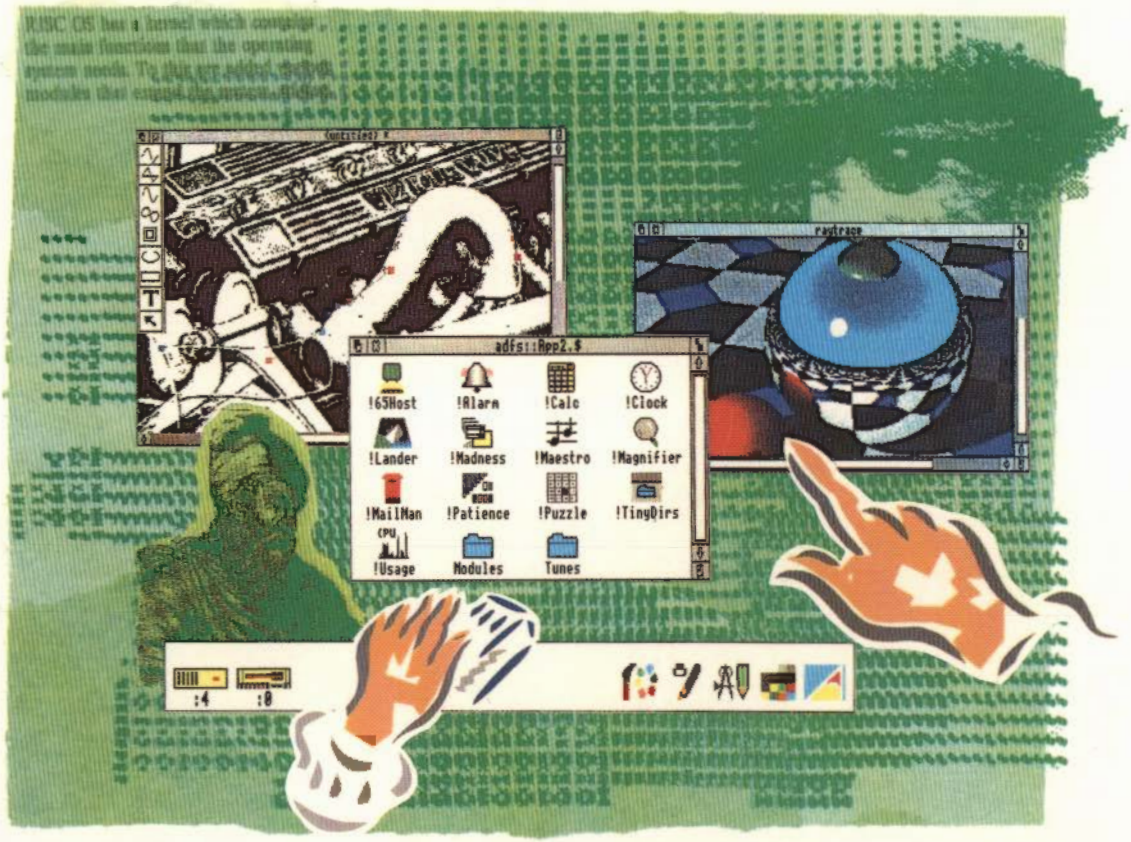
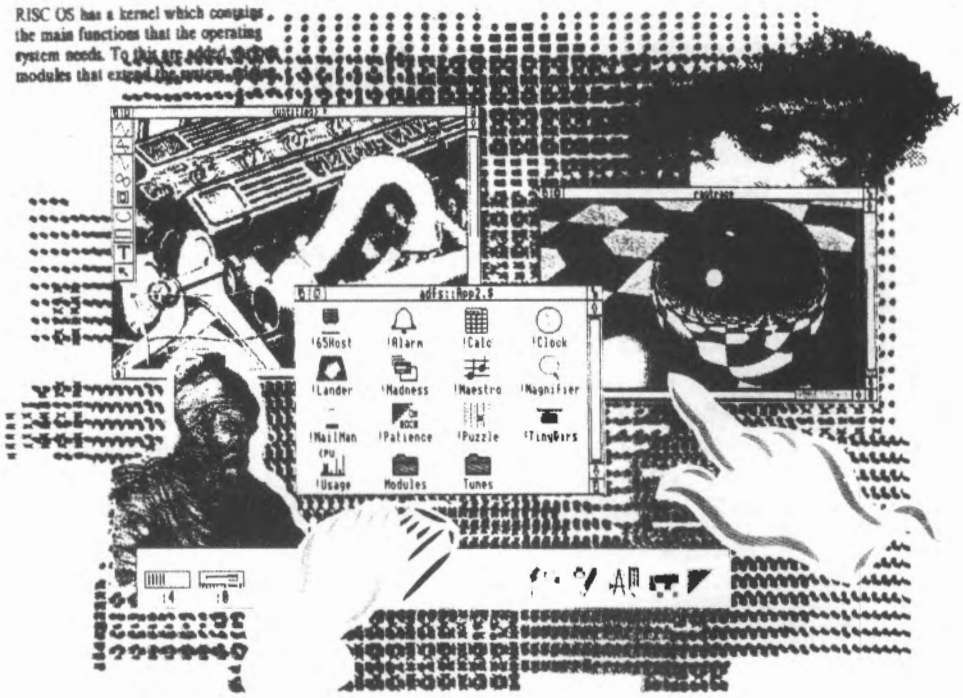
RISC OS
PROGRAMMER'S REFERENCE MANUAL
Volume III



Acorn
The choice of experience.

RISC OS
PROGRAMMER'S REFERENCE MANUAL
Volume III



Acorn
The choice of experience.

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

All correspondence should be addressed to:

Customer Service
Acorn Computers Limited
Fulbourn Road
Cambridge CB1 4JN

Information can also be obtained from the Acorn Support Information Database (SID). This is a direct dial viewdata system available to registered SID users. Initially, access SID on Cambridge (0223) 243642: this will allow you to inspect the system and use a response frame for registration.

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

Edition 1
Published 1989: Issue 1
ISBN 1 85250 062 X
Published by Acorn Computers Limited
Part number 0483,022

Contents

Part 3 - Filing systems

FileSwitch

Introduction

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

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

Other modules provide extra filing systems:

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

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.

Overview

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

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

Obviously, FileSwitch cannot know how to control every single piece of hardware that gets added to the system. The hardware is instead controlled by filing system modules that get added to the system. When these modules are initialised, 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.

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*; 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 *Application notes* at the end of this chapter.

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

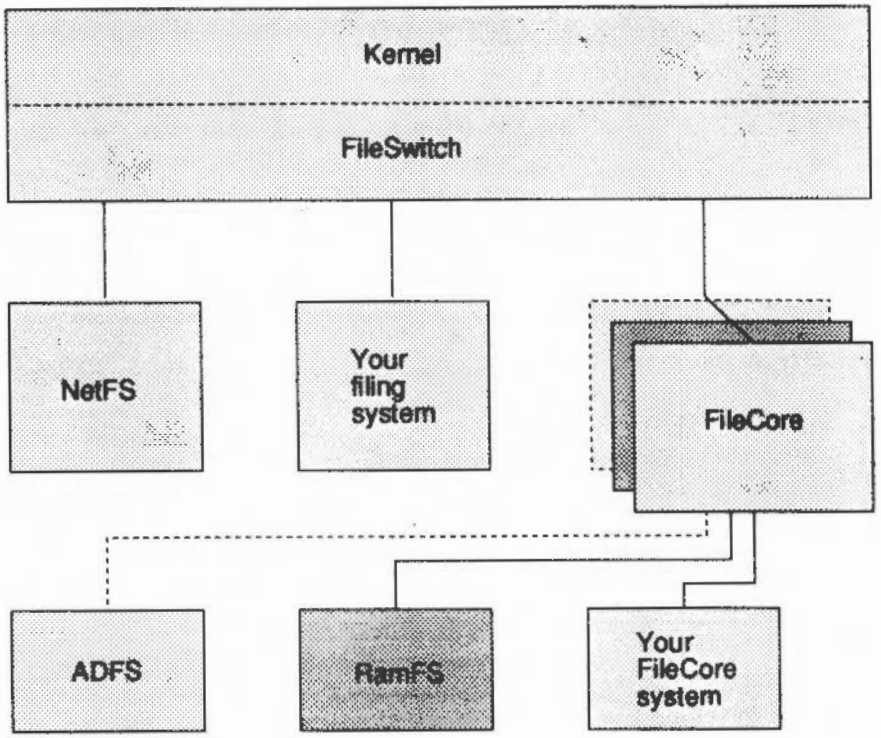
FileSwitch, it provides a common core of functions that are device independent, and it communicates with secondary modules that access the hardware.

Using FileCore to build part of your filing system imposes a more rigid structure on it, as more of the filing system is predefined. 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 next chapter entitled *FileCore*.

