the requested transfer can be put in the right place in FileSwitch's buffers, ready for whole buffer transfer to the filing system as necessary.

Note that FileSwitch holds no buffered data immediately after a file has been opened.

FSEntry_GetBytes (from an unbuffered file)

Get a byte from an unbuffered file

On entry

R1 = file handle used by your filing system

On exit

R0 = byte read, C clear
R0 = undefined, C set if attempting to read at end of file

Details

This entry point is called by FileSwitch to get a single byte from an unbuffered file from the position given by the file's sequential pointer. The sequential pointer must be incremented by one, unless the end of the file has been reached.

The file handle is guaranteed by FileSwitch not to be a directory and to have had read access granted at the time of the open.

Your filing system must not try to keep its own *EOF-error-on-next-read* flag – instead it must return with C set whenever the file's sequential pointer is equal to its extent **before** a byte is read. It is FileSwitch's responsibility to keep the *EOF-error-on-next-read* flag.

If your filing system does not support unbuffered GBPB directly, then FileSwitch will call this entry the necessary number of times to complete its client's request, stopping if you return with the C flag set (EOF).

FSEntry_PutBytes (to a buffered file), and ImageEntry_PutBytes (all cases)

Put bytes to a buffered file

On entry

R1 = file handle used by your filing system/image filing system
R2 = pointer to buffer from which to read data
R3 = number of bytes of data to read from buffer and put to file
R4 = file offset at which to put data

On exit

—

Details

This entry point is called by FileSwitch to request that you take a number of bytes, and place them in the file at the specified file offset.

The file handle is guaranteed by FileSwitch not to be a directory, and to have had write access granted at the time of the open.

The memory address is not guaranteed to be of any particular alignment. You should if possible optimise your filing system's transfers to word-aligned locations in particular, as FileSwitch's and most clients do tend to be word-aligned. The speed of your transfer routine is vital to filing system performance. An optimised example (similar to that used in FileSwitch) is given in the section entitled *Example program* on page 2-591.

The number of bytes to write, and the file offset at which to write data are guaranteed to be a multiple of the buffer size for this file. The final write will be within the file's extent, so it will not need extending.

This call is made by FileSwitch for several purposes:

- A client has called OS_GBPB to write a whole number of the buffer size at a file offset that is a multiple of the buffer size. FileSwitch requests that the filing system transfer this data directly from the client's memory. This is often the case where language libraries are being used for file access. If FileSwitch has any buffered data in the transfer range that has been modified but not yet flushed out to the filing system, then this data is discarded (as it has obviously been invalidated by this operation).
- A client has called OS_BGet/BPut/GBPB at a file offset where FileSwitch has no buffered data, and the current buffer held by FileSwitch has been modified and so must be written to the filing system. (The current FileSwitch implementation does

not maintain multiple buffers on each file. It is likely that this will remain the case, as individual filing systems have better knowledge about how to do disc-caching, and intelligent readahead and writebehind for given devices.)

- A client has called OS_GBPB to perform a more general write. FileSwitch will work out an appropriate set of data transfers. You may be called to empty FileSwitch's buffers as needed and/or to transfer data directly from the client's memory. You should make no assumptions about the exact number and sequence of such calls; as far as possible RISC OS tries to keep the calls in ascending order of file address, to increase efficiency by reducing seek times, and so on.

Note that FileSwitch holds no buffered data immediately after a file has been opened.

FSEntry_PutBytes (to an unbuffered file)

Put a byte to an unbuffered file

On entry

R0 = byte to put to file (top 24 bits zero)
R1 = file handle used by your filing system

On exit

—

Details

This entry point is called by FileSwitch to request that you put a single byte to an unbuffered file at the position given by the file's sequential file pointer. You must advance the sequential pointer by one. If the sequential pointer is equal to the file extent when this call is made, you must increase the allocated space of the file by at least one byte to accommodate the data – although it will be more efficient to increase the allocated space in larger chunks (256 bytes/1k is common).

The file handle is guaranteed by FileSwitch not to be a directory, and to have had write access granted at the time of the open.

If your filing system does not support unbuffered GBPB directly, then FileSwitch will call this entry the necessary number of times to complete its client's request.

FSEntry_Args and ImageEntry_Args

Various calls are made by FileSwitch through these entry points to deal with controlling open files. The actions are specified by R0 as follows:

FSEntry_Args 0

Read sequential file pointer

On entry

R0 = 0
R1 = file handle used by your filing system

On exit

R2 = sequential file pointer

Details

This entry point is called by FileSwitch to read the sequential file pointer for the given file. You should only support this call if your filing system uses unbuffered files.

If your filing system does not support a pointer as the concept is meaningless (kbd: for example) then it must return a pointer of 0, and **not** return an error.

FSEntry_Args 1

Write sequential file pointer

On entry

R0 = 1
R1 = file handle used by your filing system
R2 = new sequential file pointer

On exit

—

Details

This entry point is called by FileSwitch to request that you alter the sequential file pointer for a given file. You should only support this call if your filing system uses unbuffered files.

If the new pointer is greater than the current file extent then:

- if the file was opened only for reading, or only read permission was granted, then return the error ‘Outside file’

- otherwise extend the file with zeros and set the new extent to the new sequential pointer.

If you cannot extend the file you should return an error as soon as possible, and in any case before you update the extent.

If your filing system does not support a pointer as the concept is meaningless (kbd: for example) then it must ignore the call, and **not** return an error.

FSEntry_Args 2

Read file extent

On entry

R0 =2
R1 = file handle used by your filing system

On exit

R2 = file extent

Details

This entry point is called by FileSwitch to read the extent of a given file. You should only support this call if your filing system uses unbuffered files.

If your filing system does not support file extents as the concept is meaningless (kbd: for example) then it must return an extent of 0, and **not** return an error.

FSEntry_Args 3 and ImageEntry_Args 3

Write file extent

On entry

R0 = 3
R1 = file handle used by your filing system/image filing system
R2 = new file extent

On exit

—

Details

This entry point is called by FileSwitch to request that you change the extent of a file.

The file handle is guaranteed by FileSwitch not to be a directory, and to have had write access granted at the time of the open.

If the filing system does not support file extents as the concept is meaningless (kbd: for example) then it must ignore the call, and **not** return an error.

Buffered files

For buffered files, FileSwitch only calls this entry point to set the real file extent just prior to closing an open file. Your filing system should store the value of R2 in the file's catalogue information as its new length.

Unbuffered files

For unbuffered files, FileSwitch calls this entry point whenever requested to by its client.

If the new extent is less than the current sequential pointer (the file is shrinking and the pointer would lie outside the file), then you must set the pointer to the new extent.

If the new extent is greater than the current one then you must extend the file with zeros. If you cannot extend the file you should return an error as soon as possible, and in any case before you update the extent.

FSEntry_Args 4 and ImageEntry_Args 4

Read size allocated to file

On entry

R0 = 4
R1 = file handle used by your filing system/image filing system

On exit

R2 = size allocated to file by filing system

Details

This entry point is called by FileSwitch to read the size allocated to a given file. All filing systems must support this call.

FSEntry_Args 5

EOF check

On entry

R0 = 5
R1 = file handle used by your filing system/image filing system

On exit

R2 = –1 if (sequential pointer is equal to current extent), otherwise R2 = 0

Details

This entry point is called by FileSwitch to determine whether the sequential pointer for a given file is at the end of the file or not. You should only support this call if your filing system uses unbuffered files.

If a filing system does not support a pointer and/or a file extent as the concept(s) are meaningless (kbd: for example) then the treatment of the C bit is dependent on that filing system. For example, kbd: gives EOF when Ctrl-D is read from the keyboard; null: always gives EOF; and vdu: never gives EOF.

FSEntry_Args 6 and ImageEntry_Args 6

Notify of a flush

On entry

R0 = 6
R1 = file handle used by your filing system/image filing system

On exit

R2 = load address of file (or 0)
R3 = execution address of file (or 0)

Details

General details

This entry point is called by FileSwitch to request that your filing system flushes any modified data that it is holding in buffers. You should only support this call if your filing system does its own buffering in addition to that done by FileSwitch. For example, ADFS does its own buffering when doing readahead/writebehind, and so needs to use this call.

Details specific to FSEntry_Args 6

The modified data should be flushed to its storage media.

This entry point is only called if your filing system is buffered, and you set bit 27 of your filing system information word when you initialised your filing system.

Details specific to ImageEntry_Args 6

The modified data should be flushed to its image. The image should subsequently be flushed to its storage media to ensure the data's integrity.

This entry point is only called if you set bit 27 of your image filing system information word when you initialised your image filing system.

FSEntry_Args 7 and ImageEntry_Args 7

Ensure file size

On entry

R0 = 7
R1 = file handle used by your filing system/image filing system
R2 = size of file to ensure

On exit

R2 = size of file actually ensured

Details

This entry point is called by FileSwitch to ensure that a file is of at least the given size. Your file system should do just this, but need not ensure that any extra space is zeroed. All filing systems must support this call.

FSEntry_Args 8 and ImageEntry_Args 8

Write zeros to file

On entry

R0 = 8
R1 = file handle used by your filing system
R2 = file offset at which to write
R3 = number of zero bytes to write

On exit

—

Details

This entry point is called by FileSwitch to request that your filing system writes a given number of zero bytes to a given offset within a file. You should only support this call if your filing system uses buffered files.

The file handle is guaranteed by FileSwitch not to be a directory, and to have had write access granted at the time of the open.

The number of bytes to write, and the file offset at which to write data are guaranteed to be a multiple of the buffer size for this file.

FSEntry_Args 9 and ImageEntry_Args 9

Read file datestamp

On entry

R0 = 9
R1 = file handle used by your filing system/image filing system

On exit

R2 = load address of file (or 0)
R3 = execution address of file (or 0)

Details

This entry point is called by FileSwitch to read the date/time stamp for a given file. The bottom four bytes of the date/time stamp are stored in the execution address of the file. The most significant byte is stored in the least significant byte of the load address. All filing systems must support this call. If your filing system cannot stamp an open file given its handle, then it should return R2 and R3 set to zero.

FSEntry_Args 10

Inform of new image stamp

On entry

R0 = 10
R1 = file handle used by your filing system/image filing system
R2 = new image stamp of image

On exit

All registers preserved

Details

This entry point is called by FileSwitch when an image filing system has changed an image's *image stamp* (a unique identification number). The purpose of the call is to inform your filing system of the change, and to pass it the new image stamp. If your filing system does not support the root object being an image, then it should ignore this call. Otherwise – as for example in the case of FileCore – you should update your filing system's internal note of the image stamp, as you may need to use it to identify the disc at a later time.

This call is for information only, and should not require any further action. It is not called by RISC OS 2, which does not support image filing systems.

FSEntry_Close and ImageEntry_Close

Close an open file

On entry

R1 = file handle used by your filing system/image filing system
R2 = new load address to associate with file
R3 = new execution address to associate with file

On exit

—

Details

This entry point is called by FileSwitch to request that your filing system close an open file, and put a new date/time stamp on it. For ImageEntry_Close, you should then call OS_Args 255 (page 2-62) on the image after updating the structure for the closed file; this ensures that all data is flushed to the disc.

If your filing system returned from the FSEntry_Args 9 (or ImageEntry_Args 9) call with R2 and R3 both zero, then they will also have that value here, and you should not try to restamp the file. Restamping takes place if the file has been modified and FSEntry_Args 9 (or ImageEntry_Args 9) returned a non-zero value in R2.

Note that *Close and *Shut (ie close all open files) are performed by FileSwitch which passes the handles, one at a time, to the relevant filing system for closing. Filing systems should not try to support this themselves.

FSEntry_File and ImageEntry_File

Various calls are made by FileSwitch through these entry points to perform operations on whole files. The actions are specified by R0 as follows:

FSEntry_File 0 and ImageEntry_File 0

Save file

On entry

R0 = 0
R1 = pointer to filename
R2 = load address to associate with file
R3 = execution address to associate with file
R4 = pointer to start of buffer
R5 = pointer to byte after end of buffer
R6 = pointer to special field if present, otherwise 0 (FSEntry_File 0); or image filing system's handle for image that contains file (ImageEntry_File 0)

On exit

R6 = pointer to a leafname for printing *OPT 1 info

Details

This entry point is called by FileSwitch to request that your filing system saves data from a buffer held in memory to a file. FileSwitch has already validated the buffer, and ensured that the leafname is not wildcarded. If the file currently exists and is not locked, the old file is first discarded. The new file should have the same access attributes as the one it is replacing, or some default access if the file doesn't already exist. You should return an error such as `File locked` if you could not save the specified file.

FileSwitch immediately copies the leafname returned in R6, so it need not have a long lifetime. You could hold it in a small static buffer, for example.

FSEntry_File 1 and ImageEntry_File 1

Write catalogue information

On entry

R0 = 1
R1 = pointer to wildcarded filename
R2 = new load address to associate with file
R3 = new execution address to associate with file
R5 = new attributes for file
R6 = pointer to special field if present, otherwise 0 (FSEntry_File 1); or image filing system's handle for image that contains file (ImageEntry_File 1)

On exit

—

Details

This entry point is called by FileSwitch to request that your filing system updates the catalogue information for an object. If the object is a directory you must either write the information (FileCore-based filing systems do) or return an error. You must not return an error if the object does not exist.

FSEntry_File 2

Write load address

On entry

R0 = 2
R1 = pointer to wildcarded filename
R2 = new load address to associate with file
R6 = pointer to special field if present, otherwise 0

On exit

—

Details

This entry point is called by FileSwitch to request that your filing system alters the load address for a file. If the object is a directory you must either write the information (FileCore-based filing systems do) or return an error. You must not return an error if the object does not exist.

FSEntry_File 3

Write execution address

On entry

R0 = 3
R1 = pointer to wildcarded filename
R3 = execution address to associate with file
R6 = pointer to special field if present, otherwise 0

On exit

—

Details

This entry point is called by FileSwitch to request that your filing system alters the execution address for a file. If the object is a directory you must either write the information (FileCore-based filing systems do) or return an error. You must not return an error if the object does not exist.

FSEntry_File 4

Write attributes

On entry

R0 = 4
R1 = pointer to wildcarded pathname
R5 = new attributes to associate with file
R6 = pointer to special field if present, otherwise 0

On exit

—

Details

This entry point is called by FileSwitch to request that your filing system alters the attributes of an object. You must not return an error if the object does not exist.

FSEntry_File 5 and ImageEntry_File 5

Read catalogue information

On entry

R0 = 5
R1 = pointer to pathname
R6 = pointer to special field if present, otherwise 0 (FSEntry_File 5); or image filing system's handle for image that contains file (ImageEntry_File 5)

On exit

R0 = object type:

0	not found
1	file
2	directory

R2 = load address
R3 = execution address
R4 = file length
R5 = file attributes
R6 preserved (ImageEntry_File 5)

Details

This entry point is called by FileSwitch to request that your filing system returns the catalogue information for an object. You should return an error if:

- the pathname specifies a drive that is unknown
- the pathname specifies a media name that is unknown and not made available after any UpCall
- the special field specifies an unknown server or subsystem.

You should return type 0 if:

- the place specified by the pathname exists, but the leafname does not match any object there
- the place specified by the pathname does not exist.

FSEntry_File 6 and ImageEntry_File 6

Delete object

On entry

R0 = 6
R1 = pointer to filename
R6 = pointer to special field if present, otherwise 0 (FSEntry_File 6); or image filing system's handle for image that contains file (ImageEntry_File 6)

On exit

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

Details

This entry point is called by FileSwitch to request that your filing system deletes an object. FileSwitch will already have ensured that the leafname is not wildcarded. No data need be transferred to the file. You should return an error if the object is locked against deletion, but not if the object does not exist. The results refer to the object that was deleted.

FSEntry_File 7 and ImageEntry_File 7

Create file

On entry

R0 = 7
R1 = pointer to filename
R2 = load address to associate with file
R3 = execution address to associate with file
R4 = start address in memory of data
R5 = end address in memory plus one
R6 = pointer to special field if present, otherwise 0 (FSEntry_File 7); or image filing system's handle for image that contains file (ImageEntry_File 7)

On exit

R6 = pointer to a filename for printing *Opt 1 info (FSEntry_File 7 only)

Details

This entry point is called by FileSwitch to request that your filing system creates a file with a given name. R4 and R5 are used only to calculate the length of the file to be created. If the file currently exists and is not locked, the old file is first discarded. The new file should have the same access attributes as the one it is replacing, or some default access if the file doesn't already exist. You should return an error if you couldn't create the file.

FSEntry_File 8 and ImageEntry_File 8

Create directory

On entry

R0 = 8
R1 = pointer to directory name
R2 = load address (ignored by RISC OS 2)
R3 = execute address (ignored by RISC OS 2)
R4 = number of entries (0 for default)
R6 = pointer to special field if present, otherwise 0 (FSEntry_File 8); or image filing system's handle for image that contains file (ImageEntry_File 8)

On exit

—

Details

This entry point is called by FileSwitch to request that your filing system creates a directory. If the directory already exists then your filing system can do one of these:

- return without any modification to the existing directory
- attempt to rename the directory – you must not return an error if this fails.

If directories don't support load and execute addresses (which will only be of the directory type/datestamp form) then no error should be returned. Note that RISC OS 2 will ignore the load and execute addresses in R2 and R3.

FileSwitch will already have ensured that the leafname is not wildcarded. You should return an error if you couldn't create the directory.

FSEntry_File 9

Read catalogue information (no length)

On entry

R0 = 9
R1 = pointer to filename
R6 = pointer to special field if present, otherwise 0

On exit

R0 = object type
R2 = load address
R3 = execution address
R5 = file attributes

Details

This entry point is called by FileSwitch to read the catalogue information for an object, save for the object length. It is useful for NetFS with fileservers, as the length is not stored in a directory. You must not return an error if the object does not exist.

It is only ever called by *Copy under RISC OS 2; bit 26 of your filing system information word must have been set when the filing system was initialised. Otherwise FileSwitch calls FSEntry_File 5, and the length returned in R4 is ignored.

FSEntry_File 10 and ImageEntry_File 10

Read block size

On entry

R0 = 10
R1 = pointer to filename
R6 = pointer to special field if present, otherwise 0 (FSEntry_File 10); or image filing system's handle for image that contains file (ImageEntry_File 10)

On exit

R2 = natural block size of the file (in bytes)

Details

This entry point is called by FileSwitch to read the natural block size for a file (see *FSEntry_Open and ImageEntry_Open* on page 2-541). It is not called by RISC OS 2.

FSEntry_File 255

Load file

On entry

R0 = 255
R1 = pointer to wildcarded filename
R2 = address to load file
R6 = pointer to special file if present; otherwise 0

On exit

R0 corrupted
R2 = load address
R3 = execution address
R4 = file length
R5 = file attributes
R6 = pointer to a filename for printing *OPT 1 info

Details

This entry point is called by FileSwitch to request that your filing system loads a file.

FileSwitch will already have called FSEntry_File 5 and validated the client's load request. If FSEntry_File 5 returned with object type 0 then the user will have been returned the 'File 'xyz' not found' error; type 2 will have returned the ''xyz' is a directory' error; types 1 with corresponding load actions will have had them executed (which may recurse back down to load again), those with no read access will have returned 'Access violation', and those being partially or wholly loaded into invalid memory will have returned 'No writeable memory at this address'.

Therefore unless the filing system is accessing data stored on a multi-user server such as NetFS/FileStore, the object will still be the one whose info was read earlier.

The filename pointed to by R6 on exit should be the non-wildcarded 'leaf' name of the file. That is, if the filename given on entry was `$.!b*`, and the file accessed was the boot file, R6 should point to the string !Boot.

FSEntry_Func and ImageEntry_Func

Various calls are made through these entry points to deal with assorted filing system (or image filing system) control. Many of these output information. You should do this in two stages:

- amass the information into a dynamic buffer
- print from the buffer and dispose of it.

This avoids problems caused by the write character process being in the middle of spooling, or by an active task swapper.

If you add a header to output (cf `*Info`, `*Cat` and `*Ex` on ADFS) you must follow it with a blank line. You should always try to format your output to the printable width of the current window. You can read this using XOS_ReadVduVariables (page 1-730) to read the WindowWidth variable (&100), which copes with most eventualities. Don't cache the value, but read it before each output.

The actions are specified by R0 as given below.

FSEntry_Func 0

Set current directory

On entry

R0 = 0
R1 = pointer to wildcarded directory name
R6 = pointer to special field if present, otherwise 0

On exit

—

Details

This entry point is called by FileSwitch to set the current directory to the one specified by the directory name and context given. If the directory name is null, you should assume it to be the user root directory.

You should not also make the context current, but instead provide an independent means of doing so, such as *FS on the NetFS.

This entry point is called by RISC OS 2; otherwise it is only called to perform a *Opt 1 command when bit 23 of the filing system information word is clear.

FSEntry_Func 1

Set library directory

On entry

R0 = 1
R1 = pointer to wildcarded directory name
R6 = pointer to special field if present, otherwise 0

On exit

—

Details

This entry point is called by FileSwitch to set the current library directory to the one identified by the directory name and context given. If the directory name is null, you should assume it to be the filing system default (which is dependent on your implementation).

You should not also make the context current, but instead provide an independent means of doing so, such as *FS on the NetFS.

This entry point is only called by RISC OS 2.

FSEntry_Func 2

Catalogue directory

On entry

R0 = 2
R1 = pointer to wildcarded directory name
R6 = pointer to special field if present, otherwise 0

On exit

—

Details

This entry point is called by FileSwitch to catalogue the directory identified by the directory name and context given. If the directory name is null, you should assume it to be the current directory. (This corresponds to the *Cat command.)

This entry point is called by RISC OS 2; otherwise it is only called if bit 1 of the extra filing system information word is set.

FSEntry_Func 3

Examine directory

On entry

R0 = 3
R1 = pointer to wildcarded directory name
R6 = pointer to special field if present, otherwise 0

On exit

—

Details

This entry point is called by FileSwitch to print information on all the objects in the directory identified by the directory name and context given. If the directory name is null, you should assume it to be the current directory. (This corresponds to the *Ex command.)

This entry point is called by RISC OS 2; otherwise it is only called if bit2 of the extra filing system information word is set.

FSEntry_Func 4

Catalogue library directory

On entry

R0 = 4
R1 = pointer to wildcarded directory name
R6 = pointer to special field if present, otherwise 0

On exit

—

Details

This entry point is called by FileSwitch to catalogue the specified subdirectory relative to the current library directory. If the directory name is null, you should assume it to be the current library directory. (This corresponds to the *LCat command.)

This entry point is called by RISC OS 2; otherwise it is only called if bit 1 of the extra filing system information word is set.

FSEntry_Func 5

Examine library directory

On entry

R0 = 5
R1 = pointer to wildcarded directory name
R6 = pointer to special field if present, otherwise 0

On exit

—

Details

This entry point is called by FileSwitch to print information on all the objects in the specified subdirectory relative to the current library directory. If the directory name is null, you should assume it to be the current library directory. (This corresponds to the *LEx command.)

This entry point is called by RISC OS 2; otherwise it is only called if bit 2 of the extra filing system information word is set.

FSEntry_Func 6

Examine object(s)

On entry

R0 = 6
R1 = pointer to wildcarded pathname
R6 = pointer to special field if present, otherwise 0.

On exit

—

Details

This entry point is called by FileSwitch to print information on all the objects matching the wildcarded pathname and context given, in the same format as for FSEntry_Func 3. (This corresponds to the *Info command.)

This entry point is called by RISC OS 2; otherwise it is only called if bit 2 of the extra filing system information word is set.

FSEntry_Func 7

Set filing system options

On entry

R0 = 7
R1 = new option (or 0)
R2 = new parameter
R6 = 0 (cannot specify a context)

On exit

—

Details

This entry point is called by FileSwitch to set filing system options.

An option of 0 means reset all filing system options to their default values. An option of 1 is never passed to you, as FileSwitch maintains these settings. An option of 4 is used to set the boot file action. You may use other option numbers for your own purposes; please contact Acorn for an allocation.

(This corresponds to the *Opt command.)

You should return an error for bad combinations of options and parameters.

FSEntry_Func 8 and ImageEntry_Func 8

Rename object

On entry

R0 = 8
R1 = pointer to pathname of object to be renamed
R2 = pointer to new pathname for object
R6 = pointer to first special field if present, otherwise 0 (FSEntry_Func 8); or image filing system's handle for image that contains file (ImageEntry_Func 8)
R7 = pointer to second special field if present, else 0 (FSEntry_Func 8 only)

On exit

R1 = 0 if rename performed (≠0 otherwise)

Details

This entry point is called by FileSwitch to attempt to rename an object. If the rename is not 'simple' – ie just changing the file's catalogue entry – R1 should be returned with a value other than zero. (For example, the files may be on different images.) In such cases, FileSwitch will return a 'Bad rename' error.

FSEntry_Func 9

Access object(s)

On entry

R0 = 9
R1 = pointer to wildcarded pathname
R2 = pointer to access string (null, space or control-character terminated)
R6 = pointer to special field if present, otherwise 0.

On exit

—

Details

This entry point is called by FileSwitch to give the requested access to all objects matching the wildcarded name given. (This corresponds to the *Access command.)

You should ignore inappropriate owner access bits, and try to store public access bits.

This entry point is called by RISC OS 2; otherwise it is only called if bit 18 of the filing system information word is set.

FSEntry_Func 10

Boot filing system

On entry

R0 = 10

On exit

—

Details

This entry point is called by FileSwitch to request that your filing system performs its boot action.

For example, ADFS examines the boot option – as set by *Opt 4 – of the disc in the configured drive and acts accordingly (so, if boot option 2 is set, it will `*Run &.!Boot`); whereas NetFS attempts to logon as the boot user to the configured file server.

This call may not return if it runs an application.