How filing systems are related

The diagram below shows how filing system modules are related in RISC OS:



Technical Details

Terminology

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

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

Filenames

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

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

Directories

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

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

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

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

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

Filing systems

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

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

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

Special fields

Special fields are used to supply more information to the filing system than you can using standard path names; for example NetFS and NetPrint use them 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.

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. Each filing system keeps an independent record of its own current selections, such as its CSD, CSL, PSD and URD.

File attributes

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

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

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 forexpansion, so any routines that update the attributes must not alter these bits, and any routines that read the attributes must not assume these bits are clear.

Addresses / File types and date stamps

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

Load and execution addresses

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

Load address	XXXLLLLL
Execution address	GGGGGGGG

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

XXXLLLLL ≤ GGGGGGGG < XXXLLLLL + Length of file

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

File types and date stamps

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

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

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

Load address	`FFFtttdd`
Execution address	`dddddddd`

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

The file types are split into three categories:

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

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

If you type:

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

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

Additional information

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

Load-time and run-time system variables

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

An example of LoadType

For example, suppose you type

```
*LOAD mySprites
```

where the type of the file `mySprites` is &FF9. FileSwitch will issue:

```
*@LoadType_FF9 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 mySprites
```

An example of RunType

Similarly, if you typed:

```
*Run BasicProg parm1 parm2
```

where `BasicProg` is in the library, and its file type is &FFB. FileSwitch would issue:

```
*@RunType_FFB %.BasicProg parm1 parm2
```

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 "%.BasProg" parm1 parm2
```

Default settings

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

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

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

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

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

File$Path and Run$Path

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

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

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

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

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

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

Default values

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

Specifying filing system names

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

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

This would look for:

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

Resulting filenames

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

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

then:

```
*Run $.mike
```

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

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

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

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

Avoiding using File$Path and Run$Path

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

Using other path variables

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

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

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

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

System devices

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

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

The NetPrint module also provides a system device:

netprint:	the network printer

For full details, see the chapters entitled *System devices* and *NetPrint*.

Re-entrancy

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

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

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

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

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

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

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

Inperrupt code

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

FileSwitch and the kernel

FileSwitch is assembled with the kernel, and may effectively be considered as part of it. 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 the chapters *Character output* and *Character input* respectively for details.

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 below 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 (SWI &09), OS_Find (SWI &0D)

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 in the next section on ** Commands*.

Related SWIs

OS_FSControl 10 (SWI &29)

Related vectors

ByteV

OS_Byte 255 (SWI &06)

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

On entry

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

On exit

R0 preserved
R1 = previous value
R2 corrupted

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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

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

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

Related SWIs

None

Related vectors

ByteV

OS_File (SWI &08)

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

On entry

R0 = reason code
Other registers depend on reason code

On exit

R0 corrupted
Other registers depend on reason code

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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

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

R0	Action
0	Saves a block of memory as a file
1	Writes catalogue information for a named object
2	Writes load address only for a named object
3	Writes execution address only for a named object
4	Writes attributes only for a named object
5	Reads catalogue information for a named object, using File$Path
6	Deletes a named object
7	Creates an empty file
8	Creates a directory
9	Writes date/time stamp of a named file
10	Saves a block of memory as a file, and date/time stamps it
11	Creates an empty file, and time/date stamps it

12	Loads a named file, using specified path string
13	Reads catalogue information for a named object, using specified path string
14	Loads a named file, using specified path variable
15	Reads catalogue information for a named object, using specified path variable
16	Loads a named file, using no path
17	Reads catalogue information for a named object, using no path
18	Sets file type of a named file
19	Generates an error message
255	Loads a named file, using File$Path

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

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

Non-destructive operations, such as loading a file and reading and writing attributes may have wildcards in the leafname. However, only the first file found (in ASCII order of file name) will be accessed by the operation.

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
R5 = end address in memory of data

On exit

Registers preserved

Use

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

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

See also OS_File 7 and 11 (which are documented together); 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.

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

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

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

Read catalogue information for a named object

On entry

R0 = 5, 13, 15 or 17
R1 = pointer to (wildcarded) object name
If R0 = 13
    R4 = pointer to control-character terminated path string
If R0 = 15
    R4 = pointer to path variable, containing control-character terminated 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	string pointed to by R4
15	variable pointed to by R4
17	none

On exit, R0 contains the object type:

R0	Type
0	Not found
1	File found
2	Directory found

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

OS_File 6 (SWI &08)

Deletes a named object

On entry

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

On exit

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

Use

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

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

NetFS behaves unusually in two ways:

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

OS_File 7 and 11 (SWI &08)

Creates an empty file

On entry

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

On exit

Registers preserved

Use

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

Note: No data is transferred. The file does not necessarily contain zeros; the contents may be completely random. Some security-minded systems (such as NetFS/FileStore) will deliberately overwrite any existing data in the file. Other filing systems (such as ADFS) do not, so you can recover data after accidental deletion by creating small empty files that fill the rest of the disc. These will between them contain the lost data.

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 (which are documented together); 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 path string
If R0 = 14
    R4 = pointer to path variable, containing control-character terminated path string

On exit

R0 = object type (always 1, 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	string pointed to by R4
14	variable pointed to by R4
16	none
255	File$Path system variable

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

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

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

OS_File 19 (SWI &08)

Generates an error message

On entry

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

On exit

R0 = pointer to error block
V flag set

Use

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

```
"File 'xyz' not found"
"'xyz' is a file"
"'xyz' is a directory"
```

An example of its use would be:

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

OS_Args (SWI &09)

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

On entry

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

On exit

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

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

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

Value	Action
0	Read pointer/FS number
1	Write pointer
2	Read extent
3	Write Extent
4	Read allocated size
5	Read EOF status
6	Reserve space
254	Read information on file handle
255	Ensure file/files

Related SWIs

None

Related vectors

ArgV

OS_Args 0 (SWI &09)

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

On entry

R0 = 0
R1 = 0 or file handle

On exit

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

Use

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

This call indirects through ArgV.

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 ArgV.

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 ArgV.

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 ArgV.

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 ArgV.

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 ArgV.

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 - R2 preserved

Use

This call ensures on open file's size.

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

This call indirects through ArgV.

OS_Args 254 (SWI &09)

Reads information on a file handle

On entry

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

On exit

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

Use

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

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

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

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

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.

For a full definition of the filing system information word returned in R2, see the application notes at the end of this chapter on adding your own filing system.

This call indirects through ArgV.

OS_Args 255 (SWI &09)

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

On entry

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

On exit

R0 - R2 preserved

Use

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

This call indirects through ArgV.

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 (SWI &0B), OS_GBPB (SWI &0C)

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 (SWI &0A), OS_GBPB (SWI &0C)

Related vectors

BPutV

OS_GBPB (SWI &0C)

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

On entry

R0 = reason code
Other registers depend on reason code

On exit

R0 preserved
Other registers depend on reason code

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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

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

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

Related SWIs

OS_BGet (SWI &0A), OS_BPut (SWI &0B)

Related vectors

GBPBV

OS_GBPB 1 and 2 (SWI &0C)

Write bytes to an open file

On entry

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

On exit

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

Use

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

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

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

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

OS_GBPB 3 and 4 (SWI &0C)

Read bytes from an open file

On entry

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

On exit

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

Use

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

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

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

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

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

Read information on a filing system

On entry

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

On exit

R0, R2 preserved
C flag corrupted

Use

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

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

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

  The boot option byte may contain values other than 0 - 3.

- If R0 = 6, the call reads the name of the currently selected directory, and privilege status in relation to that directory. It is returned as:

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

  The privilege byte is &00 if you have 'owner' status (ie can create and delete objects in the directory) or &FF if you have 'public' status (ie are prevented from creating and deleting objects in the directory). On ADFS and other FileCore-based filing systems you always have owner status.

- If R0 = 7, the call reads the name of the library directory, and privilege status in relation to that directory. It is returned as:

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

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

OS_GBPB 8 (SWI &0C)

Reads entries from the current directory

On entry

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

On exit

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

Use

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

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

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

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

On exit, R4 contains the value which should be used on the next call (to read more names), or -1 if there are no more names after the ones read by this call. There is no guarantee that the number of objects you asked for will be

read. This is because of the external constraints some filing systems may impose. To ensure reading all the entries you want to, this call should be repeated until R4 = -1.

This call is only provided for compatibility with older programs.

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

Read entries and file information from a specified directory

On entry

R0 = 9, 10 or 11
R1 = pointer to directory name (control-character or null terminated)
R2 = start address of data in memory (word aligned if R0 = 10 or 11)
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 next item to 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 or 11 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.

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_Find (SWI &0D)

Opens and closes files

On entry

R0 = reason code
Other registers depend on reason code

On exit

Depends on reason code

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call opens and closes files.

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

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

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

When you open a file a unique file handle is returned to you. You need this for any calls you make to OS_Args (SWI &09), OS_BGet (SWI &0A), OS_BPut (SWI &0B) and OS_GBPB (SWI &0C), and to eventually close the file using OS_Find 0.

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

Related SWIs

None

Related vectors

FindV

OS_Find 0 (SWI &0D)

Closes files

On entry

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

On exit

Registers preserved

Use

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

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

Otherwise R1 must contain a file handle, that was returned by the earlier call of OS_Find that opened the file.

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 obect doesn't exist
R1 and R2 preserved

Use

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

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

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

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

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

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

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

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

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

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

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

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

Bit 0	Bit 1	Path used
0	0	File$Path system variable
0	1	string pointed to by R2
1	0	variable pointed to by R2
1	1	none

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 (SWI &09), OS_BGet (SWI &0A), OS_BPut (SWI &0B) and OS_GBPB (SWI &0C), and to eventually close the file using OS_Find 0.

OS_FSControl (SWI &29)

Controls the filing system manager and filing systems

On entry

R0 = reason code
Other registers depend on reason code

On exit

R0 preserved
Other registers depend on reason code

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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

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

R0	Action
0	Set the current directory
1	Set the library directory
2	Inform of start of new application
3	Reserved for internal use
4	Run a file
5	Catalogue a directory
6	Examine the current directory
7	Catalogue the library directory
8	Examine the library directory
9	Examine objects
10	Set filing system options
11	Set the temporary filing system from a named prefix
12	Add a filing system
13	Check for the presence of a filing system
14	Filing system selection

15	Boot from a filing system
16	Filing system removal
17	Add a secondary module
18	Decode file type into text
19	Restore the current filing system
20	Read location of temporary filing system
21	Return a filing system file handle
22	Close all open files
23	Shutdown filing systems
24	Set the attributes of objects
25	Rename objects
26	Copy objects
27	Wipe objects
28	Count objects
29	Reserved for internal use
30	Reserved for internal use
31	Convert a string giving a file type to a number
32	Output a list of object names and information
33	Convert a file system number to a file system name

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

Related SWIs

None

Related vectors

FSControlV

OS_FSControl 0 (SWI &29)

Set the current directory and (optionally) filing system

On entry

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

On exit

Registers preserved

Use

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

OS_FSControl 1 (SWI &29)

Set the library directory

On entry

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

On exit

Registers preserved

Use

This call sets the current library directory to the named one, which may be on a filing system other than the currently selected one. If the name pointed to is null, the library directory is set to the filing system default (typically $.Library, if present).

If a library is not set on FileCore-based filing systems they search in order &.Library, $.Library and @, instead of %.

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 255, 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, 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 – 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_NewApplication).

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).

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).

FileSwitch frees any temporary strings and transient blocks it has allocated and sets the temporary filing syste. 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     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 (SWI &10).
- 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 (SWI &10).
- 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 (SWI &10). 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* 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 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 (SWI &05) 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.

OS_FSControl 12 (SWI &29)

Add a filing system

On entry

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

On exit

Registers preserved

Use

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

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

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

For full information on writing a filing system module, see the section entitled *Application Notes* at the end of this chapter.

OS_FSControl 13 (SWI &29)

Check for the presence of a filing system

On entry

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

On exit

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

Use

This call checks for the presence of a filing system.

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

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

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

OS_FSControl 14 (SWI &29)

Filing system selection

On entry

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

On exit

Registers preserved

Use

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

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

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

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

OS_FSControl 15 (SWI &29)

Boot from a filing system

On entry

R0 = 15

On exit

R0 preserved

Use

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

This call may not return if boot runs an application.

For more details, see *Configure Boot and NoBoot, and the *Opt commands.

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 does not call the filing system to close open files, flush buffers, and so on. 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.

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).

If the filing system is the temporary one (but not the current one) then the temporary filing system is set to the current filing system.

R1 must be a pointer to a control-character terminated name – you cannot remove a filing system by 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.

OS_FSControl 17 (SWI &29)

Add a secondary module

On entry

R0 = 17
R1 = filing system name
R2 = 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 66 (SWI &30). 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 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 uses OS_FSControl 11 to set a temporary filing system before a command; it uses this call to restore the current filing system afterwards. This command is also called by the kernel before it calls the error handler.

OS_FSControl 20 (SWI &29)

Read location of temporary filing system

On entry

R0 = 20

On exit

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

Use

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

OS_FSControl 21 (SWI &29)

Return a filing system file handle

On entry

R0 = 21
R1 = file handle

On exit

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

Use

This call takes a file handle used by FileSwitch, and returns the internal file handle used by the filing system which it belongs to. It also returns a filing system information word. For a full definition of this, see the section entitled *Application Notes* at the end of this chapter on adding your own filing system.

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

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

OS_FSControl 22 (SWI &29)

Close all open files

On entry

R0 = 22

On exit

R0 preserved

Use

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

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

OS_FSControl 23 (SWI &29)

Shutdown filing systems

On entry

R0 = 23

On exit

R0 preserved

Use

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

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

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

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

OS_FSControl 24 (SWI &29)

Set the attributes of objects

On entry

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

On exit

Registers preserved

Use

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

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

OS_FSControl 25 (SWI &29)

Rename objects

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).

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 an object, 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 startup, then the copy is aborted and an error is generated to the Error handler of the parent of the task that called OS_FSControl 26. The call does not return; it sets the environment time variable to the time read when the copy started and issues SWI OS_Exit, setting Sys$ReturnCode to 0.

OS_FSControl 27 (SWI &29)

Wipe objects

On entry

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

On exit

Registers preserved

Use

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

OS_FSControl 28 (SWI &29)

Count objects

On entry

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

On exit

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

Use

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

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

OS_FSControl 31 (SWI &29)

Converts a string giving a file type to a number

On entry

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

On exit

R0, R1 preserved
R2 = filetype

Use

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

```
Obey
Text
```

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

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

OS_FSControl 18 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.

OS_FSControl 33 (SWI &29)

Converts a filing system number to a filing system name

On entry

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

On exit

Registers preserved

Use

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

*Access

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

Syntax

```
*Access <object spec> [<attributes>]
```

Parameters

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

Use

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

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

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

Examples

```
*access myfile l

*access myfile wr/r
```

Related commands

*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 (SWI &27 – see the chapter entitled *Conversions*) before it is added to the file. An Escape finishes the input:

Example

```
*type thisfile
this line is already in thisfile
*append thisfile
  1 some more text
<Press Esc>
*type thisfile
this line is already in thisfile
some more text
```

Related commands

*Build

*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 (SWI &27 – see the chapter entitled *Conversions*) before it is added to the file. An 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
<Press Esc>
*Type testfile
This is the first line of testfile
```

Related commands

*Append

*Cat

Lists all the objects in a directory.

Syntax

```
*Cat [<directory>]
```

Parameters

`<directory>`	a valid pathname specifying a directory

Use

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

Examples

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

Related commands

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

*CDir

Creates a directory.

Syntax

```
*CDir <directory> [<size in entries>]
```

Parameters

`<directory>`	a valid pathname specifying a directory
`<size in entries>`	how many entries the directory should hold before it needs to be expanded (NetFS only)

Use

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

Examples

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

Related commands

*Cat

*Close

Closes files.

Syntax

```
*Close
```

Parameters

None

Use

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

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

```
*adfs:Close
```

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

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

Related commands

*Shut, *Shutdown, *Bye

*Configure Boot

Causes a power on, reset or Ctrl Break to perform a boot.

Syntax

```
*Configure Boot
```

Use

The configured boot action is read at power-on or hard reset. *Configure Boot causes any kind of reset to perform a boot, 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 dealt with in the way 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.

This option can also be set from the desktop, using the Configure application.

Related commands

*Opt 4, *Configure NoBoot, *FX 255

*Configure DumpFormat

Selects the layout for *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	<n> is used, where n is in decimal
3	<&n> is used, where n is in hexadecimal

Characters which have their top bit set are normally treated as control codes. However, if n has a value between 4 and 7 (ie if bit 2 is set) they are treated as printable characters. n=5, for example, causes ASCII character 247 to be printed as + (Latin fonts only).

If bit 3 is set (ie for values of n between 8 and 15), characters are ANDed with &7F before being processed, (ie converted to ordinary characters) otherwise they are left as they are.

Use

*Configure DumpFormat <n> selects the format to be used by the *Dump, *List and *Type commands, and the vdu: output device. The command has immediate effect. The default value is 4 (GSTrans format, characters with the top bit set are printed, all 8 bits considered).

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

Example

```
*Configure DumpFormat 2
```

Related commands

*Dump, *List

*Configure FileSystem

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

Syntax

```
*Configure FileSystem <n> | <name>
```

Parameters

`<n>`	filing system number (8 for ADFS, 5 for NetFS)
`<name>`	filing system name (ADFS, Net or Ram)

Use

*Configure FileSystem selects the filing system to be used at power on or hard reset. This is done just before any boot action is taken. A banner is displayed showing what filing system is in use. (The banner is also shown on a soft reset.)