FSEntry_Func 11

Read name and boot (*OPT 4) option of disc

On entry

R0 = 11
R2 = pointer to buffer in which to put data
R6 = 0 (cannot specify a context)

On exit

—

Details

This entry point is called by FileSwitch to obtain the name of the disc that the CSD is on in the temporary filing system, and its boot option. This data should be returned in the area of memory pointed to by R2, in the following format:

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

If there is no CSD, this call should return the string ‘Unset’ for the disc name, and the boot action should be set to zero.

The buffer pointed to by R2 will not have been validated with OS_ValidateAddress, because FileSwitch doesn’t know how big the buffer has to be. It is the filing system’s responsibility to validate any buffer that it uses, and to return an error if the memory required is not valid. Under RISC OS 2 it should use the error text ‘No writable memory at this address’; under later versions it should instead look up the token BadWrt.

The buffer pointed to by R2 will not have been validated and so you should be prepared for faulting when you write to the memory. You must not put an interlock on when you are doing so.

FSEntry_Func 12

Read current directory name and privilege byte

On entry

R0 = 12
R2 = pointer to buffer in which to put data
R6 = 0 (cannot specify a context)

On exit

—

Details

This entry point is called by FileSwitch to obtain the name of the CSD on the temporary filing system, and privilege status in relation to that directory. This data should be returned in the area of memory pointed to by R2, in the following format:

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

If there is no CSD, this call should return the string ‘Unset’ for the directory name.

The privilege byte is &00 if you have ‘owner’ status (ie you 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 FileCore-based filing systems, you always have owner status.

The buffer pointed to by R2 will not have been validated with OS_ValidateAddress, because FileSwitch doesn’t know how big the buffer has to be. It is the filing system’s responsibility to validate any buffer that it uses, and to return the error ‘No writable memory at this address’ if the memory required is not valid.

This entry point is only called by RISC OS 2.

FSEntry_Func 13

Read library directory name and privilege byte

On entry

R0 = 13
R2 = pointer to buffer in which to put data
R6 = 0 (cannot specify a context)

On exit

—

Details

This entry point is called by FileSwitch to obtain the name of the library directory on the temporary filing system, and privilege status in relation to that directory. This data should be returned in the area of memory pointed to by R2, in the following format:

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

If no library is selected, this call should return the string ‘Unset’ for the library directory name.

The buffer pointed to by R2 will not have been validated with OS_ValidateAddress, because FileSwitch doesn’t know how big the buffer has to be. It is the filing system’s responsibility to validate any buffer that it uses, and to return the error ‘No writable memory at this address’ if the memory required is not valid.

This entry point is only called by RISC OS 2.

FSEntry_Func 14 and ImageEntry_Func 14

Read directory entries

On entry

R0 = 14
R1 = pointer to wildcarded directory name
R2 = pointer to buffer in which to put data
R3 = number of object names to read
R4 = offset of first item to read in directory (0 for start of directory)
R5 = length of buffer
R6 = pointer to special field if present, otherwise 0 (FSEntry_Func 14); or image filing system’s handle for image that contains file (ImageEntry_Func 14)

On exit

R3 = number of names read
R4 = offset of next item to read in directory (–1 if end)

Details

This entry point is called by FileSwitch to read the leaf names of entries in a directory into an area of memory pointed to by R2. If the directory name is null, then for filing systems the currently-selected directory should be read; for image filing systems the root directory should be read. The names are returned in the buffer as a list of null terminated strings. You must not overflow the end of the buffer, and you must only count names that you have completely inserted.

The length of buffer that FileSwitch will have validated depends on the call that was made to it:

- if it was OS_GBPB 8, then enough space will have been validated to hold [R3] 10-character long directory entries (plus their terminators)
- if it was OS_GBPB 9, then the entire buffer specified by R2 and R5 will have been validated.

Unfortunately there is no way you can tell which was used. RISC OS programmers are encouraged to use the latter.

You should return an error if the object being catalogued is not found or is a file. The following are, however, all valid return values:

- R3 = 0, R4 ≠ –1 (the buffer overflowed)
- R3 ≠ 0, R4 ≠ –1 (there are more names to read)
- R3 = 0, R4 = –1 (the previous read filled the buffer with the last name, but didn’t detect the end; now there no more names to read).

FSEntry_Func 15 and ImageEntry_Func 15

Read directory entries and information

On entry

R0 = 15
R1 = pointer to wildcarded directory name
R2 = pointer to buffer in which to put data
R3 = number of object names to read
R4 = offset of first item to read in directory (0 for start of directory)
R5 = length of buffer
R6 = pointer to special field if present, otherwise 0 (FSEntry_Func 15); or image filing system’s handle for image that contains file (ImageEntry_Func 15)

On exit

R3 = number of records read
R4 = offset of next item to read in directory (–1 if end)

Details

This entry point is called by FileSwitch to read the leaf names of entries (and their file information) in the given directory into a buffer pointed to by R2. If the directory name is null, then the currently-selected directory should be read. The names and information are returned in records, with the following format:

Offset	Contents
&00	Load address
&04	Execution address
&08	Length
&0C	Attributes
&10	Object type
&14	Object name

FileSwitch will have validated the buffer. You must not overflow the end of the buffer, and you must only count names that you have completely inserted. You should assume that the buffer is word-aligned, and your records should be so too. You may find this code fragment useful to do so:

```
ADD     R2, R2, #p2-1   ; p2 is a power-of-two, in this case 4
BIC     R2, R2, #p2-1
```

You should return an error if the object being catalogued is not found or is a file.

FSEntry_Func 16

Shut down

On entry

R0 = 16

On exit

—

Details

This entry point is called by FileSwitch to request that your filing system go into as dormant a state as possible. For example, it should place hard drives in their transit positions, etc. All files will have been closed by FileSwitch before this call is issued.

FSEntry_Func 17

Print start up banner

On entry

R0 = 17
R6 = 0 (cannot specify a context)

On exit

—

Details

This entry point is called by FileSwitch to print out a filing system banner that shows which filing system is selected. FileSwitch calls it if it receives a reset service call and the text offset value (in the filing system information block) is –1. This is to allow filing systems to print a message that may vary, such as `Acorn Econet` or `Acorn Econet no clock`.

You should print the string using XOS_… SWIs, and if there is an error return with V set and R0 pointing to an error block. This is not likely to happen.

FSEntry_Func 18

Set directory contexts

Details

This entry point is never called by FileSwitch.

FSEntry_Func 19

Read directory entries and information

On entry

R0 = 19
R1 = pointer to wildcarded directory name
R2 = pointer to buffer in which to put data
R3 = number of object names to read
R4 = offset of first item to read in directory
R5 = length of buffer
R6 = pointer to special field if present, otherwise 0

On exit

R3 = number of records read
R4 = offset of next item to read in directory (–1 if end)

Details

This entry point is called by FileSwitch to read the names of entries (and their file information) in the given directory into a buffer pointed to by R2. If the directory name is null, then the currently-selected directory should be read. The names and information are returned in records, with the following format:

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

Each record is word-aligned.

FSEntry_Func 20

Output full information on object(s)

On entry

R0 = 20
R1 = pointer to pathname (may be wildcarded under RISC OS 2 only)
R6 = pointer to special field if present, otherwise 0

On exit

—

Details

This entry point is called by FileSwitch to request that your filing system outputs full information on the given object (or, under RISC OS 2, on all the objects matching the wildcarded pathname). The format must be the same as for the *FileInfo command.

It is only called by FileSwitch if bit 25 of the filing system information word was set when the filing system was initialised. Otherwise FileSwitch will use calls to FSEntry_Func 6 to implement *FileInfo.

ImageEntry_Func 21

Notification of new image

On entry

R0 = 21
R1 = FileSwitch handle to the file
R2 = buffer size for file if known, otherwise 0

On exit

R1 = image filing system's handle for image

Details

This entry point is called by FileSwitch to notify your image filing system that FileSwitch would like it to handle a new image. This entry gives the image filing system a chance to set up internal structures so that data could be cached or buffered from the image. All future requests FileSwitch makes of the image filing system will quote the returned image filing system's handle for the image when appropriate.

The image should be flagged internally as ‘stamp image on next update’, and when it is updated its unique identification number should be updated. Whenever this number is updated the host filing system should be informed of its new value using OS_Args 8 – this is important, because otherwise the host filing system will lose track of which disc is which.

The buffer size (if given) should be treated as a hint to the sector size.

This entry point is not called by RISC OS 2.

ImageEntry_Func 22

Notification that image is about to be closed

On entry

R0 = 22
R1 = image filing system’s handle for image

On exit

—

Details

This entry point is called by FileSwitch to notify your image filing system that an image is about to be closed. All files will have been closed for you before this call is made. You should save any buffered data for this image before returning, and discard any cached data.

This entry point is not called by RISC OS 2.

FSEntry_Func 23

Canonicalise special field and disc name

On entry

R0 = 23
R1 = pointer to special field if present, otherwise 0
R2 = pointer to disc name if present, otherwise 0
R3 = pointer to buffer to hold canonical special field, or 0 to return required length
R4 = pointer to buffer to hold canonical disc name, or 0 to return required length
R5 = length of buffer to hold canonical special field
R6 = length of buffer to hold canonical disc name

On exit

R1 = pointer to canonical special field if present, otherwise 0
R2 = pointer to canonical disc name if present, otherwise 0
R3 = bytes overflow from special field buffer (ie required length if R3 = 0 on entry)
R4 = bytes overflow from special field buffer (ie required length if R4 = 0 on entry)
R5, R6 preserved

Details

This entry point is called by FileSwitch to convert the given special field and disc name to canonical (unique) forms. If no buffers are passed to hold the results, this call instead returns their required lengths, which gives FileSwitch a means of finding out this information.

FileSwitch uses this call to convert user-specified special field and disc names into a canonical (unique) form. Typically this call is used in two stages: the first to find out how much space is required in the buffers, and the second to do the conversion. For example, if a user specifies a file as NetFS#Arf:&.thing.whatsit, FileSwitch uses this call as follows:

R1 = pointer to the string 'Arf'
R2 = 0
R3 = 0
R4 = 0
R5 = any value (since R3 = 0)
R6 = any value (since R4 = 0)

NetFS returns these values:

R1 = any non-zero value
R2 = any non-zero value
R3 = required length of buffer to hold canonical special field (excluding any terminating null)
R4 = required length of buffer to hold canonical disc name (excluding any terminating null)
R5, R6 preserved

FileSwitch now allocates memory for two buffers of the lengths specified by NetFS in the R3 and R4 return values, then call NetFS again as follows:

R1 = pointer to the string 'Arf'
R2 = 0
R3 = pointer to a buffer of length R5 bytes
R4 = pointer to a buffer of length R6 bytes
R5 = length of buffer pointed to by R3
R6 = length of buffer pointed to by R4

NetFS now fills in the buffers: (R3,R5) with the special field, and (R4,R6) with the disc name. It returns:

R1 = R3 on entry (and the buffer is filled with '49.254')
R2 = R4 on entry (and the buffer is filled in with 'Arf')
R3, R4 = 0 (no overflows over the end of the buffers)
R5, R6 preserved

This entry point is not called by RISC OS 2, and is only otherwise called if bit 23 of the filing system information word is set.

FSEntry_Func 24

Resolve wildcard

On entry

R1 = pointer to directory pathname
R2 = pointer to buffer to hold resolved name, or 0 if none
R3 = pointer to wildcarded object name
R5 = length of buffer
R6 = pointer to special field if present, otherwise 0

On exit

R1 preserved
R2 = –1 if not found, else preserved
R3 preserved
R4 = –1 if FileSwitch should resolve this wildcard itself, else bytes overflow from buffer
R5 preserved

Details

This entry point is called by FileSwitch to find which object in the given directory matches the name given. If the filing system can not do a more efficient job than FileSwitch would if it were to use FSEntry_Func 14 and then to find which was the first match, then the filing system should just return with R4 = –1.

This entry point is not called by RISC OS 2, and is only otherwise called if bit 23 of the filing system information word is set.

FSEntry_Func 25 and ImageEntry_Func 25

Read defect list

On entry

R0 = 25
R1 = pointer to name of image (FSEntry_Func 25 only)
R2 = pointer to buffer
R5 = length of buffer
R6 = pointer to special field if present, otherwise 0 (FSEntry_Func 25); or image filing system's handle for image (ImageEntry_Func 25)

On exit

R0 - R6 preserved

Details

This entry point is called by FileSwitch to request that your filing system fills the given buffer with the byte offsets to the start of any defects in the specified image. The list must be terminated by the value &20000000.

It is an error if the specified image is not the root object in an image (eg it is an error to map out a defect from adfs::HardDisc4.$.fred, but not an error to map it out from adfs::HardDisc4.$).

This entry point is not called by RISC OS 2.

FSEntry_Func 26 and ImageEntry_Func 26

Add a defect

On entry

R0 = 26
R1 = pointer to name of image (FSEntry_Func 26 only)
R2 = byte offset to start of defect
R6 = pointer to special field if present, otherwise 0 (FSEntry_Func 26); or image filing system's handle for image (ImageEntry_Func 26)

On exit

R0 - R2, R6 preserved

Details

This entry point is called by FileSwitch to request that your filing system maps out the given defect from the specified image.

It is an error if the specified image is not the root object in an image (eg it is an error to map out a defect from adfs::HardDisc4.$.fred, but not an error to map it out from adfs::HardDisc4.$). If the defect cannot be mapped out because it is not free, then you should return an error.

This entry point is not called by RISC OS 2.

FSEntry_Func 27 and ImageEntry_Func 27

Read boot option

On entry

R0 = 27
R1 = pointer to pathname of any object on image (FSEntry_Func 27 only)
R6 = pointer to special field if present, otherwise 0 (FSEntry_Func 27); or image filing system's handle for image (ImageEntry_Func 27)

On exit

R0, R1, R6 preserved
R2 = boot option (as in *Opt 4,n)

Details

This entry point is called by FileSwitch to read the boot option (ie the value *n* in *Opt 4,*n*) of the image that holds the object specified by R1 (FSEntry_Func 27), or that is specified by the handle in R6 (ImageEntry_Func 27).

This entry point is not called by RISC OS 2.

FSEntry_Func 28 and ImageEntry_Func 28

Write boot option

On entry

R0 = 28
R1 = pointer to pathname of any object on image (FSEntry_Func 28 only)
R2 = new boot option
R6 = pointer to special field if present, otherwise 0 (FSEntry_Func 28); or image filing system's handle for image (ImageEntry_Func 28)

On exit

R0 - R2, R6 preserved

Details

This entry point is called by FileSwitch to request that your filing system writes the boot option (ie the value *n* in *Opt 4,*n*) of the image that holds the object specified by R1 (FSEntry_Func 28), or that is specified by the handle in R6 (ImageEntry_Func 28).

This entry point is not called by RISC OS 2.

FSEntry_Func 29 and ImageEntry_Func 29

Read used space map

On entry

R0 = 29
R1 = pointer to pathname of any object on image (FSEntry_Func 29 only)
R2 = pointer to buffer for map (pre-filled with 0s)
R5 = size of buffer
R6 = pointer to special field if present, otherwise 0 (FSEntry_Func 29); or image filing system's handle for image (ImageEntry_Func 29)

On exit

R0 - R2, R5, R6 preserved

Details

This entry point is called by FileSwitch to read the used space map for the image that holds the object specified by R1 (FSEntry_Func 29), or that is specified by the handle in R6 (ImageEntry_Func 29). It is used by the *Backup command to decide which sectors to copy.

Your filing system should fill the given buffer with 0 bits for unused blocks, and 1 bits for used blocks. The buffer must be filled to its limit, or to the image's limit, whichever is less. The 'perfect' size of the buffer can be calculated from the image's size and its block size (as returned from FSEntry_Open or ImageEntry_Open: see page 2-541). 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.

This entry point is not called by RISC OS 2.

FSEntry_Func 30 and ImageEntry_Func 30

Read free space

On entry

R0 = 30
R1 = pointer to pathname of any object on image (FSEntry_Func 30 only)
R6 = pointer to special field if present, otherwise 0 (FSEntry_Func 30); or image filing system's handle for image (ImageEntry_Func 30)

On exit

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

Details

This entry point is called by FileSwitch to read the free space for the image that holds the object specified by R1 (FSEntry_Func 30), or that is specified by the handle in R6 (ImageEntry_Func 30).

This entry point is not called by RISC OS 2.

FSEntry_Func 31 and ImageEntry_Func 31

Name image

On entry

R0 = 31
R1 = pointer to pathname of any object on image (FSEntry_Func 31 only)
R2 = pointer to new name of image
R6 = pointer to special field if present, otherwise 0 (FSEntry_Func 31); or image filing system's handle for image (ImageEntry_Func 31)

On exit

Registers preserved

Details

This entry point is called by FileSwitch to request that your filing system name the image that holds the object specified by R1 (FSEntry_Func 31), or that is specified by the handle in R6 (ImageEntry_Func 31).

This refers to the image's name (eg a disc name), rather than the name of the file containing that image.

This entry point is not called by RISC OS 2.

FSEntry_Func 32 and ImageEntry_Func 32

Stamp image

On entry

R0 = 32
R1 = pointer to pathname of any object on image (FSEntry_Func 32 only)
R2 = reason code
R6 = pointer to special field if present, otherwise 0 (FSEntry_Func 32); or image filing system's handle for image (ImageEntry_Func 32)

On exit

Registers preserved

Details

This entry point is called by FileSwitch to request that your filing system stamp the image that holds the object specified by R1 (FSEntry_Func 32), or that is specified by the handle in R6 (ImageEntry_Func 32). It is used for FileCore to communicate with an image filing system for the control and management of the disc Id of a given image. Valid values for R2 on entry are:

Value	Meaning
0	stamp image on next update
1	stamp image now

To stamp an image the image's unique identification number should be updated to a different value. This value is used to distinguish between different images with the same name, and to determine when a given image has been updated. It should be filled in the disc record disc id field when the disc is originally identified. The kind of uses expected for these calls are:

- When a Backup program wishes to cause a backup of the original to be distinguishable from the original it may use the 'stamp image now' form.

and, **for ImageEntry_Func 32 only**, the following two uses:

- When FileCore notices that a given disc may have been removed from the drive it will call the image filing system (via FileSwitch) with the 'stamp image on next update' call. This informs the image filing system that when it next changes something in that image that it should also explicitly change the unique Id number (if possible). This so that if another machine saw the disc whilst it was removed, then the changed that other machine will be given a clue that the disc has since been changed by the Id number changing – the other machine will probably discard any cached data it has as none of it could be trusted to still be accurate. Once the Id has been updated once there is no further need to update it on an update unless, of course, a further 'stamp image on next update' occurs.
- When FileCore is explicitly requested to stamp a disc it will use the 'stamp image now' call to get the message through to the relevant image filing system.

This entry point is not called by RISC OS 2.

FSEntry_Func 33

Get usage of offset

On entry

R0 = 33
R1 = pointer to pathname of any object on image (FSEntry_Func 33 only)
R2 = byte offset into image
R3 = pointer to buffer to receive object name (if object found)
R4 = length of buffer
R6 = pointer to special field if present, otherwise 0 (FSEntry_Func 33); or image filing system's handle for image (ImageEntry_Func 33)

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
2	object found; cannot share the offset with other objects
3	object found; can share the offset with other objects

Details

This entry point is called by FileSwitch to find the usage of the given offset within the image that holds the object specified by R1 (FSEntry_Func 33), or that is specified by the handle in R6 (ImageEntry_Func 33). If the offset is free, a defect or outside the image then you should return with R2 = 0. If the offset is used, but has no object name which corresponds to it (for example the free space map, FAT tables, boot block and the such), then return with R2 = 1. If the given offset is associated with only one object (such that deleting that object would definitely free the given offset), then you should return with R2 = 2. If the offset is associated with several objects (files/directories), but cannot be said to be associated with one only (for example, the disc may have one large section allocated which is used by several files within one directory), then return with R2 = 3.

You may corrupt the buffer during the search and, if you find an object (ie R2 = 2 or 3), you should return its pathname in the buffer. The pathname should not have a '$' prefix, but the first path element should have a '.' prefix, eg:

```
.a.b.c.d
```

rather than:

```
a.b.c.d
```

This entry point is not called by RISC OS 2.

FSEntry_Func 34

Notification of changed directory

On entry

R1 = pointer to null-terminated directory name
R2 = changed directory (0 ⇒ CSD, 1 ⇒ PSD, 2 ⇒ URD, 3 ⇒ Lib)
R6 = pointer to special field if present, otherwise 0

On exit

R1, R2, R6 preserved

Details

This entry point is provided so that filing systems can optimise their handling of directory caches. It is called when FileSwitch has successfully changed a directory, as indicated by R2.There is no reason for a filing system to have these directories stored, but even if it does it should not change its record of the directory; instead it should use this information to help it decide which directories to cache, and which not to.

FSEntry_GBPB

Get/put bytes from/to an unbuffered file

This entry point is used to implement multiple get byte and put byte operations on unbuffered files. It is only ever called if you set bit 28 of the file information word on return from FSEntry_Open, and you need not otherwise provide it. FileSwitch will instead use multiple calls to FSEntry_PutBytes and FSEntry_GetBytes to implement these operations.

FSEntry_GBPB 1 and 2

Put multiple bytes to an unbuffered file

On entry

R0 = 1 or 2
R1 = file handle used by your filing system
R2 = pointer to buffer

R3 = number of bytes to put to file
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 = number of bytes not transferred
R4 = initial file pointer + number of bytes transferred

Details

This entry point is called by FileSwitch to request that your filing system transfer data 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 you must fill the file with zeros between the current file extent and the specified pointer before the bytes are transferred.

The file handle is guaranteed by FileSwitch not to be a directory, and to have had write access granted at the time of the open.

FSEntry_GBPB 3 and 4

Read bytes from an open file

On entry

R0 = 3 or 4
R1 = file handle used by your filing system
R2 = pointer to buffer
R3 = number of bytes to get from file
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

Details

This entry point is called by FileSwitch to request that your filing system transfer data from a file to memory, either from the specified file pointer (R0 = 3), or from the current one (R0 = 4).

If the specified pointer is greater than or equal to the current file extent then you must not update the sequential file pointer, nor must you return an error.

The file handle is guaranteed by FileSwitch not to be a directory and to have had read access granted at the time of the open.

Your filing system must not try to keep its own *EOF-error-on-next-read* flag – instead it is FileSwitch's responsibility to keep the *EOF-error-on-next-read* flag. Unlike FSEntry_GetBytes, FileSwitch will set the C bit before it returns to its caller if your filing system returns a non-zero value in R3 – so your filing system need not handle this either.

Example program

This code fragment is an optimised routine for moving blocks of memory. It could be further enhanced to take advantage of the higher speed of memory access given by the MEMC chip if `LDM` and `STM` instructions are quad-word aligned. You should find this useful when writing your own filing systems, as efficient transfer code is **crucial** to the performance of a filing system.

```
; +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;
; MoveBytes(source, dest, size in bytes) - fast data copier from RCM
; =========

; SKS Reordered registers and order of copying to suit FileSwitch

; **Not yet optimised to do transfers to make most of 1N,3S feature of MEMC**

; extern void MoveBytes(void *source, void *destination, size_t count);

; In:   r1 = src^ (byte address)
;       r2 = dst^ (byte address)
;       r3 = count (byte count - never zero!)

; Out:  r0-r3, lr corrupt. Flags preserved

mbsrc1          RN 0
mbsrcptr        RN 1
mbdstptr        RN 2
mbcnt           RN 3
mbsrc2          RN 14                   ; Note deviancy, so care in LDM/STM
mbsrc3          RN 4
mbsrc4          RN 5
mbsrc5          RN 6
mbsrc6          RN 7
mbsrc7          RN 8
mbsrc8          RN 9
mbsrc9          RN 10
mbshftL         RN 11                   ; These two go at end to save a word
mbshftR         RN 12                   ; and an extra Pull lr!
sp              RN 13
lr              RN 14
pc              RN 15

MoveBytes ROUT

        STMDB   sp!, {lr}

        TST     mbdstptr, #3
        BNE     MovByt100               ; [dst^ not word aligned]

MovByt20                                ; dst^ now word aligned.
                                        ; branched back to from below
```

```
        TST     mbsrcptr, #3
        BNE     MovByt200               ; [src^ not word aligned]

; src^ & dst^ are now both word aligned
; count is a byte value (may not be a whole number of words)

; Quick sort out of what we've got left to do

        SUBS    mbcnt, mbcnt, #4*4      ; Four whole words to do (or more) ?
        BLT     MovByt40                ; [no]

        SUBS    mbcnt, mbcnt, #8*4-4*4  ; Eight whole words to do (or more) ?
        BLT     MovByt30                ; [no]

        STMDB   sp!, {mbsrc3-mbsrc8}    ; Push some more registers

MovByt25
        LDMIA   mbsrcptr!, {mbsrc1, mbsrc3-mbsrc8, mbsrc2}      ; NB. Order!
        STMIA   mbdstptr!, {mbsrc1, mbsrc3-mbsrc8, mbsrc2}

        SUBS    mbcnt, mbcnt, #8*4
        BGE     MovByt25; [do another 8 words]

        CMP     mbcnt, #-8*4            ; Quick test rather than chaining down
        LDMEQDB sp!, {mbsrc3-mbsrc8, pc}^       ; [finished]
        LDMDB   sp!, {mbsrc3-mbsrc8}


MovByt30
        ADDS    mbcnt, mbcnt, #8*4-4*4  ; Four whole words to do ?
        BLT     MovByt40

        STMDB   sp!, {mbsrc3-mbsrc4}    ; Push some more registers

        LDMIA   mbsrcptr!, {mbsrc1, mbsrc3-mbsrc4, mbsrc2}      ; NB. Order!
        STMIA   mbdstptr!, {mbsrc1, mbsrc3-mbsrc4, mbsrc2}

        LDMEQDB sp!, {mbsrc3-mbsrc4, pc}^       ; [finished]
        LDMDB   sp!, {mbsrc3-mbsrc4}

        SUB     mbcnt, mbcnt, #4*4


MovByt40
        ADDS    mbcnt, mbcnt, #4*4-2*4  ; Two whole words to do ?
        BLT     MovByt50

        LDMIA   mbsrcptr!, {mbsrc1, mbsrc2}
        STMIA   mbdstptr!, {mbsrc1, mbsrc2}

        LDMEQDB sp!, {pc}^      ; [finished]

        SUB     mbcnt, mbcnt, #2*4
```

```
MovByt50
        ADDS    mbcnt, mbcnt, #2*4-1*4  ; One whole word to do ?
        BLT     MovByt60

        LDR     mbsrc1, [mbsrcptr], #4
        STR     mbsrc1, [mbdstptr], #4

        LDMEQDB sp!, {pc}^              ; [finished]

        SUB     mbcnt, mbcnt, #1*4


MovByt60
        ADDS    mbcnt, mbcnt, #1*4-0*4  ; No more to do ?
        LDMEQDB sp!, {pc}^              ; [finished]


        LDR     mbsrc1, [mbsrcptr]      ; Store remaining 1, 2 or 3 bytes
MovByt70
        STRB    mbsrc1, [mbdstptr], #1
        MOV     mbsrc1, mbsrc1, LSR #8
        SUBS    mbcnt, mbcnt, #1
        BGT     MovByt70

        LDMDB   sp!, {pc}^              ; [finished]


; Initial dest^ not word aligned. Loop doing bytes (1,2 or 3) until it is

MovByt100
        LDRB    mbsrc1, [mbsrcptr], #1
        STRB    mbsrc1, [mbdstptr], #1
        SUBS    mbcnt, mbcnt, #1
        LDMEQDB sp!, {pc}^              ; [finished after 1..3 bytes]

        TST     mbdstptr, #3
        BNE     MovByt100

        B       MovByt20                ; Back to mainline code



MovByt200 ; dst^ now word aligned, but src^ isn't. just lr stacked here

        STMDB    sp!, {mbshftL, mbshftR}; Need more registers this section

        AND     mbshftR, mbsrcptr, #3   ; Offset
        BIC     mbsrcptr, mbsrcptr, #3  ; Align src^

        MOV     mbshftR, mbshftR, LSL #3    ; rshft = 0, 8, 16 or 24 only
        RSB     mbshftL, mbshftR, #32       ; lshft = 32, 24, 16 or 8  only

        LDR     mbsrc1, [mbsrcptr], #4
        MOV     mbsrc1, mbsrc1, LSR mbshftR    ; Always have mbsrc1 prepared
```

```
; Quick sort out of what we've got left to do

        SUBS    mbcnt, mbcnt, #4*4      ; Four whole words to do (or more) ?
        BLT     MovByt240               ; [no]

        SUBS    mbcnt, mbcnt, #8*4-4*4  ; Eight whole words to do (or more) ?
        BLT     MovByt230               ; [no]

        STMDB   sp!, {mbsrc3-mbsrc9}    ; Push some more registers

MovByt225
        LDMIA   mbsrcptr!, {mbsrc3-mbsrc9, mbsrc2}              ; NB. Order!
        ORR     mbsrc1, mbsrc1, mbsrc3, LSL mbshftL

        MOV     mbsrc3, mbsrc3, LSR mbshftR
        ORR     mbsrc3, mbsrc3, mbsrc4, LSL mbshftL

        MOV     mbsrc4, mbsrc4, LSR mbshftR
        ORR     mbsrc4, mbsrc4, mbsrc5, LSL mbshftL

        MOV     mbsrc5, mbsrc5, LSR mbshftR
        ORR     mbsrc5, mbsrc5, mbsrc6, LSL mbshftL

        MOV     mbsrc6, mbsrc6, LSR mbshftR
        ORR     mbsrc6, mbsrc6, mbsrc7, LSL mbshftL

        MOV     mbsrc7, mbsrc7, LSR mbshftR
        ORR     mbsrc7, mbsrc7, mbsrc8, LSL mbshftL

        MOV     mbsrc8, mbsrc8, LSR mbshftR
        ORR     mbsrc8, mbsrc8, mbsrc9, LSL mbshftL

        MOV     mbsrc9, mbsrc9, LSR mbshftR
        ORR     mbsrc9, mbsrc9, mbsrc2, LSL mbshftL

        STMIA   mbdstptr!, {mbsrc1, mbsrc3-mbsrc9}

        MOV     mbsrc1, mbsrc2, LSR mbshftR     ; Keep mbsrc1 prepared

        SUBS    mbcnt, mbcnt, #8*4
        BGE     MovByt225               ; [do another 8 words]

        CMP     mbcnt, #-8*4            ; Quick test rather than chaining down
        LDMEQDB sp!, {mbsrc3-mbsrc9, mbshftL, mbshftR, pc}^     ; [finished]
        LDMDB   sp!, {mbsrc3-mbsrc9}


MovByt230
        ADDS    mbcnt, mbcnt, #8*4-4*4  ; Four whole words to do ?
        BLT     MovByt240

        STMDB   sp!, {mbsrc3-mbsrc5}; Push some more registers

        LDMIA   mbsrcptr!, {mbsrc3-mbsrc5, mbsrc2}              ; NB. Order!
        ORR     mbsrc1, mbsrc1, mbsrc3, LSL mbshftL
```

```
        MOV     mbsrc3, mbsrc3, LSR mbshftR
        ORR     mbsrc3, mbsrc3, mbsrc4, LSL mbshftL

        MOV     mbsrc4, mbsrc4, LSR mbshftR
        ORR     mbsrc4, mbsrc4, mbsrc5, LSL mbshftL

        MOV     mbsrc5, mbsrc5, LSR mbshftR
        ORR     mbsrc5, mbsrc5, mbsrc2, LSL mbshftL

        STMIA   mbdstptr!, {mbsrc1, mbsrc3-mbsrc5}

        LDMEQDB sp!, {mbsrc3-mbsrc5, mbshftL, mbshftR, pc}^     ; [finished]
        LDMDB   sp!, {mbsrc3-mbsrc5}

        SUB     mbcnt, mbcnt, #4*4
        MOV     mbsrc1, mbsrc2, LSR mbshftR     ; Keep mbsrc1 prepared


MovByt240
        ADDS    mbcnt, mbcnt, #2*4      ; Two whole words to do ?
        BLT     MovByt250

        STMDB   sp!, {mbsrc3}; Push another register

        LDMIA   mbsrcptr!, {mbsrc3, mbsrc2}                     ; NB. Order!
        ORR     mbsrc1, mbsrc1, mbsrc3, LSL mbshftL

        MOV     mbsrc3, mbsrc3, LSR mbshftR
        ORR     mbsrc3, mbsrc3, mbsrc2, LSL mbshftL

        STMIA   mbdstptr!, {mbsrc1, mbsrc3}

        LDMEQDB sp!, {mbsrc3, mbshftL, mbshftR, pc}^            ; [finished]
        LDMDB   sp!, {mbsrc3}

        SUB     mbcnt, mbcnt, #2*4
        MOV     mbsrc1, mbsrc2, LSR mbshftR     ; Keep mbsrc1 prepared


MovByt250
        ADDS    mbcnt, mbcnt, #2*4-1*4; One whole word to do ?
        BLT     MovByt260

        LDR     mbsrc2, [mbsrcptr], #4
        ORR     mbsrc1, mbsrc1, mbsrc2, LSL mbshftL
        STR     mbsrc1, [mbdstptr], #4

        LDMEQDB sp!, {mbshftL, mbshftR, pc}^                    ; [finished]

        SUB     mbcnt, mbcnt, #1*4
        MOV     mbsrc1, mbsrc2, LSR mbshftR     ; Keep mbsrc1 prepared


MovByt260
```

```
        ADDS    mbcnt, mbcnt, #1*4-0*4
        LDMEQDB sp!, {mbshftL, mbshftR, pc}^                    ; [finished]


        LDR     mbsrc2, [mbsrcptr]      ; Store remaining 1, 2 or 3 bytes
        ORR     mbsrc1, mbsrc1, mbsrc2, LSL mbshftL
MovByt270
        STRB    mbsrc1, [mbdstptr], #1
        MOV     mbsrc1, mbsrc1, LSR #8
        SUBS    mbcnt, mbcnt, #1
        BGT     MovByt270

        LDMDB   sp!, {mbshftL, mbshftR, pc}^

; ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

        END
```

45 Writing a FileCore module

Adding your own module to FileCore

FileCore does not know how to communicate directly with the hardware that your filing system uses. Your module must provide these facilities, and declare the entry points to FileCore.

This chapter describes how to add a filing system to FileCore. You should also see the chapter entitled *Modules* on page 1-201 for more information on how to write a module.

Declaring your module

When your module initialises, it must inform FileCore of its existence. You must call FileCore_Create to do this – see page 2-228 for details. R0 tells FileCore where to find a *descriptor block*. This in turn tells FileCore the locations of all the entry points to your module's low level routines that interface with the hardware:

Descriptor block

This table shows the offsets from the start of the descriptor block, and the meaning of each word in the block:

Offset	Contains
0	Bit flags
3	Filing system number (see the chapter entitled *FileSwitch*)
4	Offset of filing system title from module base
8	Offset of boot text from module base
12	Offset of low-level disc op entry from module base
16	Offset of low-level miscellaneous entry from module base

The flag bits in the descriptor block have the following meanings:

Bit	Meaning when set
0	Hard discs need FIQ
1	Floppy discs need FIQ
2	Reserved – must be zero
3	Use only scratch space when a temporary buffer is needed
4	Hard discs support mount like floppies do (ie they fill in sector size, heads, sectors per track and density)
5	Hard discs support poll change (ie the poll change call works for hard discs and returns a sensible value; also locking them gives a sensible result)
6	Floppy discs support power-eject
7	Hard discs support power-eject

RISC OS 2 only uses bits 0 - 3; it ignores other bits.

FileCore_Create starts a new instantiation of FileCore, and, on return to your module, R0 points to the workspace that has been reserved for that new instantiation of FileCore. You must store this pointer in your module's workspace for future calls to FileCore; it is this value that tells FileCore which filing system you are (as well as enabling it to find its workspace!).

Unlike filing systems that are added under FileSwitch, the boot text offset cannot be –1 to call a routine.

Temporary buffers

The table below shows areas which may be used for temporary buffers when bit 3 of the flag word is not set:

	Scratch space	Spare screen area	Wimp free pool	RMA heap	System heap	Applic-ation area	Directory cache
FSEntry_Func 8	3	3	3	3	3	7	7
FSEntry_Close	3	3	3	3	3	7	7
FSEntry_Args 7	3	3	3	3	3	7	7
AllocCompact	3	3	3	3	3	7	7
Compact	3	3	3	3	3	7	7
*Backup X X	7	7	3	3	3	7	3
*Backup X Y	3	3	3	3	3	7	7
*Backup X X q	7	7	3	3	3	3	3
*Backup X Y q	3	3	3	3	3	3	7
*Compact	3	3	3	3	3	7	7

where AllocCompact is the auto-compact triggered when allocating space for a file, and Compact is a normal auto-compact.

Selecting your filing system

Your filing system should provide a * Command to select itself, such as *ADFS or *Net. This must call OS_FSControl 14 to inform FileSwitch that the module has been selected, thus:

```
StarFilingSystemCommand
        STMFD   R13!, {R14}
        MOV     R0, #FSControl_SelectFS
        ADR     R1, FilingSystemName
        SWI     XOS_FSControl
        LDMFD   R13!, {R15}
```

For full details of OS_FSControl 14, see page 2-98.

Other * Commands

There are no other * Commands that your filing system must provide. For many FileCore-based systems the range it provides will be enough, and your module need add no more.

Implementing SWI calls

SWI calls in a FileCore module are usually implemented by simply:

- loading R8 with the pointer to the FileCore instance private word for your module
- calling the corresponding FileCore SWI.

For example, here is how a module might implement a DiscOp SWI:

```
        STMFD   R13!, {R8, R14}     ; R12 points to module workspace
        LDR     R8, [R12, #offset]  ; R8 <- pointer to FileCore private word
        SWI     XFileCore_DiscOp
        LDMFD   R13!, {R8, R15}
```

Usually DiscOp, Drives, FreeSpace and DescribeDisc are implemented like this. Of course you can add any extra SWI calls that are necessary.

Removing your filing system

The finalise entry of your module must remove its instantiation of FileCore. For full details of how to do so, see the section entitled *Finalisation Code* on page 1-212.

Returning errors

Your module has to return errors through FileCore as follows:

The V flag must be set, and R0 is used to indicate the error:

- If bit 30 of R0 is set then, after clearing bit 30 of R0, it is a pointer to an error block.
- If bit 31 of R0 is set and bit 30 is clear, then R0 is a disc error:
  bits 0 - 20 are the disc byte address / 256
  bits 21 - 23 are the drive number
  bits 24 - 29 are the disc error number
- Else bits 30 - 31 are clear, and R0 is an error number:
  bits 0 - 7 are an error number (see list below)
  bits 8 - 29 are clear

In the latter two cases FileCore will generate a suitable error block.

The error numbers that may be returned are:

Error	Token	Default text	
11	ExtEscape	Escape	
94	Defect	Can’t map defect out	
95	TooManyDefects	Too many defects	
96	CantDelCsd	Can’t delete current directory	
97	CantDelLib	Can’t delete library	
98	CompactReq	Compaction required	
99	MapFull	Free space map full	
9A	BadDisc	Disc not formatted	(*not ADFS format*)
9B	TooManyDiscs	Too many discs	
9D	BadUp	Illegal use of ^	
9E	AmbigDisc	Ambiguous disc name	
9F	NotRefDisc	Not same disc	
A0	InUse	FileCore in use	
A1	BadParms	Bad parameters	
A2	CantDelUrd	Can’t delete user root directory	
A5	Buffer	No room for buffer	
A6	Workspace	FileCore Workspace corrupt	
A7	MultipleClose	Multiple file closing errors	
A8	BrokenDir	Broken directory	
A9	BadFsMap	Bad free space map	
AA	OneBadFsMap	One copy of map corrupt (use *CheckMap)	
AB	BadDefectList	Bad defect list	
AC	BadDrive	Bad drive	
AD	Size	Sizes don’t match	(*backups*)
AF	DestDefects	Destination disc has defects	(*backups*)

B0	BadRename	Bad RENAME
B3	DirFull	Directory full
B4	DirNotEmpty	Directory not empty
BD	Access	Access violation
C0	TooManyOpen	Too many open files
C2	Open	File open
C3	Locked	Locked
C4	Exists	Already exists
C5	Types	Types don’t match
C6	DiscFull	Disc full
C7	Disc	Disc error
C9	WriteProt	Protected disc
CA	DataLost	Data lost
CC	BadName	Bad name
CF	BadAtt	Bad attribute
D3	DriveEmpty	Drive empty
D4	DiscNotFound	Disc not found
D5	DiscNotPresent	Disc not present
D6	NotFound	Not found
D7	DiscNotFileCore	FileCore does not understand this disc
D8	NotToAnImageYouDont	Operation inapplicable to disc images
DE	Channel	Channel
FD	WildCards	Wild cards
FE	BadCom	Bad command

Module interfaces

The next section describes the interfaces to FileCore that your module must provide.

Module interfaces

Your module must provide two interfaces to FileCore: one for DiscOps, and one for other miscellaneous functions.

DiscOp entry

The entry for DiscOps does much of the work for a DiscOp SWI. It is passed the same values as FileCore_DiscOp (see page 2-223), **except**:

- an extra reason code is added to R1 allow background processing
- consequently R1 is no longer used to point to an alternative disc record instead R5 **always** points to a disc record
- R6 points to a boot block (for hard disc operations only), with the special value &80000000 indicating that none is available.

These are the reason codes that may be passed in 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		
15	Hard disc: specify	R2	

The reason codes you **must** support are 0, 1, 2, 5 and 6. You must complete the entire operation requested, or give an error if you are unable to do so.

Your routine must preserve R1 - R13 inclusive, except where noted otherwise above, ie:

- R2 must be incremented by the amount transferred for Ops 0, 1 and 2
- R3 must be incremented appropriately for Ops 1 and 2
- R4 must be decremented by the amount transferred for Ops 0, 1 and 2

You must also preserve the N, Z and C flags.

Returning errors

If there is no error then R0 must be zero on exit and the V flag clear. If there is an error then V must be set and R0 must be one of the following:

Value	Meaning
R0 < &100	internal FileCore error number
$\&100 \leq R0 < 2^{31}$	pointer to error block
$R0 \geq 2^{31}$	disc error bits: bits 0 - 20 = disc byte address / 256 bits 21 - 23 = drive bits 24 - 29 = disc error number bit 30 = 0

For a list of internal FileCore error numbers, see the section entitled *Disc errors* on page 2-278.

Background transfer

If bit 8 of R1 is set, then transfer may be wholly or partially in the background. This is an optional extension to improve performance. To reduce rotational latency the protocol also provides for transfers of indeterminate length.

R3 points to a list of address/length word pairs, specifying an exact number of sectors. The length given in R4 is treated as the length of the foreground part of the transfer. R5 is a pointer to the disc record.

Your module should return to the caller when the foreground part is complete, leaving a background process scheduled by interrupts from the controller. This process should terminate when it finds an address/length pair with a zero length field.

The foreground process can add pairs to the list at any time. To get the maximum decoupling between the processes your module should update the list after each sector. This updating **must** be atomic (use the `STMIA` instruction). Your module must be able to retry in the background.

The list is extended as below:

Offset	Contents
–8	Process error
–4	Process status
0	1st address
4	1st length
8	2nd address
12	2nd length
16	3rd address
20	3rd length
etc	
n	Loop back marker –*n* (where *n* is a multiple of 8)
n+4	Length of zero

Process error is set by the caller to 0; on an error your module should set this to describe the error in the format described above.

The bits in process status are:

Bit	Meaning when set
31	process active
30	process can be extended
0 - 29	pointer to block giving position of any error

Bits 31 and 30 are set by the caller and cleared by your module. Your module must have IRQs disabled from updating the final pair in the list to clearing the active bit.

A negative address of –*n* indicates that your module has reached the end of the table, and should get the next address/length pair from the start of the scatter list *n* bytes earlier.

Your module may be called with the scatter pointer (R3) not pointing to the first (address/length) pair. So, to find the addresses of Process error and Process status, you must search for the end of list. From this you may then calculate the start of the scatter block.

MiscOp entry

The entry for MiscOps does much of the work for a MiscOp SWI. It is passed the same values as FileCore_MiscOp (see page 2-240) – save for one reason code, noted below, which can be passed extra parameters.

- Although FileCore_MiscOp is not available in RISC OS 2, you must still provide this entry point, as other SWIs also use it. (The MiscOp SWI merely provides a convenient way of directly calling this entry point.)

These are the reason codes that may be passed in R0:

Value	Meaning
0	Mount
1	Poll changed
2	Lock drive
3	Unlock drive
4	Poll period
5	Eject disc

The reason codes you **must** support are 0, 2 and 3; for floppy drives, you **must** also support reason codes 1 and 4.

Your routine must preserve registers, and the N, Z and C flags – except where specifically stated otherwise.

You may only return an error from reason code 0 (Mount). This must be done in the same way as for the DiscOp entry; see the section entitled *Returning errors* on page 2-603.

For drives with disc sensing, reason code 1 (Poll changed) must always return *changed* in the spun-down state. If the drive is spun-up, you must return *maybe changed* if the drive has been permanently spun-up since the last 'Poll changed'; other wise you must return *changed:*

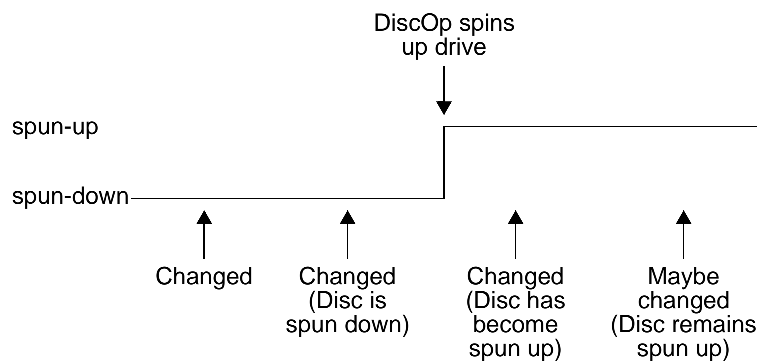


Figure 45.1 'Poll changed' returns for drives with disc sensing

Under RISC OS 2, the values returned from MiscOp 1 (Poll changed) in bits 4, 5, and 8 - 10 of R3 are ignored by FileCore.

Reason codes 2 and 3 (Lock/Unlock drive) must always perform that action. You must not try to track the state of the drive locking; FileCore does this for you.

Reason code 5 (Eject disc) will never be called if bits 6 and 7 of the descriptor block are clear, since this indicates that no drives support power-ejection. Otherwise it may get called in a variety of situations: for example, after dismounting all discs as part of shutting down all filing systems.

Reason code 5 is also called whenever FileCore issues an UpCall 1 (medium not present), or an UpCall 2 (medium not known). In this case, the top bit of the drive number is set, indicating that a disc should be ejected from the drive considered to be most appropriate. The values passed to the UpCall in R4 (the iteration count) and in R5 (the minimum timeout period) are also passed on in the same registers to the MiscOp entry point. The filing system may treat these as appropriate; for example, it may choose to eject only on iteration 0 for an auto-insert detect drive, as doing further ejects may make it hard to get a new disc into the drive.

For more details of OS_UpCall 1 and 2, see page 1-183.

46 Writing a device driver

Adding your own device driver to DeviceFS

DeviceFS does not know how to communicate directly with the hardware that your device driver uses. Your module must provide these facilities, and declare the entry points to DeviceFS.

This section describes how to add a device driver to DeviceFS. You should also see the chapter entitled *Modules* on page 1-201 for more information on how to write a module.

Registering your device driver

When your module initialises, it must register itself and its devices with DeviceFS. You must call DeviceFS_Register (see page 2-435) to register your device driver and any associated devices. Note that modules can hold more than one driver; in such cases you must call DeviceFS_Register for each one.

When you register your device driver with DeviceFS you pass it the location of an entry point to your driver's low level routines that interface with the hardware. A reason code is used to determine which of your driver's routines has been called.

- Reason codes with bit 31 clear are reserved for use by Acorn.
- Reason codes with bit 31 set are reserved for specific drivers. You do not need to register these with Acorn, although we suggest that you maintain some consistency between devices.

Registering and deregistering additional devices

You may later register additional devices by calling DeviceFS_RegisterObject (see page 2-439). This is most commonly needed for devices on a network.

You may deregister devices by calling DeviceFS_DeregisterObject (see page 2-440).

Deregistering your device driver

The finalise entry of your module must deregister all registered drivers and devices by calling DeviceFS_Deregister (see page 2-438). It must make this call for each device driver it registered.

Device driver interfaces

Calling conventions

The principal part of a device driver is the set of low-level routines that control the device's hardware. There are certain conventions that apply to them.

Private word

R8 on entry to the device driver is set to the value of R3 it passed to DeviceFS when registering by calling DeviceFS_Register. Conventionally, this is used as a private word to indicate which hardware platform is being used.

Workspace

R12 on entry to the device driver is set to the value of R4 it passed to DeviceFS when registering by calling DeviceFS_Register. Conventionally, this is used as a pointer to its workspace.

Returning errors

If a routine wishes to return an error, it should return to DeviceFS with V set and R0 pointing to a standard format error block.

Other conventions

Device driver routines must preserve R0, R1, and all other undocumented registers.

Interfaces

These are the interfaces that your device driver must provide. The entry point must be declared to DeviceFS by calling DeviceFS_Register when your device driver module is initialised.

DeviceDriver_Entry

Various calls are made by DeviceFS through this entry point when files are being opened and closed, streams halted etc. The actions are specified by R0 as follows:

DeviceDriver_Entry 0

Initialise

On entry

R0 = 0
R2 = DeviceFS stream handle
R3 = flags for opening the stream:
    bit 0 clear ⇒ stream opened for RX, set ⇒ stream opened for TX
    all others bits reserved, and should be ignored
R6 = pointer to special field control block

On exit

R2 = device driver stream handle

Details

This entry point is called as a stream is being opened onto the device driver by DeviceFS. The stream handle passed in must be stored, as you need to quote it when calling DeviceFS SWIs such as DeviceFS_Threshold, DeviceFS_ReceivedCharacter, and DeviceFS_TransmitCharacter.

The stream handle returned will be passed by DeviceFS when calling the device driver's other entry routines. It must **not** be zero, which is a reserved value.

The device driver is also passed a pointer to the special field string: see the section entitled *Special fields* on page 2-430.

You can be assumed that the special field block will remain intact until the stream has been closed.

DeviceDriver_Entry 1

Finalise

On entry

R0 = 1
R2 = device driver stream handle, or 0 for all streams

On exit

—

Details

This entry point is called when a stream is being closed. Your device driver must tidy up and ensure that all vectors have been released. This entry point is also called when a device driver is being removed, although in this case R2 is set to contain 0 indicating that all streams should be closed.

DeviceDriver_Entry 2

Wake up for TX

On entry

R0 = 2
R2 = device driver stream handle

On exit

R0 = 0 if the device driver wishes to remain dormant, else preserved

Details

This entry point is called when data is ready to be transmitted. Your device driver should set R0 to 0 if it wishes to remain dormant, or else start passing data to the physical device, calling DeviceFS_TransmitCharacter to obtain the data to be transmitted.

DeviceDriver_Entry 3

Wake up for RX

On entry

R0 = 3
R2 = device driver stream handle

On exit

—

Details

This entry point is called when data is being requested from the device driver. It is really issued to wake up any dormant device drivers, although you will always receive it when data is going to be read.

The device driver should return any data it receives from the physical device by calling DeviceFS_ReceivedCharacter. This will unblock any task waiting on data being returned.

This call is not applicable to all device drivers; most interrupt-driven buffered device drivers would be ready to receive data at any time.

DeviceDriver_Entry 4

Sleep RX

On entry

R0 = 4
R2 = device driver stream handle

On exit

—

Details

This entry point is called when data is no longer being requested from the device driver. If appropriate, the device driver can then wait to be woken up again using the 'Wake up for RX' entry point.