If you use a filing system name then it must be registered with FileSwitch at the time, so it can be converted to the filing system number that is actually stored.

If the filing system is not found 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

Causes a Shift power on, Shift reset or Shift Break to run a boot file.

Syntax

```
*Configure NoBoot
```

Use

The configured boot action is read at power-on or hard reset. *Configure NoBoot causes any kind of reset not to perform a boot – 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 dealt with in the way 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.

This option can also be set from the desktop, using the Configure application.

Related commands

*OPT 4, *Configure Boot, *FX 255

*Copy

Copies files and directories.

Syntax

```
*Copy <source spec> <destination spec> [[~]<options>]
```

Parameters

`<source spec>`	file pathname
`<destination spec>`	file pathname
`<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 (to copy the file or structure and then proceed to the next item), N (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.

In the Desktop copying can use the Wimp's free memory, and so you should not need to use this option. It's quicker not to use this option when you are copying from hard disc to floppy, as these operations are interleaved so well. However, in other circumstances this option can speed up the copying operation considerably.

- R(ecurse) Copy subdirectories and contents.
  Default OFF.

  This is useful when copying several levels of directory, since it avoids the need to copy each of the directories one by one.

- S(tamp) Restamp date-stamped files after copying.
  Default OFF.

  Useful for recording when the particular copy was made.

- (s)T(ructure) Copy only the directory structure.
  Default OFF.

  Copies the directory structure but not the files. By using this option as a first stage in copying a directory tree, access to the files is faster when they are subsequently copied.

- V(erbose) Print information on each object copied.
  Default ON.

  This gives full textual commentary on the copy operation.

Use

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

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

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

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

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

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

Examples

```
*Copy fromfile tofile rfq~c~v
```

```
*Copy :fred.data* :jim.*
```

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

Related commands

*Rename, *Delete, *Access, *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 <file spec> [<options>]
```

Parameters

`<file spec>`

`<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 '~' (e.g: ~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, rather than printing the subtotal counted in directories.
  Default OFF.

Use

*Count adds up the size of data held in one or more files that match the given 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

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

*Create

Reserves space for a new file.

Syntax

```
*Create <filename> [<length> [<exec addr> [<load addr>]]]
```

Parameters

`<filename>`	a 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.

Example

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

Related commands

*Load, *Save

*Delete

Erases a single file or directory.

Syntax

```
*Delete <filename>
```

Parameters

`<filename>`	a pathname specifying a file

Use

*Delete erases the single named file. If the file does not exist, an error message is reported. Wildcards may be used in all the components of the pathname except the last one.

Examples

`*Delete $.dir*.myfile`	Note that wildcards are permitted in all of the pathname save for the leafname.
`*Delete myfile`	Deletes `myfile` from the current directory.

Related commands

*Wipe, *Remove

*Dir

Selects a directory.

Syntax

```
*Dir [<directory>]
```

Parameters

`<directory>`	a valid pathname specifying a directory

Use

*Dir sets 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

*CDir; *Back

*Dump

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

Syntax

```
*Dump <filename> [<file offset> [<start address>]]
```

Parameters

`<filename>`	a valid pathname specifying a file
`<file offset>`	offset, in hexadecimal, from the beginning of the file from which to dump the data
`<start address>`	(in hexadecimal) display as if the file were in memory starting at this address – 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 left hand side of:

`<Start address>` + current offest 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

*Type, *List, *Configure DumpFormat.

*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 filename
`<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 is *, which will match any file within the current 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 directories name and other useful information.

Examples

```
*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

*Info, *FileInfo

*Exec

Causes input to be taken from a file

Syntax

```
*Exec [<filename>]
```

Parameters

`<filename>`	a valid pathname specifying a file

Use

`*Exec <filename>` causes character input to be taken from the given file, rather than from the keyboard or serial input buffer. The file, once open, takes priority over the keyboard or serial port input streams.

The command is commonly used from a * prompt to execute a file of * Commands contained in a command file.

`*Exec` with no filename closes the exec file. You also close an existing exec file by the command `*Exec <new exec file>`.

If an exec file is open when the Desktop is started as the configured language, it detects this and exits to the supervisor, so that the supervisor can process the character input from the exec file.

However, if you use the command *Desktop this in turn issues the command *Exec, thus closing any open exec files. This is to make it easy for you to enter the Desktop from an exec boot file – simply put the *Desktop command at the end of the file.

Example

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

Related commands

*Obey

*FileInfo

Displays full file information.

Syntax

```
*FileInfo <object>
```

Parameters

`<object>`	a valid (wildcarded) pathname specifying a file

Use

*FileInfo gives the full file information of the specified object(s); this includes load, length and execution addresses in hexadecimal, and other filing system specific information not given by *Info. FileCore based filing systems (such as ADFS and RamFS) give information on all matches of the pathname, whereas some other filing systems (such as NetFS) only give information on the first match.

Example

```
*FileInfo current

Current    WR/    Text       15:54:37.40 04-Jan-1989 000007F 0002AA2E 007A3500
```

The above example shows the information displayed by *FileInfo on ADFS. On NetFS, a second date and time are also shown: these are the settings from the file server clock.

Related commands

*Ex, *Info

*Info

Displays information about specified objects.

Syntax

```
*Info <object spec>
```

Parameters

`<object spec>` a valid pathname specifying one or more file objects

Use

*Info lists file information for any object matching the given wildcard specification.

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>]
```

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

Use

*LCat lists all the objects in the named library subdirectory. The default is the current library. *LCat is equivalent to *Cat %.

Related commands

*LEx, *Cat

*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. This command 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 is filing system dependent: in ADFS the default is the object that would be referenced by the URD; under NetFS there is no default.

Example

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

Related commands

*Configure Lib, *NoLib

*List

Displays file contents with line numbers.

Syntax

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

Parameters

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

Use

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

Example

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

Related commands

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

*Load

Loads the named file (usually a data 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 Alais$@LoadType variable associated with this file

Use

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

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

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

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

Example

```
*Load myfile 9000
```

Related commands

*Create, *Save

*Opt 1

*Opt 1 controls filing system messages.

Syntax

*Opt 1 [[,] <n>]

Parameters

<n> 0 to 3

Use

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

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

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

*Opt 4

*Opt 4 sets the filing system boot action.

Syntax

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

Parameters

<n> 0 to 3

Use

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

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

The boot file is named &.!Boot on FileCore-based filing systems (such as ADFS and RamFS), and &.!ArmBoot on NetFS.

If you want such a boot file, and want to enter the desktop after executing it, the file should end with the command *Desktop, and similarly for other languages.

Example

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

Related commands

*Configure Boot and *Configure NoBoot

*Print

Displays raw text on the screen.

Syntax

```
*Print <filename>
```

Parameters

`<filename>` a valid pathname specifying a file

Use

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

Example

```
*Print myfile
```

Related commands

*Dump, *List, *Type

*Remove

Deletes a file.

Syntax

```
*Remove <filename>
```

Parameters

`<filename>` a valid pathname specifying a file

Use

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

Related commands

*Delete, *Wipe

*Rename

Changes the name of an object.

Syntax

```
*Rename <object> <new name>
```

Parameters

`<object>`	a valid pathname specifying a file or directory
`<new name>`	a valid pathname specifying a file or directory

Use

*Rename changes the name of an object, within the same storage unit. Locked objects cannot be renamed (unlock them first by using the *Access command with the Lock option clear). It can also be used for moving files from one directory to another, or moving directories within the directory tree.

Example

```
*Rename fred jim
```

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

Moves `fred` into directory `newdata`.

Related commands

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

*Run

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* for further details)

Use

*Run executes a file, together with a list of parameters, if appropriate. The given pathname is searched for in the directories listed in the system variable Run$Path. If a matching object is a directory then it is ignored, unless it contains a !Run file.

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

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

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

Example

```
*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 (in hexadecimal by default) of the first byte to be saved
`<end addr>`	the address (in hexadecimal by default) 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 the start address). Load is the reload address (and is assumed to be same as start if omitted).

Example

```
*Save myprog 8000 + 3000

*Save myprog 8000 B000 9300 9000
```

Related commands

*Load, *SetType

*SetType

Establishes a file type for 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. A list of valid file types can be displayed by the command *Show File$Type*.

Use

*SetType sets the file type of the named file. If the file does not have a date stamp, then the current time and date are assigned to the file. Examples of file types are Palette, Font, Sprite and BASIC: a list is given in Appendix B, or can be inspected by typing `*Show File$Type*`.

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
<Press 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

*ShutDown, *Bye, *Close

*ShutDown

Closes files, logs off file servers and parks hard discs.

Syntax

```
*ShutDown
```

Parameters

None

Use

*ShutDown performs all the actions of the *Shut command, and in addition logs off all NetFS file servers and causes hard discs to be parked in a safe state for switching off the computer.