This call is not applicable to all device drivers; most interrupt-driven buffered device drivers would continue to receive and buffer data even after this call.

DeviceDriver_Entry 5

EnumDir

On entry

R0 = 5
R2 = pointer to path being enumerated

On exit

—

Details

This entry point is called as a broadcast to all device drivers when the directory structure for DeviceFS is about to be read. This allows them to add and remove non-permanent devices (such as net connections) as required.

The path supplied will be full (eg $.foo.poo) and null terminated.

DeviceDriver_Entry 6 and 7

Create buffer for TX (6), and Create buffer for RX (7)

On entry

R0 = 6 or 7
R2 = device driver stream handle
R3 = suggested flags for buffer being created
R4 = suggested size for buffer
R5 = suggested buffer handle (–1 for unique generated one)
R6 = suggested threshold for buffer

On exit

R3 - R6 modified as the device driver requires

Details

This entry point is called just before the buffer for a stream is going to be created; it allows the device driver to modify the parameters as required.

- R3 contains the buffer flags as specified when the device was registered; see the chapter entitled *The Buffer Manager* on page 4-85.
- R4 contains the suggested buffer size; this should be non-zero.
- R5 contains a suggested buffer handle. This is by default set to –1, which indicates that the buffer manager must attempt to generate a free handle.

  If you specify the handle of an existing buffer, then it will be used and not removed when finished with. For compatibility, the kernel devices use this feature to link up to buffers 1,2 or 3.
- R6 contains the threshold at which a halt event is received. This usually only applies to receive streams which want to halt the receive process, although it can be supplied on either. You may change this value by calling DeviceFS_Threshold.

DeviceDriver_Entry 8

Halt

On entry

R0 = 8
R2 = device driver stream handle

On exit

—

Details

This entry point is called when the free space has dropped below the specified threshold (set on creation, or by DeviceFS_Threshold). It is called so the device driver can – if necessary – try to stop its device from receiving more data (eg a serial device driver might perform handshaking by sending an XOff character, or asserting the RTS line) until the Resume entry point is called.

DeviceDriver_Entry 9

Resume

On entry

R0 = 9
R2 = device driver stream handle

On exit

—

Details

This entry point is called when the free space has risen above the specified threshold (set on creation, or by DeviceFS_Threshold). It is called so the device driver can – if necessary – try to resume its device receiving more data (eg a serial device driver might perform handshaking by sending an XOn character, or de-asserting the RTS line) until the Halt entry point is again called.

DeviceDriver_Entry 10

End of data

On entry

R0 = 10
R2 = device driver stream handle
R3 = –1

On exit

R3 = 0 if more data coming eventually, else –1 (ie no more data coming)

Details

This entry point is called as a result of FileSwitch calling DeviceFS to check on EOF, and DeviceFS believing that there is no more data to come. In more detail:

DeviceFS informs FileSwitch that more data is coming eventually – without calling this entry point – if:

- the stream is buffered, and its buffer still holds data
- the stream is unbuffered, and its RX/TX word is not empty

Otherwise it calls this entry point. In most cases a device driver should ignore this, and return with all registers preserved (so R3 = –1, thus there is no more data coming). In some cases, such as a scanner, you may be able to give an accurate return.

DeviceDriver_Entry 11

Stream created

On entry

R0 = 11
R2 = device driver stream handle
R3 = buffer handle (–1 if none)

On exit

—

Details

This entry point is called after a stream has finally been generated. Your device driver can then perform any important interrupt handling, set itself up and start receiving or transmitting.

Part 6 – Networking

47 Econet

Introduction

The Econet module provides the software needed to use Acorn's own Econet networking system. The software allows you to send and receive data over the network.

It is used by RISC OS modules such as NetFS and NetPrint, which provide network filing and printing facilities respectively. It is also used by various other Acorn products that use Econet, such as FileStores, Econet bridges, and so on.

Note that to use the Econet you must have an Econet expansion module fitted to your RISC OS computer. If you do not already have one, they are available from your Acorn supplier.

Overview

Econet is Acorn's own networking system, and the Econet module provides the necessary software to use it.

The main purpose of any networking system is to transfer data from one machine to another. Econet breaks up the data it sends into small parts which are sent using a well defined protocol.

Econet does not use buffers in the same way as most other input and output facilities that RISC OS provides. Instead the data is moved directly between the Econet hardware and memory. This means that each time data is transmitted or received, there has to be a block of memory available for the Econet software to use immediately, either to read data from or place data in.

These blocks of memory are administered by the Econet software, which uses control blocks to do so. Many of the SWIs interact with these control blocks, so you can set them up, read the status of an Econet transmission or reception, and release the control blocks memory when you have finished using them.

In the same way as files under the filing system use file handles, these control blocks also use handles. Just like file handles, your software must keep a record of them while you need to use them.

The Econet also provides a range of immediate operations, which allow you to exercise some control over the hardware of remote machines, assuming you get their co-operation. Some of these will work across the entire range of Acorn computers, whereas others are more hardware-dependent and so may only be possible on RISC OS machines.

Technical Details

Packets and frames

A single transmission of data on an Econet is called a *packet*. Packets travel across the network from the transmitting station to the receiving station. The most common form of packet is called a ‘four way handshake’. A ‘four way handshake’ consists of **four** frames. Each of these four frames starts with the following four bytes:

- the station number of the destination station
- the net number of the destination station
- the station number of the source station
- the net number of the source station.

These four bytes are sent in this order to facilitate decoding by the software in the receiving station.

The first frame is sent by the transmitting station; it contains the usual first four bytes, the port byte (described later), and the flag byte (also described later). This first frame is called the *scout*. The receiving station then replies with the *scout acknowledge*, which consists of just the usual first four bytes. The third frame is the *data* frame; this frame has the usual first four bytes, followed by all the data to be transferred. Lastly there is a *final acknowledge* frame which is identical to the scout acknowledge frame.

This exchange of frames can be seen with the NetMonitor and is displayed something like this.

```
FE0012008099 1200FE00 FE00120048454C500D 1200FE00
```

- the transmitting station is &12 (18 in decimal)
- the receiving station is &FE (254 in decimal)
- both stations are on net zero
- the flag byte is &80
- the port byte is &99
- the data that is transmitted is &48, &45, &4C, &50, &0D.

Receiving data and using RxCBs

Successful transmission of data requires co-operation from the receiving station. A station shows that it is ready to receive by setting up a *receive control block* (or *RxCB*). All RxCBs are kept by the Econet software and don't need to concern you. To create an RxCB all you need to do is call a single SWI (Econet_CreateReceive: see page 2-657), telling the Econet software all the required information. The Econet software will return to you a handle which you then use to refer to this particular RxCB in any further dealings with the Econet software.

The information required by the Econet software is:

- which station(s) to accept data from
- which port number(s) to accept data on
- where to put the data when it arrives.

It is important you note that when the data arrives from the transmitting station it is not buffered at all – it is taken directly from the hardware and placed in memory at the address you specify. This area of memory is referred to as a buffer (in this case a *receive buffer*). A consequence of this is that memory used for receiving Econet packets **must** be available at all times whilst the relevant RxCB is open. You **must not** use memory in application space if your program is to run within the Desktop environment.

The Econet software keeps a list of all the open RxCBs. When a scout frame comes in it is checked to see if it matches any of the currently open RxCBs:

- if it doesn't then the receiving software indicates this to the transmitting software by not sending a scout acknowledge frame
- if it does then the receiving software sends out a scout acknowledge, and then copies the data frame into the corresponding buffer
- if the data frame overruns the buffer then the receiving software does not send the final acknowledge frame.

Status of RxCB's

All RxCBs have a status value. These values are tabulated below.

7	Status_RxReady
8	Status_Receiving
9	Status_Received

The status of a particular RxCB can be read using the Econet_ExamineReceive call (page 2-659); this takes the receive handle of an RxCB and returns its status.

When an RxCB has been received into, its status will change from RxReady to Received; usually, you will then call Econet_ReadReceive (page 2-661). This returns information about the reception; most importantly it tells you how much data was received – which can be anything from zero to the size of the buffer. It also returns the value of the flag byte.

The port, station, and net are also returned; these are useful because you can open an RxCB that allows reception on any port or from any station.

Abandoning RxCB's

It is very important that when RxCBs are no longer required, either because they have been received into, or because they have not been received into within a certain time, that they are removed from the system. You do so by calling the SWI Econet_AbandonReceive (page 2-663). The major function of this call is to return to the RMA the memory that the Econet software used to hold the RxCB; obviously if RxCBs are not abandoned, they will consume memory which will not automatically be recovered by the system.

Receiving data using a single SWI

The usual sequence of operations required for software to receive data is as follows: First call SWI Econet_CreateReceive, then make numerous calls to SWI Econet_ExamineReceive until either a reception occurs, a time out occurs, or the user interferes (by pressing *Escape* for instance). Then read the RxCB (Econet_ReadReceive) if it has been received into. Finally, abandon the RxCB (Econet_AbandonReceive).

To make this task easier the Econet software provides a single SWI (Econet_WaitForReception: see page 2-664) which does the polling, the reading, and the abandoning for you. To call SWI Econet_WaitForReception you must pass in:

- the receive handle
- the amount of time you are prepared to wait
- a flag which indicates whether you wish the call to return if the user presses the Escape key.

Econet_WaitForReception returns one of four status values:

8	Status_Receiving
9	Status_Received
10	Status_NoReply
11	Status_Escape

The call will return as soon as a reception occurs; when this happens the status is *Received*. If the time limit expires then the status is usually *NoReply*, but if reception had started just after the timeout, and so was then abandoned, the status will be *Receiving*. This is not a very likely case. If the escapable flag is set then pressing the Escape key causes the call to return with the *Escape* status.

Transmitting data and using TxCB's

Transmission is roughly similar to reception; a single SWI (Econet_StartTransmit – page 2-667) is all that is required to get things started. This call requires the following information:

- the destination station (and net)
- the port number to transmit on
- the flag byte to send
- the address and length of the data to send.

SWI Econet_StartTransmit returns a handle. These handles are distinct from the handles used by the receive SWIs.

Various transport types may impose a limit on the amount of data you can send in a single packet. You can find out the limit for the transport you are using by calling Econet_PacketSize (page 2-700).

Status of TxCB's

To check the progress of your transmission you can call Econet_PollTransmit (page 2-669). This returns the status of the particular TxCB, which will be one of seven possible values:

0	Status_Transmitted
1	Status_LineJammed
2	Status_NetError
3	Status_NotListening
4	Status_NoClock
5	Status_TxReady
6	Status_Transmitting

Status_Transmitted means that your transmission has completed OK and that the data has been received by the destination machine. *Status_TxReady* means that your transmission is waiting to start, either because the Econet is busy receiving or transmitting something else, or your transmission is queued (see later for more details of this). *Status_Transmitting* is obvious; so too is *Status_NoClock*, which means that the Econet is not being clocked, or more likely your station is not plugged into the Econet. *Status_LineJammed* means that the Econet software was unable to gain access to the

Econet; this may be because other stations were transmitting, but it is more likely that there is a fault in the Econet cabling somewhere. *Status_NotListening* is returned when the destination station doesn't send back a scout acknowledge frame; this is usually because the destination station doesn't have a suitable open receive block. *Status_NetError* will be returned if some part of the four way handshake is missing or damaged; the usual cause of this status is the sender sending more data than the receiver has buffer space for, so the receiver doesn't send back the final acknowledge frame.

Retrying transmissions

Status returns like *NotListening* and *NetError* can also be caused by transient problems with the Econet such as electrical noise, or by the receiving station using its floppy disc or being otherwise too busy to accept data. Because of this it is usual to try more than once to send a packet if these status returns occur. To make this easier for you the Econet software can automatically perform these extra attempts for you. These retries are controlled by passing two further values in to the Econet_StartTransmit SWI:

- the number of times to try, referred to as the Count
- the amount of time to wait between tries, referred to as the Delay.

If the Count is either zero or one then only one attempt to transmit will take place. If the Count is two or more then retries will occur, at the specified interval (given in centiseconds). To give an example as it would be written in BASIC V:

```
10 DIM Buf% 20
20 Port%=99: Station%=7: Net%=0
50 SYS "Econet_StartTransmit",0,Port%,Station%,Net%,Buf%,20,3,100 TO Tx%
60 END
```

When this partial program was `RUN` it would try to transmit immediately, probably before the program reached the `END` statement. If this transmission failed with either *Status_NotListening* or *Status_NetError*, then the Econet software would wait for one second (100 centiseconds) and try again. If this also failed then the software would wait a further second and try for a third time. The status of the final (in this case third) transmission would be the status finally stored in the TxCB; this could be read using SWI Econet_PollTransmit. To see this we could add some extra lines to the example program:

```
 30 TxReady%=5
 40 Transmitting%=6
 60 REPEAT
 70   SYS "Econet_PollTransmit", Tx% TO Status%
 80   PRINT Status%
 90 UNTIL NOT ((Status%=TxReady%) OR (Status%=Transmitting%))
100 END
```

Now the program will show us the status of the TxCB. We would be very unlikely to see the status value ever be *Status_Transmitting* since it will only have this value for about 90µs during the two seconds it is retrying for. But it is most important that your software should be able to handle such a situation without error.

For retries to be effective you must try for at least 5 seconds. Recommended values for the Count and Delay are:

Broadcasts:	Count = 5, Delay = 5
Machine peeks:	Count = 40, Delay = 5
All other transmissions:	Delay × (Count – 1) ≥ 500

Abandoning TxCB's

As with receptions it is most important that memory used for transmitting Econet packets **must** be available at all times whilst the relevant TxCB is open. You **must not** use memory in application space if your program is to run within the Desktop environment. This is because like receptions, transmissions move data directly from memory at the address you specify to the hardware. Also, as with receptions, it is important to inform the Econet software that you have finished with your transmission and that memory required for the internal TxCB may be returned to the RMA. You do this by calling Econet_AbandonTransmit (page 2-671) with the appropriate TxHandle:

```
100 SYS "Econet_AbandonTransmit", Tx% TO FinalStatus%
110 PRINT "The final status was ";FinalStatus%
```

Transmitting data using a single SWI

To make this start, poll, and abandon sequence easier for you the Econet software provides it all as a single call (Econet_DoTransmit: see page 2-672). This call has the same inputs as SWI Econet_StartTransmit, but instead of returning a handle it returns the final status. Using this call our program would look like this:

```
10 DIM Buf% 20
20 Port%=99: Station%=7: Net%=0
40 SYS"Econet_DoTransmit",0,Port%,Station%,Net%,Buf%,20,6,100 TO Status%
50 PRINT "The final status was ";Status%
```

Converting a status to an error

As you can see this makes things a lot easier. As an aid to presenting these status values to the user there are two SWI calls to convert status values to a textual form, the most frequently used of which is the call Econet_ConvertStatusToError (page 2-677). This call takes the status and returns an error with the appropriate error number and an appropriate string describing the error.

For instance we could add an extra line to our final program:

```
80 SYS "Econet_ConvertStatusToError", Status%
```

Note that the SYS command sets unused registers to zero.

Copying the error to RAM

Our program will now RUN and always have an error, in this case the error ‘Network station not listening at line 80’. This error message is actually held in the RMA, in one of a number of error blocks used by the MessageTrans module, and so you cannot directly add to it. Furthermore, the error message will have a ‘limited lifetime’ before MessageTrans reuses the error block. Consequently, if you wish to process the error message or to preserve it, you should copy it into a buffer. To do so you can specify the location and size of such a buffer when calling Econet_ConvertStatusToError:

```
70 DIM Error% 50
80 SYS "Econet_ConvertStatusToError", Status%, Error%, 50
```

This new program will function in the same manner as the previous program except that the error block will have been copied from the Econet messages file (in the ROM) into RAM (at the address given in R1). The main reason for this is to allow the Econet software to customise the error for you.

Adding station and net numbers

If the station and net numbers are added as inputs to the call, the Econet software will add them to the output string:

```
80 SYS "Econet_ConvertStatusToError",Status%,Error%,50,Station%,Net%
```

Now the error reported will be of the form ‘Network station 7 not listening at line 80’. It is important to stress that this is a general purpose conversion. It will convert Status_Transmitted just as well as Status_NotListening, so usually you would test the returned status from Econet_DoTransmit, and only convert status values other than Status_Transmitted into errors:

```
30 Transmitted%=0
60 IF Status%=Transmitted% THEN PRINT "OK": END
```

The same program fragment could be written in assembler (this example, like all others in this chapter, uses the ARM assembler rather than the assembler included with BBC BASIC V – there are subtle syntax differences):

```
Tx      MOV     r0, #0                  ; Flag
        MOV     r1, #99                 ; Port
        MOV     r2, #7                  ; Station
        MOV     r3, #0                  ; Net
        ADR     r4, Buffer
        MOV     r5, #20                 ; Buffer length
        MOV     r6, #6                  ; Count
        MOV     r7, #100                ; Delay (in centiseconds)
        SWI     Econet_DoTransmit
        BEQ     r0, #Status_Transmitted
        LDRNE   r1, ErrorBuffer
        MOVNE   r2, #50
        SWINE   Econet_ConvertStatusToError
        MOV     pc, lr
```

Notice here in the assembler version how the return values from Econet_DoTransmit fall naturally into the input values required for Econet_ConvertStatusToError. This code fragment is not really satisfactory since no code written as either a module or a transient command should ever call the non-X form of SWIs. If the routine Tx is treated as a subroutine then it should look more like this:

```
Tx      STMFD   sp!, {lr}
        MOV     r0, #0                  ; Flag
        MOV     r1, #99                 ; Port
        MOV     r2, #7                  ; Station
        MOV     r3, #0                  ; Net
        ADR     r4, Buffer
        MOV     r5, #20                 ; Buffer length
        MOV     r6, #6                  ; Count
        MOV     r7, #100                ; Delay (in centiseconds)
        SWI     XEconet_DoTransmit
        BVS     TxExit
        TEQ     r0, #Status_Transmitted
        ADRNE   r1, ErrorBuffer
        MOVNE   r2, #50
        SWINE   XEconet_ConvertStatusToError
TxExit  LDMFD   sp!, {pc}
```

This routine returns with V clear if all went well; if V is set, then on return R0 will contain the address of a standard error block.

RISC OS 2

RISC OS 2 differs from later versions in that it doesn't use the MessageTrans module, but instead has the full text of the English error messages in ROM. When converting messages with added station numbers you must convert into your own buffer. If you give no buffer, or its length is insufficient, then the station and net numbers are ignored and RISC OS 2 returns a pointer to the normal ROM copy of the message.

Converting a status to a string

The second error conversion call is Econet_ConvertStatusToString (page 2-675), which does exactly what its name suggests. The input requirements are very similar to the string conversion SWIs supported by RISC OS. In this case you pass the status value, a buffer address, and the length of the buffer. As with Econet_ConvertStatusToError you can also pass the station and net numbers, which will be included in the output string. To illustrate this the assembler routine shown above is changed to print the status on the screen:

```
Tx      STMFD   sp!, {lr}
        MOV     r0, #0                  ; Flag
        MOV     r1, #99                 ; Port
        MOV     r2, #7                  ; Station
        MOV     r3, #0                  ; Net
        ADR     r4, Buffer
        MOV     r5, #20                 ; Buffer length
        MOV     r6, #6                  ; Count
        MOV     r7, #100                ; Delay (in centiseconds)
        SWI     XEconet_DoTransmit
        BVS     TxExit
        TEQ     r0, #Status_Transmitted
        BEQ     TxExit                  ; Everything is OK
        ADR     r1, TextBuffer
        MOV     r2, #50                 ; Text buffer length
        MOV     r5, r0                  ; Save the status value
        SWI     XOS_ConvertCardinal1    ; Convert the status number
        MOVVC   r0, r5                  ; Recall status if no error
        SWIVC   XEconet_ConvertStatusToString
        ADRVC   r0, TextBuffer
        SWIVC   XOS_Write0              ; Print the resultant string
TxExit  LDMFD   sp!, {pc}
```

MessageTrans tokens

Both Econet_ConvertStatusToError and Econet_ConvertErrorToString use MessageTrans to produce the error message or string. The message tokens for each of the status values are tabulated below. Where two tokens are listed, as for Status_NotListening, the first is for the error message – or string – without a station number inserted, and the second is for the version with the station number inserted. The files supplied with RISC OS that the Econet software uses are ‘Resources:$.Resources.Econet.Messages’ and ‘Resources:$.Resources.Global.Messages’ (used solely for the status message for Escape).

Error	Status	Token(s)
0	Status_Transmitted	TxOK
1	Status_LineJammed	LineJam
2	Status_NetError	NetErr

3	Status_NotListening	NotLstn	StnNLsn
4	Status_NoClock	NoClk	
5	Status_TxReady	TxReady	
6	Status_Transmitting	Txing	
7	Status_RxReady	RxReady	
8	Status_Receiving	Rxing	
9	Status_Received	Rxd	
10	Status_NoReply	NoReply	StnNRpy
11	Status_Escape	Escape	
12	Status_NotPresent	NotPres	StnNPrs

Flag bytes

The flag byte is sent from the transmitting station to the receiving station and can be treated as an extra seven bits of data. By convention, it is used as a simple way of distinguishing different types of packet sent to the same port, and it is worth you doing the same.

This is most useful in server type applications where it is often the case that similar data can be sent for different purposes, or some sorts of data are outside the normal scope. An example is a server that takes requests for teletext pages, but can also return the time. A different value for the flag byte allows the server to differentiate time requests from normal traffic. Another example is the printer server protocol, which uses the flag byte to indicate the packet that is the last in the print job, without having to change the data part of the packet.

Port bytes

The port byte is used in the receiving station to distinguish traffic destined for particular applications or services.

For instance the printer server protocol uses port &D1 for all its connect, data transfer, and termination traffic, whereas the file server uses port &99 for all its incoming commands. This use of separate ports for separate tasks is also exploited further by the file server protocol in that every single request for service by the user can use a different port for its reply. This prevents traffic getting confused.

The Econet software provides some support for you to use ports by providing an allocation service for port numbers. Port numbers should, if possible, be allocated for all incoming data.

Software that requires the use of fixed port numbers, like NetFS and NetPrint, can claim these fixed ports by calling Econet_ClaimPort (page 2-690). This call takes a port number as its only argument. When these claimed ports are no longer required (when the module dies for instance) it can be 'returned' by calling SWI Econet_ReleasePort (page 2-687).

Other software that would like a port number allocated to it can call Econet_AllocatePort (page 2-688), which will return a port number. While this port number is allocated no other calls to Econet_AllocatePort will return that number, until it is 'released' by calling Econet_DeAllocatePort (page 2-689) with the port number as an input. The NetFS software uses this method of allocation and deallocation to get ports to use as reply ports in the file server protocol. The Econet software keeps a table in which it records the state of each port number: this can be either free, claimed or allocated.

Freeing ports

Ports that have been claimed will not be allocated, and can only be freed by calling SWI Econet_ReleasePort. Calling SWI Econet_DeAllocatePort will return an error if the port is claimed rather than allocated. Ports that have been allocated can not be claimed, and in fact an attempt to claim an allocated port will return an error. You should be careful with software that uses allocated ports to make sure that all ports are deallocated when they are no longer required, especially after an error. The claiming and releasing of ports should likewise be carefully checked.

An example of use of the port allocator

A typical example of the use of the port allocator would be a multi-player adventure game server. The server would claim one port (eg port &1F). This port number would then be the only fixed port number in the entire protocol. When a player wished to join the game she should ask for a port to be allocated in her machine and send this port, along with all the information required to enter the game, to the game server on port &1F. If the server can't be contacted or doesn't reply within the required time the port should be deallocated and an error returned. When the server receives this packet it should check the user's entry data; if this is OK it should then allocate a port for that user and return it, along with any other information required to start the game off. When the user wants to quit the game the server should deallocate its user's port, then send the last reply to the user. The user should deallocate the port when the reply arrives or if the server doesn't reply soon enough.

To illustrate this example the user entry routine is shown below; note that this routine is coded for clarity rather than size or efficiency.

```
Entry   STMFD   sp!, {r0-r8,lr}         ; R0 points to the text string
        SWI     XEconet_AllocatePort
        BVS     Exit
```

```
        STRB    r0, Server_ReplyPort
        LDR     r1, Server_Station
        LDR     r2, Server_Net
        ADR     r3, Buffer
        MOV     r4, #?Buffer            ; Length of buffer
        SWI     XEconet_CreateReceive
        BVS     DeAllocateExit
        MOV     r8, r0                  ; Preserve the RxHandle

        LDR     r1, [sp, #0]            ; Address of text string to copy
        ADR     r4, Buffer              ; Get buffer to copy into
        MOV     r5, #0                  ; Index into Tx Buffer
        LDRB    r0, Server_ReplyPort
        STRB    r0, [r4, r5]            ; Send the port for the server
CopyLoop
        ADD     r5, r5, #1
        CMP     r5, #?Buffer            ; Have we run out of buffer?
        BHS     BufferOverflow
        LDRB    r0, [r1], #1            ; Pick up byte and move to next one
        CMP     r0, #" "                ; Is this a control character?
        MOVLT   r0, #CR                 ; Terminate as the server expects
        STRB    r0, [r4, r5]
        BGE     CopyLoop                ; Loop back for the next byte
        ADD     r5, r5, #1              ; Set entry conditions for Tx

        MOV     r0, #0
        MOV     r1, #EntryPort          ; A constant
        LDR     r2, Server_Station
        LDR     r3, Server_Net
        LDR     r6, Server_TxDelay
        LDR     r7, Server_TxCount
        SWI     XEconet_DoTransmit
        BVS     DeAllocateExit
        TEQ     r0, #Status_Transmitted
        BEQ     WaitForReply
ConvertEconetError
        ADR     r1, Buffer              ; Convert status and exit
        MOV     r2, #?Buffer
        SWI     XEconet_ConvertStatusToError
        B       DeAllocateExit

WaitForReply
        MOV     r0, r8                  ; Receive handle
        LDR     r1, Server_RxDelay
        MOV     r2, #0                  ; Don't allow ESCape
        SWI     XEconet_WaitForReception
        BVS     DeAllocateExit
        TEQ     r0, #Status_Received
        BNE     ConvertEconetError

        LDR     r0, Buffer              ; Get server return code
        CMP     r0, #0                  ; Has there been an error?
        ADR     r0, Buffer              ; Get address of reply
        BNE     DeAllocateExit          ; Yes, process error
        LDRB    r1, [r0, #4]            ; Load server's port
        STRB    r1, Server_CommandPort
```

```
Exit
        STRVS   r0, [sp, #0]            ; Poke error into return regs
        LDMFD   sp!, {r0-r8,pc}         ; Return to caller

BufferOverflowError
        DCD     ErrorNumber_BufferOverflow
        DCB     Command too long for buffer", 0
        ALIGN

BufferOverflow
        ADR     r0, BufferOverflowError
DeAllocateExit
        MOV     r1, r0                  ; Preserve the original error
        LDRB    r0, Server_ReplyPort
        SWI     XEconet_DeAllocatePort
        MOV     r0, r1                  ; Ignore deallocation errors
        CMP     pc, #&80000000          ; Set V
        B       Exit                    ; Exit through common point
```

Points to notice in the example are:

- the careful use of a single exit point
- the consistent return of errors (no matter what type)
- the opening of the receive block before doing the transmit
- the use of the ‘X’ form of SWIs.

It should be noted that the routine uses and manipulates global state as well as taking specific input and returning specific output.

Econet events

To allow Econet based programs to be kinder to other applications within the machine, it is possible for your program to be ‘notified’ when either a reception occurs or a transmission completes. This means that other applications can be using the time that your program would have spent polling, either inside Econet_DoTransmit or inside Econet_WaitForReception. This ‘notification’ is carried by an event. There are separate events for reception and for completion of transmission. These two events are:

14	Event_Econet_Rx
15	Event_Econet_Tx

On entry to the event vector:

- R0 will contain the event number, either Event_Econet_Rx or Event_Econet_Tx
- R1 will contain the receive or transmit handle as appropriate
- R2 will contain the status of the completed operation
- R3 will contain the port of the completed operation, except under RISC OS 2.

The status for receive will always be *Status_Received*, but for transmit it will indicate how the transmission completed:

0	Status_Transmitted
1	Status_LineJammed
2	Status_NetError
3	Status_NotListening
4	Status_NoClock
9	Status_Received

These events can be enabled and disabled in the normal way using OS_Byte calls.

Using events from the Wimp

If your program is a client of the Wimp then all your event routine need do is set the Wimp poll word non-zero when the event happens; see the section entitled *PollWord_NonZero 13* on page 3-122.

```
Event   TEQ     r0, #Event_Econet_Rx
        TEQNE   r0, #Event_Econet_Tx
        MOVNE   pc, lr                  ; If not, exit as fast as possible

        STMFD   sp!, {lr}               ; Must preserve all regs for others
        ADR     r14, WimpPollWord
        STR     pc, [r14]               ; Set flag with non-zero value
        LDMFD   sp!, {pc}               ; Return, without claiming vector
```

Setting up background tasks

Since the interfaces required for reception and transmission can be called from within event routines, you can set up background tasks that make full use of the facilities offered by Econet. Note that it is important to check that the handle offered in the event belongs to your program, since there may well be many programs using this facility. The example given below is of a simple background server for sending out the time. Not all of the code needed is shown, just the event routine:

```
Start   STMFD   sp!, {r0-r4,lr}
        MOV     r0, #EventV             ; The vector we want is EventV
        ADR     r1, Event               ; Where to goto when it happens
        MOV     r2, #0                  ; Required so that we can release
        SWI     XOS_Claim

        MOVVC   r0, #14                 ; Enable event
        MOV     r1, #Event_Econet_Rx
        SWIVC   XOS_Byte
        MOVVC   r0, #14                 ; Enable event
        MOV     r1, #Event_Econet_Tx
        SWIVC   XOS_Byte
```

```
        MOVVC   r0, #CommandPort        ; First open the reception
        MOV     r1, #0                  ; From any station
        MOV     r2, #0                  ; From any net
        ADR     r3, Buffer
        MOV     r4, #?Buffer
        SWIVC   XEconet_CreateReceive
        STRVC   r0, RxHandle
        STRVS   r0, [sp]
        LDMFD   sp!, {r0-r4,pc}

Event   TEQ     r0, #Event_Econet_Rx
        BNE     LookForTx
        LDR     r0, RxHandle            ; Get our global state
        TEQ     r0, r1                  ; Is it for us?
        MOVNE   r0, #Event_Econet_Rx
        MOVNE   pc, lr                  ; If not, exit as fast as possible
        STMFD   sp!, {r3-r7}            ; Only R0, R1 and R2 are free for use
        MOV     r0, r1                  ; Receive handle
        SWI     XEconet_ReadReceive     ; R4.R3 is the reply address
        BVS     Exit

        MOV     r6, r3                  ; Save the station number for later
        MOV     r0, #Module_Claim
        MOV     r3, #8 + 5              ; Two words and five bytes required
        SWI     XOS_Module              ; Memory MUST come from RMA
        BVS     Exit

        ADD     r1, r2, #8              ; Get the address of the 5 bytes
        MOV     r0, #3                  ; Set OS_Word reason code
        STRB    r0, [r1]                ; Read as a five byte time
        MOV     r0, #14                 ; Read from the real time clock
        SWI     XOS_Word
        BVS     Exit

        MOV     r0, #0                  ; Flag byte
        MOV     r3, r4                  ; Net number
        MOV     r4, r1                  ; Get the address of the 5 bytes
        LDRB    r1, [r5]                ; The reply port the client sent
        MOV     r2, r6                  ; Station number
        MOV     r5, #5                  ; Number of bytes to send
        MOV     r6, #ReplyCount
        MOV     r7, #ReplyDelay
        SWI     XEconet_StartTransmit
        BVS     Exit

        SUB     r4, r2, #8              ; R4 now in R2
        STR     r0, [r4, #4]            ; Save TxHandle in record
        ADR     r1, TxList              ; Address of the head of the list
        LDR     r2, [r1, #0]            ; Head of the list
        STR     r2, [r4, #0]            ; Add the list to new record
        STR     r4, [r1, #0]            ; Make this record the list head
```

```
        MOV     r0, #CommandPort        ; Now re-open the reception
        MOV     r1, #0                  ; From any station
        MOV     r2, #0                  ; From any net
        ADR     r3, Buffer
        MOV     r4, #?Buffer
        SWI     XEconet_CreateReceive
        STRVC   r0, RxHandle
Exit
        LDMFD   sp!, {r3-r7, pc}        ; Return claiming vector

LookForTx
        TEQ     r0, #Event_Econet_Tx
        MOVNE   pc, lr
        STMFD   sp!, {r3, lr}           ; Get two extra registers
        ADR     r3, TxList              ; The address of the head of list
        LDR     r14, [r3]               ; The first record in the list
        B       StartLooking
NextTx
        MOV     r3, r14                 ; Search the next list entry
        LDR     r14, [r3]               ; Get the link address
StartLooking
        CMP     r14, #0                 ; Is this the end of the list?
        MOVLE   r0, #Event_Econet_Tx    ; Restore entry conditions
        LDMLEFD sp!, {r3, pc}           ; Return, continuing to next owner
        LDR     r0, [r14, #4]           ; Get the handle for this record
        TEQ     r0, r1                  ; Is this event one of ours?
        BNE     NextTx                  ; No, try next record in list

        LDR     r2, [r14]               ; Get the remainder of the list
        STR     r2, [r3]                ; Remove this record from list
        MOV     r2, r14                 ; The record address for later
        SWI     XEconet_AbandonTransmit
        MOV     r0, #Module_Free
        SWI     XOS_Module              ; Return memory to RMA, ignore error
        LDMFD   sp!, {r3, lr, pc}       ; Return, claiming vector
```

This program also illustrates some of the more advanced features of Econet. In particular; it shows the ability to specify reception control blocks that can accept messages from more than one machine, or on more than one port. Receive control blocks like this are referred to as *wild*, as in *wild card matching* used in file name look up. Specifying either the station or net number (usually both) as zero means ‘match any’. The same is true of the port number, although this facility is much less useful! This wild facility does not mean that more than one packet can be received, but rather that more than one particular packet will be acceptable. Once a packet has been received, the RxCB has Status_Received and is no longer open.

It is worth noting an implementation detail here. Receive control blocks are kept by the Econet software in a list, when an incoming scout has been received the list is scanned to find the first RxCB that matches it. To ensure that things go as one would expect the Econet software that implements the SWI Econet_CreateReceive always adds wild RxCBs to the tail of the list, and normal RxCBs to the middle of the list (between the

normal and the wild ones). This ensures that when packets arrive they will be checked for exact matches before wild matches, and that if there is more than one acceptable RxCB then the one used will be the one that was opened first, ie first in first served.

Broadcast transmissions

As a complement to this concept of wild receive control blocks there are broadcast transmissions. A broadcast has both its destination station and net set to &FF, it can then be received by more than one machine. To achieve this it does not use the normal four way handshake, it is in fact a single packet. On the NetMonitor it would look something like this:

```
FFFF1200809F5052494E54200100
```

The broadcast address at the beginning (&FF, &FF), the source station and net (&12, &00), the control byte (&80), and the port (&9F) are the same as a normal scout frame, but then the data follows, in this case eight bytes.

Although the Econet software within RISC OS can transmit and receive broadcast messages of up to 1020 bytes (RISC OS 2) or 1024 bytes (later versions), other machines on Econet can't cope with messages of more than eight bytes without getting confused; this confusion causes them to corrupt such broadcasts. These other machines include things like FileStores and bridges, so beware! It is possible to transmit and/or receive zero to eight bytes without them being corrupted, but only broadcasts of exactly eight bytes can be received by BBC or Master computers, as well as being transported from net to net by bridges.

Transmitting a broadcast is exactly the same as transmitting a normal packet, all you need to do is set the destination station and net to &FF (**not** to –1).

Versions of RISC OS after 2.00 support a wider range of broadcasts, allowing local broadcasts (which are only seen on the local net) and long broadcasts (broadcasts of more than eight bytes, which new bridges will recognise and correctly propagate). To use these, set the station number to &FF, and the net number as follows:

Net	**Range**	**Size**
&FF	Global	Small (8 bytes maximum)
&FE	Global	Long (1020/1024 bytes maximum)
&FD	Local	Long (1020/1024 bytes maximum)
&FC	*reserved*	*reserved*

Note that local long broadcasts (ie net = &FD) are ignored by existing machines and bridges, and will always work.

Broadcasts don't return the status *Status_NotListening*, since there is no way for the transmitting station to determine whether or not its broadcast was received. Broadcasts are basically designed for locating resources, ie to transmit your desire to know about a

particular class of thing. Anything recognising the broadcast will reply, so you know what's what and where it is. NetFS uses broadcast to find file servers by name, and NetPrint uses broadcast to find printer servers. The example broadcast packet shown above contains the ASCII text 'PRINT ' and is, not surprisingly, a request for all printer servers to respond.

Local loopback

When transmissions take place, the destination address is checked to see if it is the local machine (ie a transmission to your own machine). If this is the case then no access to the Econet network will take place, and if a suitable receive control block exists the data is transferred directly from the transmit buffer to the receive buffer. Local loopback is most important for Wimp-based server programs, as it allows them to offer their services to the local station as easily as to all other stations on the Econet.

Broadcasts are also subject to local loopback, but differ slightly in that even if local loopback takes place access to the Econet network will still occur. This is to ensure the semantics of broadcasts. This does however cause a slight problem, in that a broadcast can be initiated to the local station via local loopback and succeed, but still fail externally with – for example – Status_NoClock. This is a slight semantic deviation that you must bear in mind when writing software that may communicate with itself or other software running on your machine by using broadcasts.

The other problem that can occur with local loopback is premature reception, caused by transmission and reception using the same port (whether by accident, or as a feature of the protocol's design), and the length of the transmission being less than or equal to the length of the receive buffer. For example to communicate with an Econet Bridge to find out if a particular net exists code like this will generally work:

```
 10 SYS "Econet_AllocatePort" TO port%
 20 SYS "Econet_CreateReceive", port%, 0, 0, RxBuffer%, 10 TO handle%
 30 $TxBuffer%  = "Bridge"
 40 TxBuffer%?6 = port%
 50 TxBuffer%?7 = NetToTestFor%
 60 SYS "Econet_DoTransmit", &83, &9C, &FF, &FF, TxBuffer%, 8, 5, 5
 80 SYS "Econet_WaitForReception", handle%, 10, 0 TO status%
 90 IF status% = 9 THEN
100   PRINT "Net number ";STR$(NetToTestFor%);" exists."
110 ENDIF
120 SYS "Econet_DeAllocatePort",port%
```

However, when the port allocator returns port &9C the program will be subject to unexpected local loopback, and the broadcast will be received internally as well as transmitted externally. This will cause the program to incorrectly report a reception from the bridge, and to interpret it as a reply indicating the existence of the desired net. The most effective way to prevent this is to only create the receive control block after the

transmission has completed. In the case above you could simply change line 20 to be line 70. In general it is not acceptable to transmit a request before opening the receive control block for the reply; however, some pre-existing protocols force the issue.

It is worth noting that the use of local loopback in the Wimp environment does require that the polling of receptions and transmissions be interleaved with calls to Wimp_Poll. If this is not done, although the data will be transferred, no notice will be taken because control will not be transferred to the receiving program.

Local loopback with zero length packets will cause the machine to lock up.

Immediate operations are not subject to local loopback.

Local loopback is not supported by RISC OS 2.

Immediate operations

There is a second class of network operations called immediate operations. These operations don't require the explicit co-operation of the destination machine; instead the co-operation is provided by the Econet software in that machine. Immediate operations are similar semantically to normal transmissions but, because they have no need for a port number, have a type instead of a flag; and most also require an extra input value. They have a separate pair of SWI calls to cause them to happen: Econet_StartImmediate (page 2-691) and Econet_DoImmediate (page 2-693).

The call Econet_StartImmediate returns a transmit handle in exactly the same way as Econet_StartTransmit and that handle should be polled and abandoned in the same way. The call Econet_DoImmediate returns a status just as Econet_DoTransmit does.

There are nine types of immediate operations:

1	Econet_Peek	Copy memory from the destination machine
2	Econet_Poke	Copy memory to the destination machine
3	Econet_JSR	Cause JSR/BL on the destination machine
4	Econet_UserProcedureCall	Execute User remote procedure call
5	Econet_OSProcedureCall	Execute OS remote procedure call
6	Econet_Halt	Halt the destination machine
7	Econet_Continue	Continue the destination machine
8	Econet_MachinePeek	Machine peek of the destination machine
9	Econet_GetRegisters	Return registers from the destination machine

The last one, Econet_GetRegisters, can only be transmitted by or received on RISC OS based machines, whereas all the others can be transmitted or received by BBC or Master series computers. The reason for this is that Econet_GetRegisters is specific to the ARM processor.

As noted earlier, Immediate operations are not subject to local loopback.

Econet_Peek and Poke

The poke operation is very similar to a transmit, in that data is moved from the transmitting station to the receiving station. The difference is that the address at which the data is received is supplied by the transmitting station. Peek is the inverse of poke; data is moved from the receiving station into the transmitting station.

Before the receiving station allows the data to be transferred (in or out), it validates the address range supplied by the transmitting station. This validation – done using the SWI XOS_ValidateAddress – takes place in an IRQ process, so having IRQs disabled will affect a machine’s ability to be peeked or poked.

This validation does not take place under RISC OS 2.

Econet_JSR, UserProcedureCall and OSProcedureCall

JSR, UserProcedureCall, and OSProcedureCall are all very similar. They send a small quantity of data, referred to as the argument buffer or arguments, to the destination machine; they then force it to execute a particular section of code. When received a JSR actually does a BL to the address given in R1, whereas UserProcedureCall and OSProcedureCall cause events to occur. These events are:

8	Event_Econet_UserRPC
16	Event_Econet_OSProc

After reception the arguments are buffered so that they may be used by the code that is called, either directly by a BL or indirectly via an event. The format of the arguments buffer is as follows: word 0 is the length (in bytes) of the arguments, then the arguments follow this first word and may be null (ie the length may be zero).

Conditions on entry to event code

The conditions on entry to the event code are:

R0 = Event number (either Event_Econet_UserRPC or Event_Econet_OSProc)
R1 = Address of the argument buffer
R2 = RPC number (passed in R1 on the transmitting station)
R3 = Station that sent the RPC
R4 = Net that sent the RPC

Conditions on entry to JSR code

The conditions on entry to code that is BL’d to for a JSR are:

R1 = Address of the argument buffer
R2 = Address of the code being executed
R3 = Station that sent the JSR
R4 = Net that sent the JSR

Format of the argument buffer

The format of the argument buffer is exactly the same in all cases. If, in the case of a JSR, the call address transmitted from the remote station is –1 (&FFFFFFFF) then the execution address will be the argument buffer itself; this means that relocatable ARM code can be sent as a JSR. Registers R0 to R4 can be used as they are preserved by the Econet software, and R13 can also be used as a full descending stack.

The transmission of Econet_OSProcedureCall is intended for use solely by system software, and is only documented here for completeness. The transmission of Econet_JSR is only provided as a compatibility feature to allow interworking with BBC and Master computers.

Econet_UserProcedure calls

The Econet_UserProcedureCall is the best method for this style of communications. It does however have some restrictions. The first of these is the most important – it is executed in the destination machine as an event caused by an interrupt, and so it has all the normal restrictions applied to interrupt code. This means that code directly executed as a result of Event_Econet_UserRPC must be fast and clean, and must not call any of the normal input or output SWI routines nor call the filing system, either directly or indirectly. This is paramount if the integrity of the destination machine is to be ensured. However, you can copy away the arguments passed and signal to a foreground task (by altering a flag) that the procedure call has arrived. It is most important that you copy the arguments away, because the buffer that they are in is only valid for the duration of the event call. This means that R1 will point to the arguments whilst you are processing the event, but afterwards the argument buffer may be overwritten. If the requirements for the processing of the call are small then it is possible to do it all within the event. An example of this is a modification of the program presented earlier that returned the time. This new program sends the time in response to a User RPC, rather than a normal packet:

```
Start   MOV     r0, #EventV             ; The vector we want is EventV
        ADR     r1, Event               ; Where to goto when it happens
        MOV     r2, #0                  ; Required so that we can release
        SWI     XOS_Claim
```

```
        MOVVC   r0, #14                 ; Enable event
        STRVC   r0, ClaimedFlag         ; Set it to a non-zero value
        MOV     r1, #Event_Econet_UserRPC
        SWIVC   XOS_Byte
        MOVVC   r0, #14                 ; Enable event
        MOV     r1, #Event_Econet_Tx
        SWIVC   XOS_Byte
        MOV     pc, lr

Event   TEQ     r0, #Event_Econet_UserRPC
        BNE     LookForTx
        TEQ     r2, #RPC_SendTime       ; Is it for us?
        MOVNE   pc, lr                  ; If not, exit as fast as possible
        LDR     r0, [r1, #0]            ; Get size of arguments
        TEQ     r0, #1                  ; Check that it is right
        MOVNE   r0, #Event_Econet_UserRPC; Restore exit registers
        MOVNE   pc, lr                  ; If not, exit as fast as possible

        STMFD   sp!, {r5-r7}            ; Only R1 to R4 are free for use
                                        ; R4.R3 is the reply address
        MOV     r6, r3                  ; Save the station number for later
        MOV     r5, r1                  ; Preserve arguments pointer
        MOV     r0, #Module_Claim
        MOV     r3, #8 + 5              ; Two words and five bytes required
        SWI     XOS_Module              ; Memory MUST come from RMA
        BVS     Exit

        ADD     r1, r2, #8              ; Get the address of the 5 bytes
        MOV     r0, #3                  ; Set OS_Word reason code
        STRB    r0, [r1]                ; Read as a five byte time
        MOV     r0, #14                 ; Read from the real time clock
        SWI     XOS_Word
        BVS     Exit

        MOV     r0, #0                  ; Flag byte
        MOV     r3, r4                  ; Net number
        MOV     r4, r1                  ; Get the address of the 5 bytes
        LDRB    r1, [r5, #4]            ; The reply port the client sent
        MOV     r2, r6                  ; Station number
        MOV     r5, #5                  ; Number of bytes to send
        MOV     r6, #ReplyCount
        MOV     r7, #ReplyDelay
        SWI     XEconet_StartTransmit
        BVS     Exit

        SUB     r4, r2, #8              ; R4 now in R2
        STR     r0, [r4, #4]            ; Save TxHandle in record
        ADR     r1, TxList              ; Address of the head of the list
        LDR     r2, [r1, #0]            ; Head of the list
        STR     r2, [r4, #0]            ; Add the list to new record
        STR     r4, [r1, #0]            ; Make this record the list head
Exit
        LDMFD   sp!, {r5-r7, pc}        ; Return claiming vector
```

```
LookForTx
        TEQ     r0, #Event_Econet_Tx
        MOVNE   pc, lr                  ; This event has only R0 to R2
        STMFD   sp!, {r3, lr}           ; Get two extra registers
        ADR     r3, TxList              ; The address of the head of list
        LDR     r14, [r3]               ; The first record in the list
        B       StartLooking
NextTx
        MOV     r3, r14                 ; Search the next list entry
        LDR     r14, [r3]               ; Get the link address
StartLooking
        CMP     r14, #0                 ; Is this the end of the list?
        MOVLE   r0, #Event_Econet_Tx    ; Restore entry conditions
        LDMLEFD sp!, {r3, pc}           ; Return, continuing to next owner
        LDR     r0, [r14, #4]           ; Get the handle for this record
        TEQ     r0, r1                  ; Is this event one of ours?
        BNE     NextTx                  ; No, try next record in list

        LDR     r2, [r14]               ; Get the remainder of the list
        STR     r2, [r3]                ; Remove this record from list
        SWI     XEconet_AbandonTransmit
        MOV     r0, #Module_Free
        MOV     r2, r14                 ; The record address
        SWI     XOS_Module              ; Return memory to RMA, ignore error
        LDMFD   sp!, {r3, lr, pc}       ; Return, claiming vector
```

You will notice how much simpler this program is when compared to the program shown earlier.

Econet_OSProcedure calls

There are five defined OS procedure calls for which only two have implementations under RISC OS. The five are:

0	Econet_OSCharacterFromNotify
1	Econet_OSInitialiseRemote
2	Econet_OSGetViewParameters
3	Econet_OSCauseFatalError
4	Econet_OSCharacterFromRemote

OSCharacterFromNotify

Econet_OSCharacterFromNotify causes the character received to be inserted into the keyboard buffer; the code that does so looks like this:

```
InsertCharacter                         ; R1 points at the argument buffer
        MOV     r0, #138                ; Insert into buffer OS_Byte
        LDRB    r2, [r1, #4]            ; Get character from buffer
        MOV     r1, #0                  ; Buffer is keyboard
        SWI     XOS_Byte
```

Whilst the desktop is running the NetFiler module provides a different handler for characters from notify. It bundles them up by station, and when none have been received for a while sends them as a Wimp message, displaying them using Wimp_ReportError. For more information see the documentation of Message_Notify on page 3-236.

OSCauseFatalError

Econet_OSCauseFatalError does exactly what its name implies. In fact it calls SWI OS_GenerateError directly from the event routine; normally this would be illegal, but since this is what the RPC is for, that is what it does. It should be observed that this can have a disastrous effect on the integrity of the machine and is not a recommended action; it is provided only for compatibility reasons.

Econet_Halt and Continue

Halt and continue are only acted upon by BBC and Master series machines; there is no implementation for receiving halt or continue on RISC OS machines or RISC iX machines.

Econet_MachinePeek

Machine peek is similar to peek, except that it is not possible to specify the address to be peeked, but rather four bytes are returned that identify the machine that is being machine peeked. Machine peek is used by some of the system software in RISC OS to quickly decide if a particular machine is present or not. The four bytes returned by machine peek are as follows:

Byte(s)	Value
1 and 2	Machine type number
3	Software version number
4	Software release number

Machine type numbers

Machine type numbers are as follows:

&0000	Reserved
&0001	Acorn BBC Micro Computer (OS 1 or OS 2)
&0002	Acorn Atom
&0003	Acorn System 3 or System 4
&0004	Acorn System 5
&0005	Acorn Master 128 (OS 3)
&0006	Acorn Electron (OS 0)
&0007	Acorn Archimedes (OS 6)
&0008	Reserved for Acorn
&0009	Acorn Communicator

&000A	Acorn Master 128 Econet Terminal
&000B	Acorn FileStore
&000C	Acorn Master 128 Compact (OS 5)
&000D	Acorn Ecolink card for Personal Computers
&000E	Acorn UNIX workstation
&000F to &FFF9	Reserved
&FFFA	SCSI Interface
&FFFB	SJ Research IBM PC Econet interface
&FFFC	Nascom 2
&FFFD	Research Machines 480Z
&FFFE	SJ Research File Server
&FFFF	Z80 CP/M

Software version and release number

The software version and release numbers are stored in two bytes. These two bytes are encoded in packed BCD (Binary Coded Decimal) and represent a number between 0 and 99. The easiest way to display packed BCD is to print it as if it was hexadecimal data:

```
ReportStationVersion
        MOV     r2, r0                  ; Station number in R0
        MOV     r3, r1                  ; Net number in R1
        MOV     r0, #Econet_MachinePeek
        ADR     r4, Buffer
        MOV     r5, #?Buffer
        MOV     r6, #40                 ; Count
        MOV     r7, #5                  ; Delay
        SWI     XEconet_DoImmediate
        MOVVS   pc, lr
        TEQ     r0, #Status_Transmitted
        BEQ     PrintVersion
        TEQ     r0, #Status_NotListening ; from Machine peek
        MOVEQ   r0, #Status_NotPresent  ; return as "Not present"
        ADR     r1, Buffer
        MOV     r2, #?Buffer
        SWI     XEconet_ConvertStatusToError
        MOV     pc, lr
```

Networking

```
PrintVersion
        LDR     r3, [r2]                ; Buffer address on exit from SWI
        MOV     r0, r3, ASR #24         ; Get top byte
        ADR     r1, Buffer
        MOV     r2, #?Buffer
        SWI     XOS_ConvertHex2         ; Print BCD as hex
        SWIVC   XOS_Write0              ; Display output
        SWIVC   XOS_WriteI+"."          ; Divide release from version number
        MOVVC   r0, r3, ASR #16         ; Get version number in place
        ANDVC   r0, r0, #&FF            ; Only the version number
        ADRVC   r1, Buffer
        MOVVC   r2, #?Buffer
        SWIVC   XOS_ConvertHex2         ; Print BCD as hex
        SWIVC   XOS_Write0              ; Display output
        MOV     pc, lr
```

We recommend that when using Econet_MachinePeek you use a Count of 40 and a Delay of 5.