Related commands

*Shut, *Bye, *Close

*Spool

Sends every character written to the screen to the named file.

Syntax

```
*Spool [<filename>]
```

Parameters

`<filename>`	a valid pathname specifying a file

Use

*Spool <filename>	closes any existing spool file and opens a new file to receive data which is sent to the VDU.

`*Spool` (or `*SpoolOn`) with no filename closes the spool file. You also close an existing spool file by the command `*Spool` (or `*SpoolOn`) `<new spool file>`.

Example

Load a simple BASIC program (called, say Test), type `*Spool x` and LIST the program; switch spooling off with `*Spool`, and examine the spooled file using the desktop, to see its distinctive icon. The spooled file, `x`, can be read by a text editor. Drag the file icon to the Edit icon on the icon bar, and an editing window opens with the text of the spooled file.

```
*BASIC
>LOAD "Test"
>*Spool x          spool on – to file x
>LIST
      .
      .
>*Spool            spool off
```

Related commands

*SpoolOn

*SpoolOn

Add characters written to the screen to the end of an existing file.

Syntax

```
*SpoolOn [<filename>]
```

Parameters

`<filename>`	a valid pathname specifying a file

Use

`*SpoolOn <filename>` is similar to *Spool, except that it takes the name of an existing file. Characters written to the screen are also added to the end of the file.

`*SpoolOn` (or `*Spool`) with no filename closes the spool file. You also close an existing spool file by the command `*SpoolOn` (or `*Spool`) `<new spool file>`.

Example

```
*BASIC
>LOAD "Test"
>*SpoolOn myfile    append text to file myfile which already exists
>LIST
      .
      .
>*SpoolOn           close spool file
```

Related commands

*Spool

*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

*SetType, *Info

*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 spaces

Use

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

Example

```
*Type -file myfile -tabexpand
```

Related commands

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

*Up

Moves the current directory up the directory tree.

Syntax

```
*Up [<levels>]
```

Parameters

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

Use

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

Note that while NetFS supports this command, some fileservers do not, so you may get a *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

*Wipe

Deletes one or more objects.

Syntax

```
*Wipe <file spec> [<options>]
```

Parameters

`<file spec>`	a valid pathname specifying an object (or, with wildcards, a group of objects)
`<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 '~' (e.g: ~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).

Application notes

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 section depends on how you intend to write your own filing system:

- if you are not using FileCore, then you should read this section, which tells you how to add a filing system to FileSwitch
- if you are using FileCore, then you should ignore this section and instead read the next chapter entitled *FileCore*.

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

Declaring your 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 this chapter's section on *SWI calls* 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)

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
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 Filing system supports FSEntry_Func 18

Bits 16 - 23 are reserved and should be set to zero.

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.

In addition, bits 0 - 7 contain the filing system identification number. Currently allocated ones are:

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
RISCiXFS	24
Streamer	25
SCSIFS	26
Digitiser	27
Scanner	28
MultiFS	29

For an allocation, contact Acorn Computers in writing.