Econet_GetRegisters

Econet_GetRegisters is similar to machine peek, in that a fixed amount of information is returned from the destination machine; in this case it is 80 bytes (20 words). The registers are returned in the following order: R0 to R14, PC plus PSR, R13_irq, R14_irq, R13_svc, and R14_svc. The FIQ registers are not returned because they are used by the Econet software, and so would always be the same, and of no interest since they would reflect the state of the part of the Econet software that transmits data. It is worthwhile aligning the receive buffer for a machine peek so that each of the 20 words is on a word boundary; this makes loading them easier.

Protection against immediate operations

Because these immediate operations can be quite intrusive it is possible to prevent their reception by manipulating an internal variable of the Econet software. There is one bit in this internal variable for each operation, and you can set or clear each bit. There is also a default value for each bit which is held in CMOS RAM. The SWI that allows you to manipulate this internal variable is Econet_SetProtection (page 2-681). These bits are held in a single word; the bit assignments are as follows:

Bit	Immediate operation protected against
0	Peek
1	Poke
2	Remote JSR
3	User procedure call
4	OS procedure call
5	Halt
6	Continue – must be zero on RISC OS computers

7	Machine peek – must be zero on RISC OS computers
8	Get registers
9 - 30	Reserved – must be zero.
31	Write new value to the CMOS RAM

To protect against or disable the reception of a particular immediate operation, the appropriate bit should be set in the internal variable. The SWI Econet_SetProtection call replaces the OldValue with the NewValue, The NewValue is calculated like this:

NewValue = (OldValue AND R1) EOR R0.

Altering the protection held in CMOS RAM

When the Econet software is started up (as a result of Ctrl-Break, or *RMReInit) then the value held in CMOS RAM will be used to initialise the internal variable. To alter the value held in CMOS RAM the entry value of R0 to SWI Econet_SetProtection should have bit 31 set, which causes the resultant value to be written not only to the internal variable, but also to the CMOS RAM. To read the current value you should use SWI Econet_SetProtection with R0=0, and R1=&FFFFFFFF.

Reading your station and net numbers

To establish what your station number is and which net you are connected to (if you have more than one), the Econet software provides a call to return these two values: Econet_ReadLocalStationAndNet (page 2-674). If you don’t have more than one net then the net number (returned in R1) will be zero.

The local net number is in fact obtained from a bridge whenever the Econet module is initialised (eg when the machine is turned on). If this fails, say because there is no clock or the bridge is not switched on, then the local net number is reported as zero.

These values are the same as those reported by *Help Station (in fact *Help Station calls SWI Econet_ReadLocalStationAndNet to get the values).

Extracting station numbers from a string

To ensure that all Econet oriented software presents a consistent user interface there is a SWI call to read a station and/or net number from a supplied string. This call, Econet_ReadStationNumber (page 2-683), is used by both NetFS and NetPrint for all their command line processing. In the case of software that has a concept of a current station (and net) number the return value of –1 should mean ‘use the existing value’ – this is how *FS works, for example. Where there isn’t a current value, as would be expected in a transient command such as *Notify, the return of –1 for the station number

should be treated as an error and the return of –1 as a net number should imply the use of zero as a net number. The following is the beginning (and some of the end) of a transient command:

```
CommandStart
        LDRB    r0, [r1]                ; Check the first argument exists
        TEQ     r0, #0                  ; Zero means no arguments
        BEQ     SyntaxError             ; Exit with error

        SWI     XEconet_ReadStationNumber
        MOVVS   pc, lr                  ; Must be able to cope
        CMP     r2, #-1                 ; No station number given
        BEQ     NoStationNumberError
        CMP     r3, #-1                 ; No net number given
        MOVEQ   r3, #0                  ; Means use zero

        MOV     pc, lr

SyntaxError
        ADR     r0, ErrorGetRegsSyntax
        ORRS    pc, lr, #VFlag

ErrorGetRegsSyntax
        DCD     ErrorNumber_Syntax
        DCB     "Syntax: *Command <Station number>"
        DCB     0
        ALIGN

NoStationNumberError
        ADR     r0, ErrorUnableToDefault
        ORRS    pc, lr, #VFlag

ErrorUnableToDefault
        DCD     ErrorNumber_UnableToDefault
        DCB     "Either a station number or a full"
        DCB     " network address is required"
        DCB     0
        ALIGN
```

Converting station and net to a string

The kernel provides two inverse functions that convert a station and net number pair into a string. See *OS_ConvertFixedNetStation (SWI &E9)* on page 1-486 and *OS_ConvertNetStation (SWI &EA)* on page 1-488 for exact details.

Conventions and values

The following conventions apply to the various values that the Econet uses:

Station numbers

Station numbers are normally in the range 1 to 254. The station number zero is used in SWI Econet_CreateReceive to indicate that reception may occur from any station. The station number 255 is used in SWI Econet_StartTransmit and in SWI Econet_DoTransmit to indicate that a broadcast is to take place. Station number 255 is also used in SWI Econet_CreateReceive to indicate that reception may occur from any station; you may also use station number zero for this purpose, but its use is deprecated, and may be withdrawn in the future.

Net numbers

Net numbers are normally in the range 1 to 251. The value zero means the local Econet net; in a SWI Econet_CreateReceive it is taken to indicate that reception may occur from any net. The net numbers 255, 254 and 253 are used in SWI Econet_StartTransmit and in SWI Econet_DoTransmit to indicate that a broadcast is to take place. Net number 255 is also used in SWI Econet_CreateReceive to indicate that reception may occur from any station; the use of zero to indicate wild reception is deprecated.

Although RISC OS fully supports top-bit-set net numbers (ie 128 - 251), certain Econet devices – such as bridges – will not propagate them, leading to problems. You should beware of this.

Port numbers

Port numbers are normally in the range 1 to 254, although some values are reserved – as shown in the table below:

Port	**Allocation**
&54	DigitalServicesTapeStore
&99	FileServerCommand
&9C	Bridge
&9E	PrinterServerInquiryReply
&9F	PrinterServerInquiry
&B0	FindServer
&B1	FindServerReply
&B2	TeletextServerCommand
&B3	TeletextServerPage
&D0	OldPrinterServerData
&D1	PrinterServerData
&D2	TCPIPProtocolSuite

&D3	SIDFrameSlave
&D4	Scrollarama
&D5	Phone
&D6	BroadcastControl
&D7	BroadcastData
&D8	ImpressionLicenceChecker
&D9	DigitalServicesSquirrel
&DA	SIDSecondary
&DB	DigitalServicesSquirrel2
&DC	DataDistributionControl
&DD	DataDistributionData
&DE	ClassROM
&DF	PrinterSpoolerCommand

Port numbers zero and 255 currently have a special meaning: they may be used as arguments to SWI Econet_CreateReceive to indicate that reception may occur regardless of the port number on the incoming packet. This use of zero to indicate wild reception is deprecated, and will be withdrawn in the future.

For an allocation of a port number you must contact Acorn.

Flag bytes

Flag byte values are in the range 0 to 127 (&7F). When passed in a word to a SWI, bits 8 - 31 inclusive must be zero. Bit 7 is ignored by RISC OS, to maintain compatibility with some older software that used this bit. To clarify, flag bytes &87 and &07 are acceptable as input to a transmission SWI (and both represent the value &07), but &107 is not acceptable. Reception SWIs all return values with bit 7 clear (ie &00 to &7F).

Transmission semantics

The transmission semantics are simple. When a transmission is started the client's control information (passed in registers) is stored in a record in a linked list within Econet workspace. At regular intervals the list is scanned, and those records that should be actually transmitted at that moment are passed to the FIQ software. When that particular transmission attempt completes the status of the record is changed accordingly. This means that if two transmissions are started at the same time, they will interleave their transmission retries.

When a transmission has completed but failed:

- if the count is non-zero the delay is added to the predicted start time to give the next start time
- otherwise the status is set to *Status_NotListening* (or *Status_NetError*).

This means that as far as possible the time out time will be the Delay multiplied by the (Count – 1).

Local loopback

Versions of RISC OS after RISC OS 2 have added support for local loopback. Transmissions directed at your own station number will be ‘received’ if there is an acceptable receive block open by directly copying the data. This applies to broadcast transmissions and wild receptions as well as to calls that explicitly address your machine.

Service Calls

Service_ReAllocatePorts
(Service Call &48)

Econet restarting

On entry

R1 = &48 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This call is made whenever Econet restarts. It is then up to the Econet software to allocate ports, set up TxCBs and RxCBs, etc.

Service_EconetDying
(Service Call &56)

Econet is about to leave

On entry

R1 = &56 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This call is made whenever Econet is about to leave. It is then up to the Econet software to release ports, delete RxCBs and TxCBs etc.

Service_ProtocolDying
(Service Call &83)

Part of the AUN Driver Control Interface

Use

This service call is part of the AUN Driver Control Interface, used to interface a network interface's driver module to a protocol module. Third parties wishing to develop network interfaces for use with AUN may obtain further details on request from Acorn.

Service_FindNetworkDriver
(Service Call &84)

Part of the AUN Driver Control Interface

Use

This service call is part of the AUN Driver Control Interface, used to interface a network interface's driver module to a protocol module. Third parties wishing to develop network interfaces for use with AUN may obtain further details on request from Acorn.

Service_NetworkDriverStatus (Service Call &8B)

Part of the AUN Driver Control Interface

Use

This service call is part of the AUN Driver Control Interface, used to interface a network interface's driver module to a protocol module. Third parties wishing to develop network interfaces for use with AUN may obtain further details on request from Acorn.

SWI Calls

Econet_CreateReceive (SWI &40000)

Creates a Receive Control Block

On entry

R0 = port number
R1 = station number
R2 = net number
R3 = buffer address
R4 = buffer size in bytes

On exit

R0 = handle
R2 = 0 if R2 on entry is the local 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 creates a Receive Control Block (RxCB) to control the reception of an Econet packet. It returns a handle to the RxCB.

The buffer must remain available all the time that the RxCB is open, as data received over the Econet is read directly from hardware to the buffer. You must not use memory in application space if your program is to run under the Desktop. Instead, you should use

memory from the RMA. To do so, claim the memory using OS_Module 6 (see page 1-237), and – after abandoning the receive control block – return the space to the RMA using OS_Module 7 (see page 1-238).

Related SWIs

Econet_ExamineReceive (page 2-659), Econet_WaitForReception (page 2-664), Econet_AbandonAndReadReceive (page 2-695)

Related vectors

None

Econet_ExamineReceive (SWI &40001)

Reads the status of an RxCB

On entry

R0 = handle

On exit

R0 = status

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 status of an RxCB, which may be one of the following:

7	Status_RxReady
8	Status_Receiving
9	Status_Received

It returns less information than Econet_ReadReceive, so is faster and corrupts fewer registers. You should use it to poll a reception when not using Econet_WaitForReception.

Related SWIs

Econet_CreateReceive (page 2-657), Econet_WaitForReception (page 2-664), Econet_ConvertStatusToString (page 2-675), Econet_ConvertStatusToError (page 2-677)

Related vectors

None

Econet_ReadReceive (SWI &40002)

Returns information about a reception, including the size of data

On entry

R0 = handle

On exit

R0 = status
R1 = 0, or flag byte if R0 = 9 (Status_Received) on exit
R2 = port number
R3 = station number
R4 = net number
R5 = buffer address
R6 = buffer size in bytes, or amount of data received if R0 = 9 on exit (Status_Received)

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 information about a reception; most importantly, it tells you how much data was received, if any, and the address of the buffer in which it was placed. The buffer address is the same as that passed to Econet_CreateReceive (page 2-657). You can call this SWI before a reception has occurred.

The status of the RxCB may be one of the following:

7	Status_RxReady
8	Status_Receiving
9	Status_Received

The returned values in R3 and R4 (the net and station numbers) are those of the transmitting station if the status is Status_Received; otherwise they are the same values that were passed in to Econet_CreateReceive.

Related SWIs

Econet_CreateReceive (page 2-657), Econet_WaitForReception (page 2-664), Econet_AbandonAndReadReceive (page 2-695)

Related vectors

None

Econet_AbandonReceive (SWI &40003)

Abandons an RxCB

On entry

R0 = handle

On exit

R0 = status

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call abandons an RxCB, returning its memory to the RMA. The reception may have completed (R0 = 9 – Status_Received – on exit), in which case the information in the RxCB (such as the sending station number, and the amount of data sent) will be lost. The data in the receive buffer remains unaffected. If the reception is in progress when this SWI is called, then information in the RxCB is lost, as above.

Related SWIs

Econet_CreateReceive (page 2-657), Econet_WaitForReception (page 2-664), Econet_AbandonAndReadReceive (page 2-695)

Related vectors

None

Econet_WaitForReception (SWI &40004)

Polls an RxCB, reads its status, and abandons it

On entry

R0 = handle
R1 = delay in centiseconds
R2 = 0 to ignore Escape; else Escape ends waiting

On exit

R0 = status
R1 = 0, or flag byte if R0 = 9 (Status_Received) on exit
R2 = port number
R3 = station number
R4 = net number
R5 = buffer address
R6 = buffer size in bytes, or amount of data received if R0 = 9 on exit (Status_Received)

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode and in USR mode

Re-entrancy

SWI is not re-entrant

Use

This call repeatedly polls an RxCB (that you have already set up with Econet_CreateReceive) until a reception occurs, or a timeout occurs, or the user interferes (say by pressing Escape). It then reads the status of the RxCB before abandoning it.

The status of the RxCB may be one of the following:

8	Status_Receiving
9	Status_Received
10	Status_NoReply
11	Status_Escape

The returned values in R3 and R4 (the net and station numbers) are those of the transmitting station if the status is Status_Received; otherwise they are the same values that were passed in to SWI Econet_CreateReceive.

Note that because this interface enables interrupts it should not be called from within either interrupt service code or event routines.

During the loop when the polling of the RxCB and of Escape takes place, the processor is put in USR mode with IRQs enabled; this allows callbacks to occur.

Related SWIs

Econet_ExamineReceive (page 2-659), Econet_ReadReceive (page 2-661),
Econet_AbandonReceive (page 2-663),
Econet_AbandonAndReadReceive (page 2-695)

Related vectors

None

Econet_EnumerateReceive (SWI &40005)

Returns the handles of open RxCBs

On entry

R0 = index (1 to start with first receive block)

On exit

R0 = handle (0 if no more receive blocks)

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the handles of open RxCBs. On entry R0 is the number of the RxCB being asked for (1, 2, 3…). If the value of R0 is greater than the number of open RxCBs, then the value returned as the handle will be 0, which is an invalid handle.

This call should not be made from an IRQ or event routine as, although it will not fail, errors and omissions are likely to occur in the returned information.

Related SWIs

Econet_CreateReceive (page 2-657),
Econet_ReadReceive (page 2-661), Econet_AbandonReceive (page 2-663)

Related vectors

None

Econet_StartTransmit (SWI &40006)

Creates a Transmit Control Block and starts a transmission

On entry

R0 = flag byte
R1 = port number
R2 = station number
R3 = net number
R4 = buffer address
R5 = buffer size in bytes
R6 = count
R7 = delay in centiseconds

On exit

R0 = handle
R1 corrupted
R2 = buffer address
R3 = station number
R4 = 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 creates a Transmit Control Block (TxCB) to control the transmission of an Econet packet. It then starts the transmission.

The buffer must remain available all the time that the TxCB is open, as data transmitted over the Econet is read directly from the buffer to hardware. You must not use memory in application space if your program is to run under the Desktop. Instead, you should use memory from the RMA. To do so, claim the memory using OS_Module 6 (see page 1-237), and – after abandoning the transmit control block – return the space to the RMA using OS_Module 7 (see page 1-238).

The value returned in R4 (the net number) will be the same as that passed in R3 unless that number is equal to the local net number; in that case the net number will be returned as zero.

Related SWIs

Econet_PollTransmit (page 2-669), Econet_AbandonTransmit (page 2-671), Econet_DoTransmit (page 2-672)

Related vectors

None

Econet_PollTransmit (SWI &40007)

Reads the status of a TxCB

On entry

R0 = handle

On exit

R0 = status

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 status of a TxCB, which may be one of the following:

0	Status_Transmitted
1	Status_LineJammed
2	Status_NetError
3	Status_NotListening
4	Status_NoClock
5	Status_TxReady
6	Status_Transmitting

Related SWIs

Econet_StartTransmit (page 2-667), Econet_AbandonTransmit (page 2-671)

Related vectors

None

Econet_AbandonTransmit (SWI &40008)

Abandons a TxCB

On entry

R0 = handle

On exit

R0 = status

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 abandons a TxCB, returning its memory to the RMA. The returned status is the same as for Econet_PollTransmit.

Related SWIs

Econet_StartTransmit (page 2-667), Econet_PollTransmit (page 2-669)

Related vectors

None

Econet_DoTransmit (SWI &40009)

Creates a TxCB, polls it, reads its status, and abandons it

On entry

R0 = flag byte
R1 = port number
R2 = station number
R3 = net number
R4 = buffer address
R5 = buffer size in bytes
R6 = count
R7 = delay in centiseconds

On exit

R0 = status
R1 corrupted
R2 = buffer address
R3 = station number
R4 = net number

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode and in USR mode

Re-entrancy

SWI is not re-entrant

Use

This call creates a TxCB and repeatedly polls it until it finishes transmission, or it exceeds the count of retries. It then reads the final status of the TxCB before abandoning it.

The status of the TxCB may be one of the following:

0	Status_Transmitted
1	Status_LineJammed
2	Status_NetError
3	Status_NotListening
4	Status_NoClock

The value returned in R4 (the net number) will be the same as that passed in R3 unless that number is equal to the local net number; in that case the net number will be returned as zero.

Note that because this interface enables interrupts it should not be called from within either interrupt service code or event routines.

During the loop when the polling of the TxCB and of Escape takes place, the processor is put in USR mode with IRQs enabled; this allows callbacks to occur.

Related SWIs

Econet_StartTransmit (page 2-667), Econet_PollTransmit (page 2-669), and Econet_AbandonTransmit (page 2-671)

Related vectors

None

Econet_ReadLocalStationAndNet (SWI &4000A)

Returns a computer's station number and net number

On entry

No parameters passed in registers

On exit

R0 = station number
R1 = net number

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 returns a computer's station number and Econet net number. The net number will be zero if there are no Econet bridges present on the network.

For more information, see the section entitled *Reading your station and net numbers* on page 2-647.

Related SWIs

None

Related vectors

None

Econet_ConvertStatusToString (SWI &4000B)

Converts a status to a string

On entry

R0 = status
R1 = pointer to buffer
R2 = buffer size in bytes
R3 = station number
R4 = net number

On exit

R0 = buffer
R1 = updated buffer address
R2 = updated buffer size 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 converts a status to a string found in the messages file. This is then copied into RAM, including the station and net numbers, giving a string such as:

```
Network station 59.254 not listening
```

If the status given in R0 is invalid (ie not in the range 0 - 14), this will cause a data abort or an address exception. If the station/net number given in R3/R4 is invalid, no station information is given.

Under RISC OS 2 the string is not read from the messages file, but is instead read direct from the ROM.

Related SWIs

Econet_ConvertStatusToError (page 2-677)

Related vectors

None

Econet_ConvertStatusToError (SWI &4000C)

Converts a status to a string, and then generates an error

On entry

R0 = status
R1 = pointer to error buffer
R2 = error buffer size in bytes
R3 = station number
R4 = net number

On exit

R0 = pointer to error block
V flag is set

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 status to a string found in the messages file. This is then copied into RAM, including the station and net numbers, giving a string such as:

```
Network station 59.254 not listening
```

If the station/net number given in R3/R4 is invalid, no station information is given.

Finally this call returns an error by setting the V flag, with R0 pointing to the error block.

If you use a buffer address of zero, then the string is left in a buffer in the MessageTrans workspace.

Under RISC OS 2 the string is not read from the messages file, but is instead read direct from the ROM.

Related SWIs

Econet_ConvertStatusToString (page 2-675)

Related vectors

None

Econet_ReadProtection (SWI &4000D)

Reads the current protection word for immediate operations

On entry

No parameters passed in registers

On exit

R0 = current protection value

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 protection word for immediate operations. Various bits in the word, when set, disable corresponding immediate operations:

Bit	Immediate operation
0	Peek
1	Poke
2	Remote JSR
3	User procedure call
4	OS procedure call
5	Halt
6	Continue – always zero on RISC OS computers
7	Machine peek – always zero on RISC OS computers
8	Get registers
9 - 31	Reserved – must be zero

Note – This call is deprecated. You should preferably use the call Econet_SetProtection (page 2-681) to read the protection word instead of this call.

Related SWIs

Econet_SetProtection (page 2-681)

Related vectors

None

Econet_SetProtection (SWI &4000E)

Sets or reads the protection word for immediate operations

On entry

R0 = EOR mask word
R1 = AND mask word

On exit

R0 = old value

Interrupts

Interrupts are enabled on write-through to CMOS, preserved otherwise
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the protection word for immediate operations as follows:

New value = (old value AND R1) EOR R0

Various bits in the word, when set, disable corresponding immediate operations:

Bit	Immediate operation
0	Peek
1	Poke
2	Remote JSR
3	User procedure call
4	OS procedure call
5	Halt
6	Continue – must be zero on RISC OS computers
7	Machine peek – must be zero on RISC OS computers

8	Get registers
9 - 30	Reserved – must be zero
31	Write new value to the CMOS RAM

Normally this call sets or reads the current value of the word. A default value for this word is held in CMOS RAM.

The most useful values of R0 and R1 are:

Action	R0	R1
Set current value	new value (0 - &1FF)	0
Read current value	0	&FFFFFFFF
Set new default value	&80000000 + new value	0

You should use this call to read the value of the protection word, rather than Econet_ReadProtection (page 2-679).

Using this call to read is also the preferred method for detecting the presence of the Econet drivers, since doing so can never return an unexpected error. Detecting the error 'No such SWI' allows software dependent upon Econet to report its absence. Example code is given in the section entitled *Application notes* on page 2-704.

Related SWIs

None

Related vectors

None

Econet_ReadStationNumber (SWI &4000F)

Extracts a station and/or net number from a supplied string

On entry

R1 = address of string to read

On exit

R1 = address of terminating space or control character
R2 = station number (–1 for not found)
R3 = net number (–1 for not found)

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call extracts a station and/or net number from a supplied string. For an example of its use, see the section entitled *Extracting station numbers from a string* on page 2-647.

Related SWIs

None

Related vectors

None

Econet_PrintBanner (SWI &40010)

Prints the string ‘Acorn Econet’ followed by a newline

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

This call prints the string ‘Acorn Econet’ followed by a newline. The string is fetched from a message file with the token ‘AcrnEco’. If the Econet network data clock is not present then this call instead prints the string ‘Acorn Econet, no clock’ followed by a newline. In this case, the token used is ‘EcoNClk’.

This call uses OS_Write0 and OS_NewLine, and so cannot be called from within either interrupt service code or event routines.

Related SWIs

None

Related vectors

None

Econet_ReadTransportType (SWI &40011)

Returns the underlying transport type to a given station

On entry

R0 = station number
R1 = net number
R2 = 2

On exit

R0, R1 preserved
R2 = transport type (0 ⇒ not known, 1 ⇒ Internet, 2 ⇒ Econet, 3 ⇒ Nexus)

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 used by clients to determine the underlying transport type to a given station. They can then use this information to determine the optimum transmission strategy to use, based on prior empirical knowledge of the different transport types.

This call is unnamed – but still available by number – in both RISC OS 2 and RISC OS 3 (version 3.00).

Related SWIs

None

Related vectors

None

Econet_ReleasePort (SWI &40012)

Releases a port number that was previously claimed

On entry

R0 = port number

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 releases a port number that was previously claimed by calling Econet_ClaimPort (page 2-690).

You must not use this call for port numbers that have been previously claimed using Econet_AllocatePort (page 2-688); instead, you must call Econet_DeAllocatePort (page 2-689).

Related SWIs

Econet_ClaimPort (page 2-690)

Related vectors

None

Econet_AllocatePort (SWI &40013)

Allocates a unique port number

On entry

No parameters passed in registers

On exit

R0 = port 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 allocates a unique port number that has not already been claimed or allocated.

When you have finished using the port number, you should call Econet_DeAllocatePort (page 2-689) to make it available for use again.

Related SWIs

Econet_DeAllocatePort (page 2-689)

Related vectors

None

Econet_DeAllocatePort (SWI &40014)

Deallocates a port number that was previously allocated

On entry

R0 = port number

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 deallocates a port number that was previously allocated by calling Econet_AllocatePort (page 2-688).

You must not use this call for port numbers that have been previously claimed using Econet_ClaimPort (page 2-690); instead, you must call Econet_ReleasePort (page 2-687).

Related SWIs

Econet_AllocatePort (page 2-688)

Related vectors

None

Econet_ClaimPort (SWI &40015)

Claims a specific port number

On entry

R0 = port number

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 claims a specific port number. If it has already been claimed or allocated, an error is generated.

When you have finished using the port number, you should call Econet_ReleasePort (page 2-687) to make it available for use again.

Related SWIs

Econet_ReleasePort (page 2-687)

Related vectors

None

Econet_StartImmediate (SWI &40016)

Creates a TxCB and starts an immediate operation

On entry

R0 = operation type
R1 = remote address or Procedure number
R2 = station number
R3 = net number
R4 = buffer address
R5 = buffer size in bytes
R6 = count
R7 = delay in centiseconds

On exit

R0 = handle
R1 corrupted
R2 = buffer address
R3 = station number
R4 = net number

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call creates a TxCB and starts an immediate operation. For full details see the section entitled *Immediate operations* on page 2-639.

The buffer must remain available all the time that the TxCB is open, as data transmitted over the Econet is read directly from the buffer to hardware. You must not use memory in application space if your program is to run under the Desktop. Instead, you should use memory from the RMA. To do so, claim the memory using OS_Module 6 (see page 1-237), and – after abandoning the transmit control block – return the space to the RMA using OS_Module 7 (see page 1-238).

The value returned in R4 (the net number) will be the same as that passed in R3 unless that number is equal to the local net number; in that case the net number will be returned as zero.

Related SWIs

Econet_DoImmediate (page 2-693)

Related vectors

None

Econet_DoImmediate (SWI &40017)

Creates a TxCB for an immediate operation, polls it, reads its status, and abandons it

On entry

R0 = operation type
R1 = remote address or procedure number
R2 = station number
R3 = net number
R4 = buffer address
R5 = buffer size in bytes
R6 = count
R7 = delay in centiseconds

On exit

R0 = status
R1 corrupted
R2 = buffer address
R3 = station number
R4 = net number

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode and in USR mode

Re-entrancy

SWI is re-entrant

Use

This call creates a TxCB for an immediate operation, and repeatedly polls it until it finishes transmission or it exceeds the count of retries. It then reads the final status of the TxCB before abandoning it. For full details see the section entitled *Immediate operations* on page 2-639.

The value returned in R4 (the net number) will be the same as that passed in R3 unless that number is equal to the local net number; in that case the net number will be returned as zero.

Note that because this interface enables interrupts it should not be called from within either interrupt service code or event routines.

During the loop when the polling of the TxCB and of Escape takes place, the processor is put in USR mode with IRQs enabled; this allows callbacks to occur.

Related SWIs

Econet_StartImmediate (page 2-691)

Related vectors

None

Econet_AbandonAndReadReceive (SWI &40018)

Abandons a reception and returns information about it, including the size of data

On entry

R0 = handle

On exit

R0 = status
R1 = 0, or flag byte if R0 = 9 (Status_Received) on exit
R2 = port number
R3 = station number
R4 = net number
R5 = buffer address
R6 = buffer size in bytes, or amount of data received if R0 = 9 on exit
(Status_Received)

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 abandons an RxCB, returning its memory to the RMA. It also returns information about the reception; most importantly, it tells you how much data was received, if any, and the address of the buffer in which it was placed. The buffer address is the same as that passed to Econet_CreateReceive (page 2-657). You can call this SWI before a reception has occurred.

The status of the RxCB may be one of the following:

7	Status_RxReady
9	Status_Received

The returned values in R3 and R4 (the net and station numbers) are those of the transmitting station if the status is Status_Received; otherwise they are the same values that were passed in to Econet_CreateReceive.

This call is not available in RISC OS 2, nor in RISC OS 3 (version 3.00).

Related SWIs

Econet_CreateReceive (page 2-657), Econet_ReadReceive (page 2-661), Econet_AbandonReceive (page 2-663)

Related vectors

None

Econet_Version (SWI &40019)

Returns the version of software for the underlying transport to a given station

On entry

R0 = station number
R1 = net number

On exit

R0, R1 preserved
R2 = version number × 100 (eg 547 for version 5.47)

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 used by clients to determine the version of software that handles the underlying transport to a given station. If both R0 and R1 are set to zero on entry, this call instead returns the version number of the top-level software to which RISC OS passes the Econet SWIs.

This call is not available in RISC OS 2, nor in RISC OS 3 (version 3.00).

Related SWIs

None

Related vectors

None

Econet_NetworkState (SWI &4001A)

Returns the state of the underlying transport to a given station

On entry

R0 = station number
R1 = net number

On exit

R0, R1 preserved
R2 = transport state (0 ⇒ fully functional, 1 ⇒ no clock signal)

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 state of the underlying transport to a given station. The state returned is transport type dependent, but you may always assume that a value of zero means that the transport is fully functional.

You should only use the returned value as a hint to the exact state; in other words, it is suitable for display but not for decision making. Using this call is no substitute for proper error handling; to determine if a particular transmit will fail, you must do the transmit and be prepared for it to fail.

Related SWIs

Econet_PrintBanner (page 2-684)

Related vectors

None

Econet_PacketSize (SWI &4001B)

Returns the maximum packet size recommended on the underlying transport to a given station

On entry

R0 = station number
R1 = net number

On exit

R0, R1 preserved
R2 = maximum permitted packet size, 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 returns the maximum recommended packet size on the underlying transport to a given station. Larger packets will not necessarily be rejected, but their use is not recommended. The size returned is transport type dependent.

This call is intended for use by modules supplying protocols; you do not need to use it in application software. For maximum efficiency the protocol module should negotiate the packet size once. Since the recommended packet size may differ between the stations at either end of a transmission, the protocol module should interrogate both stations and take the lower value returned.

Related SWIs

None

Related vectors

None

Econet_ReadTransportName (SWI &4001C)

Returns the name of the underlying transport to a given station

On entry

R0 = station number
R1 = net number

On exit

R0, R1 preserved
R2 = pointer to null terminated name of transport

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 name of the underlying transport to a given station. You can use this to insert the transport name into (for example) a status conversion.

Related SWIs

None

Related vectors

None

* Commands

The only * Command the Econet module responds to is *Help Station, which displays the current net and station numbers of the machine. It also displays a ‘No clock’ message if applicable. For more details of the *Help command, see page 1-980.

Application notes

The following code is the preferred way of testing for the presence of the Econet drivers. It calls the SWI Econet_SetProtection with R0 and R1 set such that the call attempts to read the Econet protection word; doing so can never return an unexpected error. Detecting the error 'No such SWI' allows software dependent upon Econet to report its absence by generating an error; note the use of MessageTrans to do so:

```
        STMFD   sp!, {r0-r7,lr}
        MOV     r0, #0
        MVN     r1, #0
        SWI     XEconet_SetProtection
        BVC     ExitFindEconet
        LDR     r2, ErrorNumber_NoSuchSWI
        LDR     r0, [r0]
        TEQ     r0, r2
        BNE     ExitFindEconet
        ADR     r0, Error_NoEconet
        MOV     r1, #0                  ; No message file - use global
        MOV     r2, #0                  ; No buffer - use internal one
        MOV     r3, #0
        MOV     r4, #0                  ; No parameters
        MOV     r5, #0
        MOV     r6, #0
        MOV     r7, #0
        SWI     XMessageTrans_ErrorLookup
ExitFindEconet
        STRVS   r0, [sp,#0]
        LDMFD   sp!, {r0-r7,pc}

ErrorNumber_NoSuchSWI
        DCD     &000001E6

Error_NoEconet
        DCD     &00000312
        DCB     "NoEco", 0
```

48 File server protocol interface

The user environment

Handles

A client is identified and authenticated to the file server by its station number and three *handles*. When a user logs on the file server creates these handles by opening directories; the handles identify to the file server the environment in which to interpret commands and to look up filenames presented by the client. The file server closes the handles when the user logs off again. The three handles which comprise the user environment are the currently selected directory or CSD (see page 2-12 and page 2-163), the user root directory or URD (see page 2-12 and page 2-194), and the library directory or Lib (see page 2-12 and page 2-172). Incidentally, the handles passed to the client are only used for the client/server communication, and are **not** the file server's own file handles for the directories.

Usually the client machine's software deals with the manipulation of these handles, but you can define your own environment by opening several directories and declaring a set of these handles as representing the current environment. Thus you can execute commands in a number of different environments.

Protocol Block Formats

Standard Tx Header

The initial protocol blocks that the client sends to the file server take a standard form. This form is known as the *standard Tx header*:

Byte	Meaning
1	reply port
2	function code
3	handle for user root directory (URD)
4	handle for currently selected directory (CSD)
5	handle for library directory (Lib)

The *reply port* is the Econet port on which the client station is prepared to receive a response from the file server. The *function code* indicates to the file server which operation to perform; for a list of available function codes, see the section entitled *File*

Server Function Codes on page 2-708. The three handles define the environment for the command, as described above. The command is sent to the file server on port &99, which is known as the *command port*.

Standard Rx Header

The responses that the file server returns to the client also take a standard form, known as the *standard Rx header*:

Byte	Meaning
1	command code
2	return code

The *command code* indicates to the client what action (if any) the client should take upon receiving this response. The command code is principally used when responding to a ‘Decode command line’ function (see page 2-710). The *return code* gives the status of the command passed to the file server:

- Zero indicates that the command step completed successfully
- Non-zero values are an error number indicating what error has occurred; the remainder of the message contains an ASCII string describing the error, which is terminated by a carriage return.

Standard Data Types

The file server protocols use standard data types for most operations.

Multi-byte values

In all cases multi-byte values are stored low byte first.

Object and user names

All specifications of object and user names may be 8-bit values, save that &80 is reserved since it is used to indicate the end of data in some file server protocols.

File size

File servers only support files up to 16 Megabytes, so all pointer operations and file length indications use 24 bit quantities (stored low byte first, of course). Econet protocols do not support larger files.

Attributes

The attributes for an object are stored in a single byte with the bit set to '1' meaning as follows:

Bit	Meaning if set
0	object has public read access
1	object has public write access
2	object has owner read access
3	object has owner write access
4	object is locked against owner deletion
5	object is a directory
6	object is a protected directory

Note that public lock is always implicitly set.

Date

The date is stored in two bytes thus:

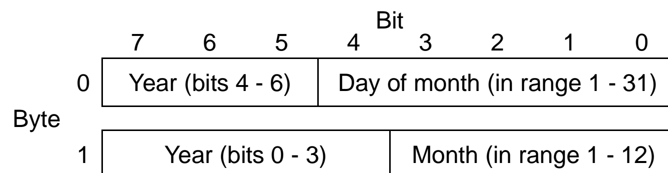


Figure 48.1 File server protocol date format

As you can see, the year is encoded as a seven bit number (ie 0 - 127); this is used to specify the offset from 1981.

Access rights

The access rights to a directory are encoded in a single byte:

Value	Meaning
0	owner access
&FF	public access

Object type

The object type is encoded in a single byte:

Value	Meaning
0	object not found
1	object is a file
2	object is a directory
3	object is an image file (ie both file and directory)

Privilege

A client may have different levels of privilege. The values for each privilege level are as follows:

Value	Meaning	Character equivalent
&00	locked	L
&40	fixed	F
&80	normal	
&FF	system manager	S

The character equivalents are used by some command line interfaces.

Object names

Object names are currently limited by the filing system, usually 10 characters. They may contain 8 bit values and are not case sensitive.

User identifiers

User identifiers are normally 10 characters with a provision for group identifiers ie: *group.name*, each of 10 characters.

Disc titles

The disc title is the name that the server exports to network clients. The length is determined by the host file system however it must start with a letter and consists of alphanumeric characters (ie: A-Z, a-z, 0-9), “-” and “_”.

Passwords

The password file associated with the server holds encrypted passwords, privilege levels, boot options and space allocation information for each user.

File Server Function Codes

A summary of the file server function codes is given below. *Function* is the function code number, *Privilege* shows whether the client has to have privilege, and *Logged-on* shows whether the client has to be logged-on:

Function	Privilege	Logged-on	Description
0	no	yes †	Decode command line
1	no	yes	Save file
2	no	yes	Load file
3	no	yes	Examine
4	no	yes	Catalogue header
5	no	yes	Load as
6	no	yes	Open object
7	no	yes	Close object
8	no	yes	Get byte
9	no	yes	Put byte
10	no	yes	Get bytes
11	no	yes	Put bytes
12	no	yes	Read random access arguments
13	no	yes	Set random access arguments
14	no	no	Read disc information
15	no	yes	Read current users' information
16	no	no	Read file server date and time
17	no	yes	Read 'End-of-file' status
18	no	yes	Read object information
19	no	yes	Set object attributes
20	no	yes	Delete object
21	no	yes	Read user environment
22	no	yes	Set user boot option
23	no	yes	Log off
24	no	yes	Read user's information
25	no	no	Read file server version number
26	no	yes	Read disc free space information
27	no	yes	Create directory, specifying size
28	yes	yes	Set file server date and time
29	no	yes	Create file
30	no	yes	Read user free space
31	yes	yes	Set user free space
32	no	yes	Read client UserId
33	no	yes	Read current users' info (extended)
34	no	yes	Read user's information (extended)
35			reserved
36	yes	yes	Manager interface
37			reserved

† There is no need to be logged-on to decode the *I Am command.

Interfaces

This section deals individually with each of the commands and functions available to client software. The exchange of packets is detailed and the format of requests and responses is given.

Decode command line

A number of the operations performed by the file server are initiated by the sending of a command line.

The command line syntaxes which the Acorn File Server will accept are as follows (commands in bold are new Level 4 commands):

```
*Access object_spec [attributes]
*Bye
*CDir directory
*Delete object
*Dir [directory]
*I am user_name [password]
*Info object_spec
*Lib [directory]
*Logon user_name [password]
*Pass old_password new_password
*Rename object new_name
*SDisc [:]disc_spec
```

Management specific commands:

```
*FSShutdown
*Logoff user_name|user_number
*NewUser user_name
*Priv user_name [new_privilege]
*RemUser user_name
```

The syntax of some of the above commands differ from the equivalent RISC OS commands, because the command line will already have been processed before the command is issued to the file server.

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 0)
6…	command line, terminated by CR

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3…	command dependent results

The possible command codes in the standard Rx header that the file server may return are:

Code	Meaning
0	no further action needed (ie command complete)
1	reserved
2	reserved
3	reserved
4	*Info
5	*I Am
6	*SDisc
7	*Dir
8	unrecognised command
9	*Lib
10	disc information, function code 14 called
11	users information, function code 15 called

Some commands require further action by the client, in which case the file server will also return (in byte 3 onwards) any decoded parameters or data which the client will need to complete the command:

Return from *Info

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header (command code = 4)
3 - 12	object name, padded with spaces
13	space
14 - 21	load address, padded with zeros
22	space
23 - 30	execution address, padded with zeros
31 - 33	spaces
34 - 39	length padded with zeros
40 - 42	spaces
43 - 48	access details (eg LWR/WR), padded with spaces
49 - 53	spaces
54 - 61	date (DD:MM:YY)
62	space
63 - 68	System Internal Name (SIN), padded with zeros
69	terminating negative byte (&80)

'Spaces' are ASCII spaces (&20). 'Zeros' are ASCII zeros (&30), **not** null bytes (&00).

Return from *I Am

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header (command code = 5)
3	new URD handle
4	new CSD handle
5	new Lib handle
6	boot option (bits 0 - 3 significant)

Return from *SDisc

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header (command code = 6)
3	new URD handle
4	new CSD handle
5	new Lib handle

Return from *Dir

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header (command code = 7)
3	new CSD handle

Return from unrecognised command

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header (command code = 8)
3…	command string, terminated by CR

Return from *Lib

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header (command code = 9)
3	new Lib handle

Save file

This is the actual save operation. This protocol is used after the command line has been decoded, either by the file server or the local operating system.

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1	reply port
2	1 (function code)
3	acknowledge port
4	CSD handle
5	Lib handle
6 - 9	file load address
10 - 13	file execute address
14 - 16	file size
17…	file name, terminated by CR

The file server's reply is sent to the client's reply port, as specified in the client's initial packet (see above):

Byte	Meaning
1 - 2	standard Rx header (command code = 3 if leaf name is returned, otherwise = 0)
3	data port
4 - 5	maximum data block size
6…	leaf name, terminated by CR (if returned)

The client and file server now enter the 'data transfer' phase of the protocol where the file server acknowledges the receipt of each data packet. If there is no data to be sent (eg a zero length file) then this phase is omitted. If the file server detects an error during the data transfer phase (eg a disc error) then the phase is allowed to complete, but the Save file operation is aborted, and the error status is given in the return code of the 'final acknowledge' (see below).

The client sends each block of data to the file server's data port, as specified in the file server's initial reply (see above):

Byte	Meaning
1…	a block of data, up to the maximum data block size

The file server acknowledges the receipt of each block by sending to the client's acknowledge port, as specified by the client in its initial transmission (see above):

Byte	Meaning
1	a single byte of undefined value

When the file server receives the final data block it instead acknowledges it with the 'final acknowledge', which is the terminating packet of the protocol. It sends this to the client's reply port, which – again – was specified in the client's initial packet:

Byte	Meaning
1 - 2	standard Rx header
3	attributes
4 - 5	date

Load file

This is the actual load operation. This protocol is used after the command line has been decoded, either by the file server or the local operating system.

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1	reply port
2	2 (function code)
3	data port
4	CSD handle
5	Lib handle
6…	file name, terminated by CR – may be wildcarded

The file server's reply is sent to the client's reply port, as specified in the client's initial packet (see above):

Byte	Meaning
1 - 2	standard Rx header (command code = 14 if leaf name is resolved, otherwise = 0)
3 - 6	file load address
7 - 10	file execute address
11 - 13	file size
14	file access
15 - 16	file creation date
17…	leaf name, terminated by CR, with wild-cards resolved (if returned)

The client and file server now enter the 'data transfer' phase of the protocol. If the file is of zero length then this phase is omitted. If the file server detects an error (eg a disc error) then the required amount of data will be sent, but its data content is undefined.

The file server sends each block of data to the client's data port, as specified in the client's initial packet:

Byte	Meaning
1…	data blocks of undefined size repeated until 'file size' data has been sent (maximum data block size is currently 4k)

The client does not acknowledge these packets.

When the file server has sent the final data block it then sends the terminating packet of the protocol to the client's reply port, which – again – was specified in the client's initial packet:

Byte	Meaning
1 - 2	standard Rx header

Examine

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 3)
6	argument to examine function: 0 ⇒ return all information, in machine readable format 1 ⇒ return all information, in character string format 2 ⇒ return file title only, in machine readable format 3 ⇒ return file title and access, in character string format
7	directory entry point (0 - 255)
8	number of entries to examine (1 - 255, or 0 for all)
9…	name of directory to be examined, terminated by CR

The argument passed in byte 6 specifies the format and amount of information to be returned by the file server.

The directory entry point gives the entry number within the directory from which to examine. Conventionally the first entry in a directory is entry number zero.

The number of entries to examine specifies how many entries are to be examined, so is usually determined by the buffer space available to the client. A parameter of zero in this case conventionally demands that all entries in the directory from the entry point to the end of the directory be examined.

Information may be returned in two ways: as a character string, or in a machine readable format:

- Information that is returned in character string format is in a fixed format – including separators – that is suitable for direct output. Carriage returns may occur within such strings. Individual directory entries are delimited by zero bytes (&00), the final entry being terminated by a negative byte (&80).
- Information that is returned in machine readable format consists of a defined number of bytes, and so there are no delimiters between entries, although the final entry is still terminated by a negative byte (&80).

The file server's reply is sent to the client's reply port, as specified in the client's initial packet (see above):

Byte	Meaning
1 - 2	standard Rx header
3	number of entries actually examined
4	number of entries in directory
5…	argument dependent information

The different formats of byte 5 onwards are given below.

Return for all information in machine readable format (argument = 0)

Byte	Meaning
5 - 14	object name padded to 10 characters with spaces
15 - 18	load address
19 - 22	execute address
23	attributes
24 - 25	date
26 - 28	System Internal Name (SIN)
29 - 31	object length

Return for all information in character string format (argument = 1)

Byte	Meaning
5…	character string of all information data (see above for character string format/separators)

Return for file title only, in machine readable format (argument = 2)

Byte	Meaning
5	10 (object name length for BBC MOS)
6…	object name padded with spaces

Return for file title and access, in character string format (argument = 3)

Byte	Meaning
5…	character string giving object name and formatted access string (see above for character string format/separators)

Catalogue Header

The client initiates the exchange by sending to the file server’s command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 4)
6…	directory name, terminated by CR (null name ⇒ catalogue CSD)

The file server’s reply is sent to the client’s reply port, as specified in the client’s initial packet (see above):

Byte	Meaning
1 - 2	standard Rx header
3 - 13	last component of directory name padded with spaces
14	character indicating ownership of directory (‘O’ or ‘P’)

15 - 17	three space characters (&20)
18 - 33	current disc name padded with spaces, terminated by CR, negative byte (&80)

Load as

This is exactly the same as Load file (Function code = 2), except that the file name is looked up in the CSD and then in the Lib. The error returned if the file name is not found in either directory is ‘Bad command'.

The protocol is identical, save that the client’s initial packet has a function code of 5 (for ‘Load as’) rather than 2 (for ‘Load file’).

Open object

This function code creates a handle for the object specified with the access type requested. Such handles are used for performing random access operations, and also for manipulating the user's environment. An object will be opened only if the client has the necessary access rights to the object. When opening directories these must be specified as already existing. A file can be opened several times for reading, but only once for update. A file will be created with default size of &400 bytes if it does not already exist, and is opened for update, and the client specifies a new file (byte 6 = 0). Machine-dependent limits are imposed on the number of handles a client is allowed to have open at any one time. BBC machines support 8, Master series and Archimedes clients are allowed 16. These values include 3 handles which are automatically allocated when the client logs on, therefore a BBC machine will be able to open a further 5 objects.

The client initiates the exchange by sending to the file server’s command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 6)
6	zero ⇒ create a new file, deleting existing data non-zero ⇒ object must already exist
7	zero ⇒ open object for update non-zero ⇒ open object for reading only
8…	object name, terminated by CR

The file server’s reply is sent to the client’s reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header (command code = 1 if leafname is returned, otherwise = 0)
3	object’s handle
4…	leaf name, terminated by CR (if returned)

Close object

This function indicates to the file server that the handle passed as argument is no longer needed and that all of the updated data in the file should be written out to the disc. A handle of zero indicates to the file server that all handles to open **files** are to be closed. This call does **not** close handles to directories if the handle given is zero.

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 7)
6	handle

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header

Get byte

The next four function codes deal with the facilities that the file server provides to enable the user to perform random access operations on open files.

These operations have an additional protocol to ensure the integrity of the data exchanged, provided by a *sequence number*. The sequence number is a single bit held in both client and file server which differentiates between:

- successive reads of a file using the pointer held in the file server
- repeated reads of the same byte, because the operation failed at the previous attempt.

The client sends the sequence number in the least significant bit of the flag byte of the Econet control block. The file server returns its copy of the sequence number with the data to allow the client to detect data sequencing errors. The client should invert its copy of the sequence number after every successful transaction with the file server. If the client detects a data packet with the incorrect sequence number, then the client should be prepared to repeat the request.

The *Get byte* function code reads a single byte from the file at the position specified by the file server's internal file pointer. The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1	reply port
2	8 (function code)
3	file handle

The file server’s reply is sent to the client’s reply port, as specified in the client’s initial packet (see above):

Byte	Meaning
1 - 2	standard Rx header
3	byte read (&FE if reading first byte after file end)
4	&00 ⇒ normal read operation &80 ⇒ last byte in the file &C0 ⇒ first byte after file end

Put byte

This function code writes a single byte to the file at the position specified by the file server's internal file pointer. The client initiates the exchange by sending to the file server’s command port:

Byte	Meaning
1	reply port
2	9 (function code)
3	file handle
4	byte to be written

The file server’s reply is sent to the client’s reply port, as specified in the client’s initial packet (see above):

Byte	Meaning
1 - 2	standard Rx header

Get bytes

This operation allows the client to read blocks of data. The client may supply an offset within the file at which to start the operation, or may use the sequential file pointer maintained by the file server. The protocol includes a sequence number as described for *Get byte* and *Put byte*. The client initiates the exchange by sending to the file server’s command port:

Byte	Meaning
1	reply port
2	10 (function code)
3	data port
4	CSD handle
5	Lib handle
6	file handle
7	zero ⇒ use supplied offset non-zero ⇒ use file server sequential pointer
8 - 10	number of bytes to transfer
11 - 13	file offset (ignored if byte 7 non-zero)

The file server's reply is sent to the client's reply port, as specified in the client's initial packet (see above):

Byte	Meaning
1 - 2	standard Rx header

The client and file server now enter the 'data transfer' phase of the protocol. If the transfer is of zero length then this phase is omitted. If the file server detects an error (eg a disc error) then the required amount of data will be sent, but its data content is undefined. If a read extends over the end of the file then the requested amount of data will be returned, but the data content of the bytes beyond the end of the file is undefined.

The file server sends each block of data to the client's data port, as specified in the client's initial packet:

Byte	Meaning
1…	data blocks of undefined size repeated until 'transfer size' data has been sent (maximum data block size is currently 4k)

The client does not acknowledge these packets.

When the file server has sent the final data block it then sends the terminating packet of the protocol to the client's reply port, which – again – was specified in the client's initial packet:

Byte	Meaning
1 - 2	standard Rx header
3	&00 ⇒ all OK &80 ⇒ read includes last byte in file
4 - 6	number of valid data bytes transferred

Put bytes

This operation allows the client to write blocks of data. The client may supply an offset within the file at which to start the operation, or may use the sequential file pointer maintained by the file server. The protocol includes a sequence number as described for *Get byte* and *Put byte*. The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1	reply port
2	11 (function code)
3	acknowledge port
4	CSD handle
5	Lib handle
6	file handle
7	zero ⇒ use supplied offset non-zero ⇒ use file server sequential pointer
8 - 10	number of bytes to transfer
11 - 13	file offset (if supplied)

The file server's reply is sent to the client's reply port, as specified in the client's initial packet (see above):

Byte	Meaning
1 - 2	standard Rx header
3	data port
4 - 5	maximum data block size

The client and file server now enter the 'data transfer' phase of the protocol where the file server acknowledges the receipt of each data packet. If there is no data to be sent then this phase is omitted. If the file server detects an error during the data transfer phase (eg a disc error) then the phase is allowed to complete, but the operation is aborted, and the error status is returned in the return code of the 'final acknowledge' (see below).