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
        ADR     R1, FilingSystemName
        SWI     XOS_FSControl
        LDMFD   R13!, {PC}
```

For full details of OS_FSControl 14, see the section on *SWI calls* earlier in this chapter.

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 remaining chapters of 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*, 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 the section on *SWI calls* earlier in this chapter.

Filing system interface: calling conventions

The principal part of a 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 either to access hardware devices directly or to set up FIQ registers as necessary.

Using the stack

R13 in supervisor mode is used as the system stack pointer. The filing system may use this full descending stack. When the filing system is entered you should take care not to push too much onto it, 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 to write out the buffer first, before giving it to the new owner. If an error occurs in writing out the buffer, 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 is set to the value of R3 passed to FileSwitch in the OS_FSControl 12 call that initialised the filing system. 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 address into the register.

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.

Dealing with access

Generally FileSwitch does not make calls to 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 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.

Filing system interfaces

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

FSEntry_Open

On entry

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

On exit

R0 = file information word (not the same as the file system information word)
R1 = file handle used by your filing system (0 if not found)
R2 = buffer size for FileSwitch to use (0 if file unbuffered)
R3 = file extent (buffered files only)
R4 = space currently allocated to file (buffered files)

FileSwitch calls this entry point to open a file for read or write, and to create it if necessary. The reason code given in R0 has the following meaning:

Value	Meaning
0	Open for read
1	Create and open for update
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 exisiting programs which use this as a method of testing the existence of an object. This is also used to open new directory contexts which may be written via FSEntry_Func.

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).

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 two handles that have special meanings:

- a handle of zero means that no file is open
- a handle of –1 is used to indicate 'unset' directory contexts (see FSEntry_Func).

The 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

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

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

The buffer size returned in R2 must be power of 2 between 64 and 1024 inclusive.

FSEntry_GetBytes from...

This call is used to get a single byte or a group of bytes from an open file. There are two distinct cases to consider, depending on whether the file was opened as buffered or unbuffered:

...a buffered file

Get bytes from a buffered file

On entry

R1 = file handle
R2 = memory address to put data
R3 = number of bytes to read
R4 = file offset to get data from

On exit

—

This call reads a number of bytes and places 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. A highly optimised example (similar to that used in RISC OS) is given at the end of this chapter.

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 in toone 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.

...an unbuffered file

Get a byte from an unbuffered file

On entry

R1 = file handle

On exit

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

This call is used 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 the filing system does not support unbuffered GBPB directly, then this entry is called by FileSwitch the number of times requested by the client to complete his request, stopping if it returns C set (EOF).

FSEntry_PutBytes to...

This call is used to put a single byte or group of bytes to a file. Again, there are two distinct cases to consider:

...a buffered file

Put bytes to a buffered file

On entry

R1 = file handle
R2 = memory address to take data from
R3 = number of bytes to put to file
R4 = file offset to put data to

On exit

—

This call is used to 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. A highly optimised example (similar to that used in FileSwitch) is given at the end of this chapter.

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.

...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

On exit

—

This call is used to put a single byte to an unbuffered file at the position given by the file's sequential file pointer. The sequential pointer must be advanced by one. If the sequential pointer is equal to the file extent when this call is made, the allocated space of the file must be incresed by at least one byte to accomodate 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 the filing system does not support unbuffered GBPB directly, then this entry is called by FileSwitch the number of times requested by the client to complete his request.

FSEntry_Args

Various calls are made through this entry point 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

On exit

R2 = sequential file pointer

This call is used to read the sequential file pointer for a given file. Only filing systems which use unbuffered files should support this call.

If the 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
R2 = new sequential file pointer

On exit

—

This call is used to alter the sequential file pointer for a given file. Only filing systems which use unbuffered files should support this call.

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 zeroes 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 the 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

On exit

R2 = file extent

This call is used to read the extent of a given file. Only filing systems which use unbuffered files should support this call.

If the 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

Write file extent

On entry

R0 = 3
R1 = file handle
R2 = new file extent

On exit

—

For buffered files, this call is only issued internally by FileSwitch in order to set the real file extent just prior to closing an open file. The filing system should store the value of R2 in the file's catalogue information as its new length.

For unbuffered files, this call is passed directly through from the user.

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

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

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.

FSEntry_Args 4

Read size allocated to file

On entry

R0 = 4
R1 = file handle

On exit

R2 = size allocated to file by filing system

This call is used 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

On exit

C bit is set if (sequential pointer is equal to current extent), clear otherwise

This call is used to determine whether the sequential pointer for a given file is at the end of the file or not. Only filing systems which use unbuffered files should support this call.

If the 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 the 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

Notify of a flush

On entry

R0 = 6
R1 = file handle

On exit

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

This call is used to notify the filing system to flush any modified data that it is holding in buffers to its storage media. It is only needed when the filing system does its own buffering in addition to that done by FileSwitch. For example, ADFS does so when doing readahead/writebehind.

It is only called by FileSwitch if the filing system is buffered, and bit 27 of the filing system information word was set when the filing system was initialised.

FSEntry_Args 7

Ensure file size

On entry

R0 = 7
R1 = file handle
R2 = size of file to ensure

On exit

R2 = size of file actually ensured

This call is used to ensure that a file is of at least the given size. You must not rely on this call to zero out file storage for you. All filing systems must support this call.

FSEntry_Args 8

Write zeros to file

On entry

R0 = 8
R1 = file handle
R2 = file address to write zeros at
R3 = number of zero bytes to write

On exit

—

This call is used to take a number of zero 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. A highly optimised example (similar to that used in FileSwitch) is given at the end of this chapter.

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..

All filing systems which use buffered files must support this call.

FSEntry_Args 9

Read file datestamp

On entry

R0 = 9
R1 = file handle

On exit

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

This call is used 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 a filing system cannot stamp an open file given its handle, then it should return R2 and R3 set to zero.

FSEntry_Close

Close an open file

On entry

R1 = file handle
R2 = new load address to associate with file
R3 = new execution address to associate with file

On exit

—

This call is used to close an open file and put a new date/time stamp on it.

If the filing system returned from the FSEntry_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 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 filing system for closing. Filing systems should not try to support this themselves.

FSEntry_File

This call is used to perform operations on whole files depending on the value of R0 as follows:

FSEntry_File 0

Save file

On entry

R0 = 0
R1 = pointer to filename (null terminated)
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

On exit

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

This call saves an area of memory to a file. FileSwitch has already validated the area for saving from, and ensured that the leafname is not wildcarded. An error such as `File locked` should be returned if the specified file could not be saved.

The leafname is immediately copied by FileSwitch, so need not have a long lifetime. You could hold it in a small static buffer, for example.

FSEntry_File 1

Write catalogue information

On entry

R0 = 1
R1 = pointer to wildcarded filename (null terminated)
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

On exit

—

This call updates the catalogue information. 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 (null terminated)
R2 = new load address to associate with file
R6 = pointer to special field if present, otherwise 0

On exit

—

This call 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 (null terminated)
R3 = execution address to associate with file
R6 = pointer to special field if present, otherwise 0

On exit

—

This call 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 (null terminated)
R5 = new attributes to associate with file
R6 = pointer to special field if present, otherwise 0

On exit

—

This call alters the attributes of an object. You must not return an error if the object does not exist.

FSEntry_File 5

Read catalogue information

On entry

R0 = 5
R1 = pointer to pathname (null terminated)
R6 = pointer to special field if present, otherwise 0

On exit

R0 = object type

0	not found
1	file
2	directory

R2 = load address
R3 = execution address
R4 = file length
R5 = file attributes

This call 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

Delete object

On entry

R0 = 6
R1 = pointer to filename (null terminated)
R6 = pointer to special field if present, otherwise 0

On exit

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

This call deletes an object. FileSwitch will already have ensured that the leafname is not wildcarded. No data need be transferred to the file. An error should be returned 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

Create file

On entry

R0 = 7
R1 = pointer to filename (null terminated)
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

On exit

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

This call creates a file with a given name. R4 and R5 are used only to calculate the length of the the file to be created. If the file currently exists and is not locked, the old file is first discarded. An error should be returned if the file could not be created.

FSEntry_File 8

Create directory

On entry

R0 = 8
R1 = pointer to directory name (null terminated)
R4 = number of entries (0 for default)
R6 = pointer to special field if present, otherwise 0

On exit

—

This call 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.

FileSwitch will already have ensured that the leafname is not wildcarded. An error should be returned if the directory could not be created.

FSEntry_File 9

Read catalogue information (no length)

On entry

R0 = 9
R1 = pointer to filename (null terminated)
R6 = pointer to special field if present, otherwise 0

On exit

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

This call returns 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 called by FileSwitch if bit 26 of the filing system information word was set when the filing system was initialised. Otherwise FSEntry_File 5 is called, and the length returned in R4 is ignored.

FSEntry_File 255

Load file

On entry

R0 = 255
R1 = pointer to wildcarded filename (null terminated)
R2 = address to load file
R6 = pointer to special file if present; otherwise zero

On exit

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

FileSwitch has always called FSEntry_File 5 and validated the client's load request before calling FSEntry_File 255. 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'

Therfore 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 filename !Boot.

FSEntry_Func

Various calls are made through this entry point to deal with assorted 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 WrCh process being in the middle of spooling, or by an active task swapper.

If you add a header to output (cf `*Info *` 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_ReadVduVars &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 (null terminated)
R6 = pointer to special field if present, otherwise zero

On exit

—

This call is used to set the current directory to the one specified by the directory name and context given. If the directory name is null, it is assumed 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.

FSEntry_Func 1

Set library directory

On entry

R0 = 1
R1 = pointer to wildcarded directory name (null terminated)
R6 = pointer to special field if present, otherwise zero

On exit

—

This call is used to set the current library directory to the one identified by the directory name and context given. If the directory name is null, it is assumed 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.

FSEntry_Func 2

Catalogue directory

On entry

R0 = 2
R1 = pointer to wildcarded directory name (null terminated)
R6 = pointer to special field if present, otherwise zero

On exit

—

This call is used to catalogue the directory identified by the directory name and context given. If the directory name is null, it is assumed to be the current directory. (This corresponds to the *Cat command.)

FSEntry_Func 3

Examine current directory

On entry

R0 = 3
R1 = pointer to wildcarded directory name (null terminated)
R6 = pointer to special field if present, otherwise zero

On exit

—

This call is used to print information on all the objects in the directory identified by the directory name and context given. If the directory name is null, it is assumed to be the current directory. (This corresponds to the *Ex command.)

FSEntry_Func 4

Catalogue library directory

On entry

R0 = 4
R1 = pointer to wildcarded directory name (null terminated)
R6 = pointer to special field if present, otherwise zero

On exit

—

This call is used to catalogue the specified subdirectory relative to the current library directory. If the directory name is null, it is assumed to be the current library directory. (This corresponds to the *LCat command.)

FSEntry_Func 5

Examine library directory

On entry

R0 = 5
R1 = pointer to wildcarded directory name (null terminated)
R6 = pointer to special field if present, otherwise zero

On exit

—

This call is used to print information on all the objects in the specified subdirectory relative to the current library directory. If the directory name is null, it is assumed to be the current library directory. (This corresponds to the *LEx command.)

FSEntry_Func 6

Examine object(s)

On entry

R0 = 6
R1 = pointer to wildcarded pathname (null terminated)
R6 = pointer to special field if present, otherwise zero.

On exit

—

This call is used to print information on all the objects matching the wildcarded pathname and context given, in the same format as for Examine directory. (This corresponds to the *INFO command.)

FSEntry_Func 7

Set filing system options

On entry

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

On exit

—

This call is used 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

Rename object

On entry

R0 = 8
R1 = pointer to first pathname (null terminated)
R2 = pointer to second pathname (null terminated)
R6 = pointer to first special field if present, otherwise 0
R7 = pointer to second special field if present, else 0

On exit

R1 = 0 if rename performed (<>0 otherwise)

This call is used 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. In this case, FileSwitch will return a 'Bad rename' error.

FSEntry_Func 9

Access object(s)

On entry

R0 = 9
R1 = pointer to wildcarded pathname (null terminated)
R2 = pointer to access string (null, space or control-character terminated)

On exit

—

This call is used 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.

FSEntry_Func 10

Boot filing system

On entry

R0 = 10

On exit

—

The filing system should perform its boot action on this call. For example, ADFS examines the boot option (as set by *OPT 4) of the disc in the configured drive and acts accordingly, for example `*Run &.!Boot` if boot option 2 is set; 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 = memory address to put data
R6 = 0 (cannot specify a context)

On exit

—

This call is used 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 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 = memory address to put data
R6 = 0 (cannot specify a context)

On exit

—

This call is used 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 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 13

Read library directory name and privilege byte

On entry

R0 = 13
R2 = memory address to put data

On exit

—

This call is used 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 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 14

Read directory entries

On entry

R0 = 14
R1 = pointer to wildcarded directory name (null terminated)
R2 = memory address 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 = buffer length
R6 = pointer to special field if present, otherwise zero

On exit

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

This call is used 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 the currently-selected 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.

FSEntry_Func 15

Read directory entries and information

On entry

R0 = 15
R1 = pointer to wildcarded directory name (null terminated)
R2 = memory address 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 = buffer length
R6 = pointer to special field if present, otherwise zero

On exit

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

This call is used to read the leaf names of entries (and their file information) in the given directory into an area of memory 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 (null terminated)

FileSwitch will have validated the area of memory. 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

—

On this call, the filing system should attempt to go into as dormant a state as possible. For example, it should place Winchester 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

—

This call is used 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

On entry

R0 = 18
R1 = new currently-selected directory handle (0 = no change, –1 for 'Unset')
R2 = new user root directory handle (0 = no change, –1 for 'Unset')
R3 = new library handle (0 = no change, –1 for 'Unset')
R6 = 0 (cannot specify a context)

On exit

R1 = old selected directory handle (–1 if 'Unset')
R2 = old user root directory handle (–1 if 'Unset')
R3 = old library handle (–1 if 'unset')

This call is used to redefine (or read) the currently-selected directory, user root directory and library handles. FileSwitch will have ensured that all handles being written are on the same filing system.

This call is only ever made to filing systems that have bit 24 set in the filing system information word.

FSEntry_Func 19

Read directory entries and information

On entry

R0 = 19
R1 = pointer to wildcarded directory name (null terminated)
R2 = memory address to put data
R3 = number of object names to read
R4 = offset of first item to read in directory
R5 = buffer length
R6 = pointer to special field if present, otherwise zero

On exit

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

This call reads the names of entries (and their file information) in the given directory into an area of memory 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 (null terminated)

Each record is word-aligned.

FSEntry_Func 20

Output full information on object(s)

On entry

R0 = 20
R1 = pointer to wildcarded pathname (null terminated)
R6 = pointer to special field if present, otherwise zero.

On exit

—

This call is used to output full information on all the objects matching the wildcarded pathname given. 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.

FSEntry_GBPB

Get/put bytes from/to an unbuffered file

This entry point isused 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
R2 = start address of buffer in memory
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

This call is used to 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
R2 = start address of buffer in memory

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

This call is used to transfer data from the file to memory at either the specified file pointer (R0 = 3) or 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 a highly 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
```

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.