The client sends each block of data to the file server's data port, as specified in the file server's initial reply (see above):

Byte	Meaning
1…	a block of data, up to the maximum data block size

The file server acknowledges the receipt of each block by sending to the client's acknowledge port, as specified by the client in its initial transmission (see above):

Byte	Meaning
1	a single byte of undefined value

When the file server receives the final data block it instead acknowledges it with the ‘final acknowledge’, which is the terminating packet of the protocol. It sends this to the client’s reply port, which – again – was specified in the client’s initial packet:

Byte	Meaning
1 - 2	standard Rx header
3	undefined
4 - 6	number of valid data bytes transferred

Read random access information

This function code allows the client to discover information about files for which he currently has handles. The client initiates the exchange by sending to the file server’s command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 12)
6	file handle
7	0 ⇒ read sequential file pointer
	1 ⇒ read file extent (the amount of valid data)
	2 ⇒ read file size (the space allocated for the file)

The file server’s reply is sent to the client’s reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3 - 5	information requested

Set random access information

The client initiates the exchange by sending to the file server’s command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 13)
6	file handle
7	0 ⇒ set sequential file pointer
	1 ⇒ set file extent
8 - 10	value to set

The file server’s reply is sent to the client’s reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header

Read disc information

This function returns the disc configuration of the file server. Conventionally the file server's drives are logically numbered from zero upwards. However, this number is **not** the same as the drive number returned, which is the physical drive number. The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 14)
6	first logical drive number to interrogate
7	number of drives to interrogate (0 for all drives)

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3	number of drives found
4	physical drive number of first drive requested
5 - 20	disc name of first drive requested, padded with spaces
21…	further 17-byte entries in same format

Read current users' information

This function returns the currently logged on users of the file server, their station numbers and associated privileges. Conventionally the logged on user entries are numbered from zero. The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 15)
6	first entry for which to get information
7	number of entries for which to get information (0 for all)

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning	
1 - 2	standard Rx header	
3	number of entries returned	
4	station number of first user	} repeated for each entry returned
5	network number of first user	
6…	name of first user, terminated by CR	
n	privilege of first user	

Read file server date and time

It is not necessary to be logged on to the file server to use this function code. The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 16)

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3 - 4	date
5	hours (0 - 23)
6	minutes (0 - 59)
7	seconds (0 - 59)

Read 'End-of-file' status

This function is valid for file handles only. The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 17)
6	file handle

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3	zero ⇒ pointer within file non-zero ⇒ pointer outside file

Read object information

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 18)
6	1 ⇒ read object creation date
	2 ⇒ read load and execute address
	3 ⇒ read object length
	4 ⇒ read object attributes and access rights
	5 ⇒ read all object information
	6 ⇒ read access rights and cycle number of directory
	7 ⇒ read unique identifier
7…	object name, terminated by CR

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705). The reply's contents depend on the argument passed with the call:

Reply to arguments 1 - 5

Byte	Meaning
1 - 2	standard Rx header
3	object type
4…	requested results only, returned in the following order: load address (4 bytes), execute address (4 bytes), length (3 bytes), attributes (1 byte), date (2 bytes), access rights (1 byte)

Reply to argument 6

Byte	Meaning
1 - 2	standard Rx header
3	undefined
4	0
5	10 (length of directory name)
6 - 15	directory name padded with spaces
16	access rights
17	number of entries in directory

Reply to argument 7

Byte	Meaning	
1 - 2	standard Rx header	
3	object type	
4 - 9	unique identifier for that object on that server (SIN + disc number):	
	bits 0 - 23	file system System Identification Number (SIN)
	bits 24 - 31	file server disc number
	bits 32 - 47	filing system number

Set object attributes

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 19)
6	1 ⇒ set load address, execute address, and attributes
	2 ⇒ set load address
	3 ⇒ set execute address
	4 ⇒ set attributes
	5 ⇒ set creation date
7…	parameters to set (depend on byte 6)
n…	file name, terminated by CR

The lengths of the parameters to set are the same as the lengths of the parameters returned by *Read object information*: see page 2-726.

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header

Delete object

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 20)
6…	object name, terminated by CR

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3 - 6	load address
7 - 10	execute address
11 - 13	file length
14	file attributes

Read user environment

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 21)

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3	16 (length of disc name)
4 - 19	name of currently selected disc, padded with spaces
20 - 29	name of CSD, padded with spaces
30 - 39	name of Lib, padded with spaces

Set user boot option

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 22)
6	new boot option (bits 0 - 3 significant)

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header

Log off

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 23)

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header

Read single user's information

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 24)
6…	user name, terminated by CR

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3	user's privilege
4	user's station number
5	user's network number

Read file server version number

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 25)

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3 - 11	a text string describing the file server type
12	a space character (ASCII &20)
13 - 16	a text string of the form *n.xy* which is the version

Read disc free space

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 26)
6…	disc name, terminated by CR

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3 - 5	free space on disc (in sectors of &100 bytes)
6 - 8	disc size (in sectors of &100 bytes)

Create directory, specifying size

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 27)
6	maximum number of sectors to allocate
7…	name of directory, terminated by CR

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header

Set file server date and time

It is necessary to be logged on to the file server, with privilege, to set the date and time parameters.

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 28)
6 - 7	date
8	hours (0 - 23)
9	minutes (0 - 59)
10	seconds (0 - 59)

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header

Create file

This function creates a file of the size and type specified. The contents will automatically be set to zeros for security reasons.

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 29)
6 - 9	new file's load address
10 - 13	new file's execute address
14 - 16	new file's length
17…	new file's name, terminated by CR

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3	new file's attributes
4 - 5	new file's creation date

Read user free space

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 30)
6…	UserId for free space reading, terminated by CR; CR alone means return the free space of the client

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3 - 6	available space for UserId, in bytes

Set user free space

This function code is only legal for privileged users. The UserId specified is that of the client whose space allocation is to be amended.

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 31)
6 - 9	new value for available space, in bytes
10…	UserId, terminated by CR

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header

Read client UserId

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 32)

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3…	UserId of client, terminated by CR

Read current users' information (extended)

This function returns the currently logged on users of the file server, their station numbers and associated privileges. Conventionally the logged on user entries are numbered from zero. The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 33)
6	first entry for which to get information
7	number of entries for which to get information (0 for all)

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning	
1 - 2	standard Rx header	
3	number of entries returned	
4	station number of first user	repeated for each entry returned
5	network number of first user	
6	task number of first user	
7…	name of first user, terminated by CR	
n	privilege of first user	

Read single user's information (extended)

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header (function code = 34)
6…	user name, terminated by CR

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3	user's privilege
4	user's station number
5	user's network number
6	user's task number

Manager Interface

This function allows the system manager to manipulate all the details concerning the users of the system. The password file will then be updated accordingly.

You must be a privileged user to use this function.

The client initiates the exchange by sending to the file server's command port:

Byte	Meaning
1 - 5	standard Tx header, Function code = 36 (&24)
6	argument: 0 ⇒ read number of entries in password file 1 ⇒ read entry from password file 2 ⇒ write user profile in password file 3 ⇒ add new user 4 ⇒ remove user 5 ⇒ set privilege 6 ⇒ logoff user 7 ⇒ shutdown server
7…	argument dependent parameters (see below)

The argument passed in byte 6 specifies the function to be performed by the file server. Some of these functions require further parameters, which are given in byte 7 onwards of this initial protocol block. These are detailed below.

Read number of entries in password file (argument = 0)

No argument dependent parameters are passed with the initial protocol block.

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3 - 6	number of users

Read entry from password file (argument = 1)

The argument dependent parameters passed with the initial protocol block are:

Byte	Meaning
7 - 8	user number for which to get information

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header
3 - 6	user profile index
7	privilege
8	boot option (bits 0 - 3 significant)
9 - 12	spaces
13	station (if client logged on)
14	net (if client logged on)
15	allowed to log on flag (bit 1 significant)
16 - 37	user name, terminated by CR
38 - 60	password, terminated by CR
61…	URD name, terminated by CR

Write user profile in password file (argument = 2)

The argument dependent parameters passed with the initial protocol block are:

Byte	Meaning
7 - 10	reserved (must be zero)
11	privilege
12	boot option (bits 0 - 3 significant)
13 - 18	spaces
19	allowed to log on flag (bit 1 significant)
20 - 41	user name, terminated by CR
42 - 64	password, terminated by CR
65…	URD name, terminated by CR

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header

Add new user (argument = 3)

The argument dependent parameters passed with the initial protocol block are:

Byte	Meaning
7	user name, terminated by CR

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header

Remove user (argument = 4)

The argument dependent parameters passed with the initial protocol block are:

Byte	Meaning
7	user name, terminated by CR

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header

Set privilege (argument = 5)

The argument dependent parameters passed with the initial protocol block are:

Byte	Meaning
7	user name, terminated by CR
n…	new privilege ('S', 'L' or 'F'; or null for normal), terminated by CR

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header

Logoff user (argument = 6)

The argument dependent parameters passed with the initial protocol block are:

Byte	Meaning
7	user name, terminated by CR

The file server's reply is sent to the client's reply port, as specified by the client in its standard Tx header (see page 2-705):

Byte	Meaning
1 - 2	standard Rx header

Shutdown server (argument = 7)

No argument dependent parameters are passed with the initial protocol block.

Error messages

The server responds with errors under certain circumstances. The errors generated are as follows:

Network reported errors

Error string	Error Number
Insufficient space	&5C
Too much data	&83
Bad privilege letter	&8C
Bad user name	&AC
Bad rename	&B0
Already a user	&B1
Directory full	&B3
Is a directory	&B5
Too many users	&B8
Password must be between 6 and 22 characters	**&B9**
Insufficient privilege	&BA
Wrong password	&BB
User not known	&BC
Access violation	&BD
Insufficient access	&BD
Is a file	&BD
Who are you?	&BF
Too many open files	&C0
Already open	&C2
Disc full	&C6
Bad name	&CC
Bad directory name	&CC
Bad drive	&CD
Invalid access string	&CF
Not found	&D6
File not found	&D6
Channel	&DE
Sorry, not supported	**&FD**
Bad command	&FE
Server not available	**&FF**
Server has shut down	**&FF**
No more receive buffers	**&FF**
Failed to create user profile	**&FF**
Server internal error, please report to system manager	**&FF**

Errors in bold are new Level 4 errors.

Internal errors

Password file not found
Unable to open password file
No devices found, unable to start …
Unable to open/find choices file
Unable to find floppy disc
Unable to mount …
Unable to execute …
Error in exports file, unable to start
Unable to find exports file

49 The Broadcast Loader

Introduction and Overview

The Broadcast Loader enables files to be effectively broadcast to multiple clients, effectively increasing Econet transport throughput. It works in the following way:

When a client requests a file from a file server, it first broadcasts a request onto the network to ask if any other clients are loading the same file. If no other client is loading it, then it proceeds to load the file itself from the file server as normal. If during the loading process other clients ask for the same file, then they are acknowledged by the first client, and they wait for the first client to finish loading the file after which it then broadcasts the file to all the waiting clients.

This module is not supplied as a standard part of RISC OS 2, but will run under it, and is available as a separate product.

Performance

The Broadcast Loader greatly reduces the time taken to load the same file or application to a number of users. To a first approximation, the performance of a system using the Broadcast Loader to load a long file to n Clients will be $2 \times$ (time to load single copy) as opposed to $n \times$ (time to load single copy).

FileSwitch call interception

The Broadcast Loader works by intercepting some FileSwitch calls to NetFSEntry_File and dealing with them as appropriate. This is done using the SWI OS_FSControl (13) to return a pointer to the FileSwitch copy of the NetFS filing system control block, that has been modified to be non-relocatable. The Broadcast Loader then modifies the data pointed to so that when FileSwitch despatches calls to NetFSEntry_File they are in fact despatched to the Broadcast Loader first.

File servers supported

All of the Acorn file servers – Level 2, Level 3, FileStore and Level 4 – as well as the SJ Research MDFS products, are compatible with the Broadcast Loader.

Retransmission and errors

Files are transmitted from the broadcast server to clients in chunks of approximately one thousand bytes with sequence numbers. If a client enters the transaction during the file transfer, or misses a packet due to transmission errors or other reasons, then requests for missing blocks are made and retransmissions made to complete the transaction. A system of timeouts and error messages is provided to ensure no lock-up or erroneous condition can occur.

50 BBC Econet

Introduction and Overview

The BBC Econet module provides emulation of certain obsolete OSBYTE and OSWORD calls used by old 6502-based BBC computers, thus making it easier for you to port code that uses these calls.

This module is provided solely to support old programs. You should not use these calls in any new programs you write.

Technical details

Summary of calls

The following calls are provided, which emulate the corresponding obsolete OSBYTE and OSWORD calls:

Call	Notes
OS_Byte 50	
OS_Byte 51	
OS_Byte 52	
OS_Word 16	All 8 sub-reason codes are emulated (Transmit, Peek, Poke, JSR, User Procedure Call, Machine type, Halt and Continue)
OS_Word 17	Both sub-reason codes are emulated (OpenRx and ReadRx)
OS_Word 19	Only these function codes are supported: 0 read file server number 1 write file server number 2 read printer server number 3 write printer server number 4 read protection mask 5 write protection mask 8 read local station number 12 read printer server name 13 set printer server name 15 read file server retry delay 16 set file server retry delay 17 translate net number
OS_Word 20	All 3 sub-reason codes are supported (Do File Server Operation, Notify, and Cause Remote Error)

Correspondence between old and new calls

All the above calls use exactly the same parameters as the corresponding obsolete OSBYTE and OSWORD calls. The table below shows the correspondence between the register used on the 6502 to pass a parameter, and the register used on the ARM to pass the same parameter:

6502 register	ARM register
A	R0 (bits 0-7)
X	R1 (bits 0-7)
Y	R2 (bits 0-7)

Bits 8-31 of the ARM registers are ignored.

For more information on any of the obsolete OSBYTE and OSWORD calls, see the *Econet Advanced User Guide*.

Implementation

The BBC Econet module claims the ByteV and WordV vectors. If it recognises an OS_Byte or OS_Word as one that it supports, it first checks the presence of the module(s) that it needs to emulate the call. (These are Econet, NetFS and/or NetPrint.) It then translates the OS_Byte or OS_Word call to appropriate SWI call(s) to these modules.

Restrictions

OS_Byte 50 (poll transmission) and OS_Byte 51 (poll receive block) may enable interrupts and hence should **not** be called from within interrupt handlers, service code or event routines. During these calls the processor may be put in USR mode with interrupts enabled; this allows CallBacks to occur.

51 Hourglass

Introduction and Overview

The Hourglass module will change the pointer shape to that of an hourglass. You can optionally also display:

- a percentage figure
- two ‘LED’ indicators for status information (one above the hourglass, and one below).

Note that cursor shapes 3 and 4 are used (and hence corrupted) by the hourglass. You should not use these shapes in your programs.

Normally the Hourglass module is used to display an hourglass on the screen whenever there is prolonged activity on the Econet. The calls to do so are made by the NetStatus module, which claims the EconetV vector. See the chapter entitled *Software vectors* on page 1-63 and the chapter entitled *NetStatus* on page 2-759 for further details.

The hourglass should also be used by any software that may take some time to do a particular job, especially when:

- there is no other indication of activity
- the processing time is file size dependent (some users may have files much bigger than you expect)
- the processing time is processor speed dependent (some users may be in a screen mode that is hungry for memory bandwidth).

Software using the hourglass should, whenever possible, use the percentage feature; see the section entitled *Example programs* on page 2-757 for an example of this.

The rest of this chapter details the SWIs used to control the hourglass.

SWI Calls

Hourglass_On (SWI &406C0)

Turns on the hourglass

On entry

—

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This turns on the hourglass. Although control returns immediately there is a delay of 1⁄3 of a second before the hourglass becomes visible. Thus you can bracket an operation by Hourglass_On/Hourglass_Off so that the hourglass will only be displayed if the operation takes longer than 1⁄3 of a second.

You can set a different delay using Hourglass_Start (page 2-751).

Hourglass_On’s are nestable. If the hourglass is already visible then a count is incremented and the hourglass will remain visible until an equivalent number of Hourglass_Off’s are done. The LEDs and percentage indicators remain unchanged.

The example below illustrates the use of bracketing calls to Hourglass_On / Hourglass_Off:

```
DoLoadAndProcess
        STMFD   sp!, { r0-r5, lr }
        MOV     r0, #OSFile_Load
        ADR     r2, Buffer
        MOV     r3, #0
        SWI     XOS_File
        BVS     ExitLoadAndProcess
        CMP     r4, #0
        BEQ     ExitLoadAndProcess
        SWI     XHourglass_On
        BVS     ExitLoadAndProcess
        ADR     r1, Buffer
ProcessLoop
        LDRB    r0, [ r1 ], #1
        BL      ProcessByte
        BVS     FinishProcess
        SUBS    r4, r4, #1
        BNE     ProcessLoop
FinishProcess
        SWI     XHourglass_Off
ExitLoadAndProcess
        STRVS   r0, [ sp, #0 ]
        LDMFD   sp!, { r0-r5, pc }
```

Related SWIs

Hourglass_Off (page 2-748), Hourglass_Start (page 2-751)

Related vectors

None

Hourglass_Off (SWI &406C1)

Turns off the hourglass

On entry

—

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 decreases the count of the number of times that the hourglass has been turned on. If this makes the count zero, it turns off the hourglass.

When the hourglass is removed the pointer number and colours are restored to those in use at the first Hourglass_On.

From RISC OS 3 onwards, the system also turns the percentage display off if leaving the level that turned it on, even if the hourglass itself is not turned off. See page 2-753 for an example of this.

Related SWIs

Hourglass_On (page 2-746), Hourglass_Smash (page 2-750)

Related vectors

None

Hourglass_Smash (SWI &406C2)

Turns off the hourglass immediately

On entry

—

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 turns off the hourglass immediately, taking no notice of the count of nested Hourglass_On's. If you use this call you must be sure neither you, nor anyone else, should be displaying an hourglass.

When the hourglass is removed the pointer number and colours are restored to those in use at the first Hourglass_On, except under RISC OS 2.

Related SWIs

Hourglass_Off (page 2-748)

Related vectors

None

Hourglass_Start (SWI &406C3)

Turns on the hourglass after a given delay

On entry

R0 = delay before start-up (in centiseconds), or 0 to suppress the hourglass

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 works in the same way as Hourglass_On, except you can specify your own start-up delay.

If you specify a delay of zero and the hourglass is currently off, then future Hourglass_On and Hourglass_Start calls have no effect. The condition is terminated by the matching Hourglass_Off, or by an Hourglass_Smash.

Related SWIs

Hourglass_On (page 2-746), Hourglass_Off (page 2-748)

Related vectors

None

Hourglass_Percentage (SWI &406C4)

Displays a percentage below the hourglass

On entry

R0 = percentage to display (if in range 0 - 99), else turns off percentage

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 controls the display of a percentage below the hourglass. If R0 is in the range 0 - 99 the value is displayed; if it is outside this range, the percentage display is turned off.

The default condition of an hourglass is not to display percentages.

For a full example of the use of Hourglass_Percentage, see the section entitled *Example programs* on page 2-757.

From RISC OS 3 onwards, lower levels of calls cannot alter the hourglass percentage once a higher level call is using it. Furthermore, Hourglass_Off automatically turns the percentage display off when leaving the level that turned it on, even if the hourglass itself is not turned off. For example:

```
SYS "Hourglass_On"
  SYS "Hourglass_On"
  SYS "Hourglass_Percentage",10         :REM sets to 10%
  SYS "Hourglass_Percentage",20         :REM sets to 20%
    SYS "Hourglass_On"
    SYS "Hourglass_Percentage",50       :REM DOESN'T set to 50%
    SYS "Hourglass_Off"
  SYS "Hourglass_Percentage",30         :REM sets to 30%
  SYS "Hourglass_Off"                   :REM turns off percentages
SYS "Hourglass_Off"                     :REM turns off hourglass
```

Related SWIs

None

Related vectors

None

Hourglass_LEDs (SWI &406C5)

Controls the display indicators above and below the hourglass

On entry

R0, R1 = values used to set LEDs' word

On exit

R0 = old value of LEDs' word

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 two display indicators above and below the hourglass, which can be used to display status information. These are controlled by bits 0 and 1 respectively of the LEDs' word. The indicator is on if the bit is set, and off if the bit is clear. The new value of the word is set as follows:

New value = (Old value AND R1) EOR R0

The default condition is all indicators off.

Related SWIs

None

Related vectors

None

Hourglass_Colours (SWI &406C6)

Sets the colours used to display the hourglass

On entry

R0 = new colour to use as colour 1 (&00BBGGRR, or –1 for no change)
R1 = new colour to use as colour 3 (&00BBGGRR, or –1 for no change)

On exit

R0 = old colour being used as colour 1
R1 = old colour being used as colour 3

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 colours used to display the hourglass. Alternatively you can use this call to read the current hourglass colours by passing parameters of –1.

The default colours are:

Colour 1	cyan
Colour 3	blue

This call is not available in RISC OS 2.

Related SWIs

None

Related vectors

None

Example programs

The examples below illustrate the use of Hourglass_Percentage.

```
DoLoadAndProcess
        STMFD   sp!, { r0-r5, lr }
        MOV     r0, #OSFile_Load
        ADR     r2, Buffer
        MOV     r3, #0
        SWI     XOS_File
        BVS     ExitLoadAndProcess
        CMP     r4, #0
        BEQ     ExitLoadAndProcess
        SWI     XHourglass_On
        BVS     ExitLoadAndProcess
        ADR     r1, Buffer
        MOV     r2, #0
        ;       Compute a constant, in R3, such that as the index
        ;       in R2 goes from 0 to the maximum value, in R4, the
        ;       result of (R2 * R3) DIV 2^24 goes from 0 to 100.
        ;       R3 = (100 * 2^24) DIV R4.
        MOV     r5, #100 :SHL: 24       ; So we get a percentage
        MOV     r14, r4                 ; R3 := R5 DIV R4
        CMP     r14, r5, LSR #1
DivisionLoop1
        MOVLS   r14, r14, LSL #1
        CMPLS   r14, r5, LSR #1
        BLS     DivisionLoop1
        MOV     r3, #0
DivisionLoop2
        CMP     r5, r14
        SUBCS   r5, r5, r14
        ADC     r3, r3, r3
        MOV     r14, r14, LSR #1
        CMP     r14, r4
        BCS     DivisionLoop2
        ;       R3 is now a simple constant
ProcessLoop
        MUL     r0, r2, r3
        MOV     r0, r0, ASR #24
        SWI     XHourglass_Percentage   ; Call with result
        LDRVCB  r0, [ r1 ], #1
        BLVC    ProcessByte             ; May also return V set
        BVS     InternalError
        ADD     r2, r2, #1              ; Move the index
        TEQ     r2, r4
        BNE     ProcessLoop
FinishProcess
```

```
        SWI     XHourglass_Off
ExitLoadAndProcess
        STRVS   r0, [ sp, #0 ]
        LDMFD   sp!, { r0-r5, pc }

InternalError
        MOV     r1, r0                  ; Preserve the actual error
        SWI     XHourglass_Off          ; Ignore possible error
        MOV     r0, r1                  ; Retore real error
        CMP     pc, #&80000000          ; Set V, to indicate an error
        B       ExitLoadAndProcess
```

Or in BBC BASIC V:

```
DEF PROCLoadAndProcess( Name$ )
  LOCAL Length%, Index%: LOCAL ERROR
  SYS "OS_File", 255, Name$, Buffer%, 0 TO ,,,, Length%
  IF Length%<>0 THEN
    SYS "Hourglass_On"
    ON ERROR LOCAL: RESTORE ERROR: SYS "Hourglass_Off": ERROR ERR, REPORT$
    FOR Index% = 0 TO Length%
      SYS "Hourglass_Percentage", (100 * Index%) DIV Length%
      PROCProcessByte( Buffer%?Index% )
    NEXT Index%
    SYS "Hourglass_Off"
  ENDIF
ENDPROC
```

52 NetStatus

Introduction and Overview

The NetStatus module controls the display of an hourglass on the screen whenever there is prolonged activity on the Econet.

It claims EconetV, and examines the reason for each call that is made to the vector. It in turn makes an appropriate call to the Hourglass module, so that the appearance of the Hourglass indicates the status of the net. The Hourglass has two 'LEDs', one on top and one on the bottom:

- if only the top LED is on, then your station is trying to receive
- if only the bottom LED is on, then your station is trying to transmit
- if both LEDs are on, then your station is waiting for a broadcast reply.

It also displays percentage figures (when it is able to do so meaningfully) which show the percentage of a transfer that has completed.

Technical Details

This table shows how NetStatus converts the reason codes for calls to EconetV (listed in the chapter entitled *Software vectors*) into the SWI calls that it makes to the Hourglass module:

Reason code	SWI call
NetFS_Start…	Hourglass_On
NetFS_Part…	Hourglass_Percentage
NetFS_Finish…	Hourglass_Off
NetFS_StartWait	Hourglass_LEDs (both on)
Econet_StartTransmission	Hourglass_LEDs (only top one on)
Econet_StartReception	Hourglass_LEDs (only bottom one on)
NetFS_FinishWait	Hourglass_LEDs (both off)
Econet_FinishTransmission	Hourglass_LEDs (both off)
Econet_FinishReception	Hourglass_LEDs (both off)

Versions of RISC OS after 2.0 also change the colour of the hourglass for Broadcast Load and Save calls (as made by the Broadcast Loader). The colours used are:

Type of call	Colours
Broadcast Load	Green/blue
Broadcast Save	Red/blue