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.

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

On a floppy drive the following recording formats are available:

Format	Tracks	Density	Sectors/track	Bytes/sector	Storage
L	80	Double	16	256	640 Kbytes
D	80	Double	5	1024	800 Kbytes
E	80	Double	5	1024	800 Kbytes

Using the L format you can create 47 entries in each directory; top-bit-set characters are not allowed in pathnames. This is the *old format*. Using the D or E formats, 77 entries may be created, and top-bit-set characters are allowed in pathnames.

E format has the following additional advantages:

- files need not be stored contiguously, so you don't need to compact it (however, FileCore does try to create E format 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.

These are because E format uses a new disc map; L and D formats are said to use the *old map*.

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. 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 the chapter entitled *Communications within RISC OS* for details.

In fact, disc names may be used in any pathname given to the system. When used in a pathname, the disc name (or number) must be prefixed by a colon. Examples of pathnames with disc specifiers are:

```
*CAT :MikeDisc.fonts
*INFO :4.LIB*.*
```

Note that :drive really means :drive.$.

Disc names can have wildcards in them, so long as the name only matches one of the discs that FileCore knows about for the filing system. If more than one name matches FileCore will return an `Ambiguous disc name` error.

You are very strongly recommended to use disc names rather than drive numbers when you write programs.

Changing discs

FileCore keeps track of eight disc names per filing system, on a first in, first out basis. When you eject a floppy disc from the drive, FileCore still 'knows' about it. This means that if there are any directories set on that disc (the current directory, user root directory, or library), they will still be associated with it. Thus any attempt to load or run a file will result in a `Disc not present/known` error.

However, this means that you can replace the disc and still use it, as if it had never been ejected. The same applies to open files on the disc; they remain open and associated with that disc until they are closed.

You can cause the old directories to be overridden by `*Mounting` a new disc once it has been inserted. This resets the CSD and so on. Alternatively, if you unset the directories (using *NoDir, *NoLib and *NoURD), then FileCore will use certain defaults when operations on these are required.

If there is no current directory, FileCore will use $ on the default drive. This is the configured default, or the one set by the last *DRIVE command.

If there is no user root directory set, then references to that directory will use $ on the default drive.

If there is no library set, then FileCore will try &.Library, $.Library and then the current directory, in that order.

Current selections

The currently selected directory, user root directory and library directory are all stored independently for each FileCore-based filing system.

Disc addresses

The 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 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.

Disc records

A *disc record* describes the shape and format of a disc. It is 64 bytes long:

Offset	Meaning
0	$\log_2$ of sector size
1	sectors per track
2	heads (1 for old directories)
3	density (1, 2 or 4 – ie 256, 512 or 1024 bytes per sector) if applicable
4	length of id field of map fragment, in bits
5	$\log_2$ of bytes for each map bit (0 for old map)
6	track to track sector skew for random access file allocation
7	boot option
8	reserved
9	number of zones in the map

10	bits in zone which are neither map bits nor special Zone 0 Bytes
12	disc address of root directory
16	disc size in bytes
20	disc id
22	disc name
32 - 63	reserved bytes

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.

Reserved bytes should be set to zero.

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.

Defect lists

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 &200000XX. 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
```

Boot blocks

Hard discs contain a 512 byte *boot block* at disc address &C00, which contains important information. (On a disc with 256-byte sectors, such as ADFS uses, this corresponds to sectors 12 and 13 on the disc.) A boot block has the following format:

Offset	Contents
&000 upwards	Defective sector list (see above)
&1BF downwards	Hardware-dependent parameters
&1C0 - &1FF	Disc record (see above)

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

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.

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
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		
15	Hard disc: specify	R2	

The option bits have the following meanings:

Bit 4

This bit is set if an alternate defect list for a hard disc is to be used. This is assumed to be in RAM 64 bytes after the start of the disc record pointed to by R5.

This bit may only be set for old map discs.

Bit 5

If this bit is set, then the meaning of R3 is altered. It does not point to the area of RAM to or from which the disc data is to be transferred. Instead, it points to a word-aligned list of memory address/length pairs. All but the last of these lengths must be a multiple of the sector size. These word-pairs are used for the transfer until the total number of bytes given in R4 has been transferred.

On exit, R3 points to the first pair which wasn't fully used, and this pair is updated to reflect the new start address/bytes remaining, so that a subsequent call would continue from where this call has finished.

This bit may only be set for reason codes 0 - 2.

Bit 6

If this bit is set then escape conditions are ignored during the operation, otherwise they cause it to be aborted.

Bit 7

If this bit is set, then the usual time-out for floppy discs of 1 second is not used. Instead FileCore will wait (forever if necessary) for the drive to become ready.

The disc address must be on a sector boundary for reason codes 0 - 2, and on a track boundary for other reason codes. Note that you must make allowances for any defects, as the disc address is not corrected for them.

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.

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 *Application notes* later in this chapter, on adding a filing system to FileCore.

The only start-up option (passed in bits 25 - 31 of R3) currently supported is *No directory state* which is indicated by setting bit 30. All other bits representing start-up options must be clear.

If the filing system does not support background transfers of data, R5 must be zero.

The hard disc map sizes are given using 1 byte for each disc. The byte should contain *map size/256* (ie 2 for the old map). This is just a good guess and should not involve starting up the drives to read from them You might store this in the CMOS RAM.

You must store the FileCore instance private word returned by this SWI in your module workspace; it is your module's means of identifying itself to FileCore.

When your module calls the addresses returned in R1 - R3, it must be in SVC mode with R12 holding the value of R0 that this SWI returned. Interrupts need not be disabled. R0 - R11 and R13 will be preserved by FileCore over these calls.

Related SWIs

None

Related vectors

None

FileCore_Drives (SWI &40542)

Returns information on the filing system's drives

On entry

R8 = pointer to FileCore instance private word

On exit

R0 = default drive
R1 = number of floppy drives
R2 = number of hard disc drives

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns information on the filing system's drives.

Related SWIs

None

Related vectors

None

FileCore_FreeSpace (SWI &40543)

Returns information on a disc's free space

On entry

R0 = pointer to disc specifier (null terminated)
R8 = pointer to FileCore instance private word

On exit

R0 = total free space on disc
R1 = size of largest object that can be created

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the total free space on the given disc, and the largest object that can be created on it.

Related SWIs

None

Related vectors

None

FileCore_FloppyStructure (SWI &40544)

Creates a RAM image of a floppy disc map and root directory entry

On entry

R0 = pointer to buffer
R1 = pointer to disc record describing shape and format
R2 bit 7 set for old directory structure
bit 6 set for old map
R3 = pointer to list of defects

On exit

R3 = total size of structure created

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call creates a RAM image of a floppy disc map and root directory entry.

The pointer to a list of defects is only needed for new map discs. They must be byte addresses giving the start of defective sectors, and terminated with &20000000.

You do not need to know a FileCore instantiation private word to use this call; instead the disc record tells FileCore which filing system is involved.

Related SWIs

None

Related vectors

None

FileCore_DescribeDisc (SWI &40545)

Returns a disc record describing a disc's shape and format

On entry

R0 = pointer to disc specifier (null terminated)
R1 = pointer to 64 byte block
R8 = pointer to FileCore instance private word

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns a disc record in the 64 byte block passed to it. The record describes the disc's shape and format.

Related SWIs

None

Related vectors

None

*Back

Exchanges current and previous directories.

Syntax

```
*Back
```

Use

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

Related commands

*Dir, which changes to a specified directory

*Backup

Copies the used part of a floppy disc.

Syntax

```
*Backup <source drive> <dest drive> [Q]
```

Parameters

`<source drive>`	the number of the floppy drive (0 to 3)
`<dest drive>`	as source drive, above

`[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 a whole floppy disc to another, with the exception of free space. 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

*Dismount, *Shutdown, *Shut, *Close

*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-format disc (whether floppy or hard) has the correct checksums and is consistent with the directory tree. If only one copy of the map is good, it allows you to rewrite the bad one with the information in the good one.

Example

```
*CheckMap :Mydisc
```

Related commands

*Defect, *Verify

*Compact

Collects free spaces together on L- and D- format discs, and old map hard discs

Syntax

```
*Compact [<disc spec>]
```

Parameters

`<disc spec>` the name of the disc or number of the disc drive

Use

*Compact collects free spaces together 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 that is larger than the biggest single free space. Because the free space has been gathered together, the maximum size of file you can fit on the disc will be as high as is possible.

The maximum size of file you can add to an E-format disc does not depend on how fragmented the free space is, so there is not the same need to compact them. This command is still useful, as it will attempt to gather together any fragmented files, and generally tidy the disc up.

Example

```
*Compact :0
```

Related commands

*CheckMap, *Map, *FileInfo.

*Configure Dir

Mounts a disc on power on.

Syntax

```
*Configure Dir
```

Use

*Configure Dir mounts a disc on power on (see *Mount) on all FileCore-based filing systems that support mounting (ADFS does, RamFS doesn't). NoDir is the default setting.

This option can also be set from the desktop, using the Configure application.

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

*Configure NoDir

Does not mount a disc on power on.

Syntax

```
*Configure NoDir
```

Use

*Configure NoDir does not mount a disc on power on (see *Mount) on any FileCore-based filing system that supports mounting (ADFS does, RamFS doesn't). This is the default setting.

This option can also be set from the desktop, using the Configure application.

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 Dir , *Configure Drive

*Defect

Marks a part of a disc as defective so that it will no longer be used.

Syntax

```
*Defect <disc spec> <disc add>
```

Parameters

`<disc spec>`	the name of the disc or number of the disc drive
`<disc add>`	the address where the defect exists (provided by the *Verify command if the disc is faulty)

Use

If a physical defect occurs in an unallocated part of an E-format disc, *Defect will render that part of the disc inaccessable by altering the 'map' of the disc.

The disc address must be a multiple of 256 – that is, it must end in '00'.

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, that part of the disc is then unallocated, so you can repeat the *Defect command to make the defective part of the disc inaccessable.

Sometimes the disc will be too badly damaged for you to successfully delete objects listed by the *Defect command. In such cases the damage cannot be repaired, and you must restore the objects from a recent backup.

Example

```
*Verify mydisc
Disc error 08 at :0/00010400
*Defect mydisc 10400
$.mydir must be moved
.myfile1 has defect at offset 800
.myfile2 must be moved
```

Related commands

*Verify, *CheckMap

*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 closes files, unsets directories that were set on the given disc, and parks 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, (it is essential if the disc is to taken away and modified on another computer), but the *Shutdown command is usually to be preferred, especially when switching off the computer.

Example

```
*Dismount
```

Related commands

*Mount.

*Drive

Sets the current drive.

Syntax

```
*Drive <drive>
```

Parameters

`<drive>` the number of the disc drive, from 0 to 7

Use

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 amount of unused disc space.

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

*Free, *Compact

*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 sets the directory to the disc root directory, sets the library directory (if it is currently unset) to $.Library, and unsets the URD (User Root Directory). If no disc spec is given, the default drive is used. The command is preserved for the sake of compatibility with earlier Acorn operating systems.

Example

```
*Mount :mydisc
```

Related commands

*Dismount

*NameDisc

Changes a disc's name.

Syntax

```
*NameDisc <disc spec> <disc name>
```

Parameters

`<disc spec>` the present name of the disc or number of the disc drive
`<disc name>` a text string up to 10 characters long

Use

*NameDisc (or alternatively, *NameDisk) allows you to change a disc's name.

Example

```
*NameDisc :0 DataDisc
```

Related commands

None

*NoDir

Unsets the current directory.

Syntax

```
*NoDir
```

Use

*NoDir unsets the current directory.

Related commands

*NoURD, *NoLib

*NoLib

Unsets the library.

Syntax

```
*NoLib
```

Use

*NoLib unsets the library.

Related commands

*Lib

*NoURD

Unsets the User Root Directory (URD).

Syntax

```
*NoURD
```

Use

*NoURD unsets the User Root Directory.

Related commands

*NoDir, *URD

*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. Titiles 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.

Related commands

*Cat displays the titles of directories.

*URD

Sets the User Root Directory.

Syntax

```
*URD [<directory>]
```

Parameters

`<directory>`	any valid pathname specifying a directory

Use

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

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

Example

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

Related commands

*NoURD

*Verify

Checks a disc for readability.

Syntax

```
*Verify [<disc spec>]
```

Parameters

`<disc spec>` the name of a disc or number of the disc drive, from 0 to 7 (ie applies to floppy discs and hard discs)

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.

Example

```
*Verify 4
```

```
*Verify :Mydisc
```

Related commands

*Defect

Application notes

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 FileSwitch.

This section describes how to add a filing system to FileCore. You should also see the chapter entitled *Modules* 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 this chapter's section on *SWI calls* for details.

R0 tells FileCore where to find a *descriptor block*. This in turn tells FileCore the locations of the entry points to your module's low level routines that interface with the hardware:

Descriptor block

Offset	Contains
0	flags: bit 0 set if winnie needs FIQ bit 1 set if floppy needs FIQ bit 2 set if able to support background ops
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

FileCore_Create starts a new instantiation of FileCore, and the pointer to the new workspace that has been reserved for FileCore is returned in R0 to your module. You must store this in your module 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.

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 the chapter entitled *FileSwitch*.

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 chapter entitled *Modules*.

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, except:

- an extra reason code is added to R1 allow background processing
- consequently R1 can no longer used to point to an alternative disc record; instead R5 is used
- R6 points to a boot block (for hard disc operations only).

These are the reason codes that may be passed in R0:

Value	Meaning
0	Verify
1	Read sectors
2	Write sectors
3	Floppy disc: read track Hard disc: read Id
4	Write track
5	Seek (used only to park)
6	Restore
7	Floppy disc: step in
8	Floppy disc: step out
15	Hard disc: specify

The reason codes you **must** support are 0, 1, 2, 5 and 6.

Your routine must preserve R1, R5 - R13, and the N, Z and C flags. R2 must be incremented by the amount transferred for Ops 0, 1 and 2; otherwise you must preserve it. R3 must be incremented appropriately for Ops 1 and 2; otherwise you must preserve it. R4 must be decremented by the amount transferred for Ops 0, 1 and 2; otherwise you must preserve it.

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 ≤ R0 < 2^{31}	pointer to error block
R0 ≥ 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

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 must point 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 to be filled in.

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	reserved

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 the end of list must be searched for. From this the start of the scatter block may be calculated.

Miscellaneous entry

This entry performs various miscellaneous tasks, depending on the value of R0.

Your routine must preserve registers, and the N, Z and C flags.

You may only return an error from reason code 0 (Mount); this must be done in the same way as for the DiscOp entry.

Miscellaneous entry 0 — Mount

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 only)

On exit

R1 - R5 preserved

For a floppy disc, this asks you to read in the free space map and identify the format. The suggested density to try first is given in the disc record. Identifying the format consists of filling in the density, and for old format discs the sector size, sectors per track, heads, disc size and root dir.

For a hard disc, this asks you to read in the boot block. If the disc doesn't have one, your module will have to generate one itself.

Miscellaneous entry 1 — Poll changed

On entry

R0 = 1
R1 = drive
R2 = sequence number

On exit

R2 = sequence number
R3 = result flags

The sequence number is to ensure no changes are lost due to reset being pressed. Both your module and the FileCore incarnation should start with a sequence number of 0 for each drive. Your module increments the sequence number with each change of state. If your module 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
6 empty works
7 changed works

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.

Miscellaneous entry 2

Lock drive

On entry

R0 = 2
R1 = drive

On exit

—

This can only be called for a floppy drive. It should at least ensure that the drive light stays on until unlocked.

Miscellaneous entry 3

Unlock drive

On entry

R0 = 3
R1 = drive

On exit

—

This can only be called for a floppy drive.

Miscellaneous entry 4

Poll period

On entry

R0 = 4
R1 = drive

On exit

R5 = minimum polling period (in centi-seconds), or
–1 if disc changed doesn't work
R6 = pointer to media type string eg 'disc' for ADFS

This call informs FileCore of the minimum period between polling for disc insertion. This is so that drive lights do not remain continously illuminated. The values are re-exported by FileCore in the up calls MediaNotPresent and MediaNotKnown. The value applies to all drives rather than a particular drive.

ADFS

Introduction

ADFS is the Advanced Disc Filing System. It is a module that, together with FileSwitch and FileCore, provides a disc-based filing system.

Most of the facilities that you will use with ADFS are in fact provided by FileCore and FileSwitch, and you should read the chapters on those modules in conjunction with this one.

Overview

ADFS is a module that provides the hardware-dependent part of a disc-based filing system. It uses FileCore, and so conforms to the standards for a module that does so; see the chapter entitled *FileCore* for details.

It provides:

- a * Command to select itself (*ADFS)
- a * Command to format discs (*Format)
- various configure options, accessed using *Configure
- four SWIs that give access to corresponding FileCore SWIs
- two further SWIs to set the address of an alternative hard disc controller, and to set the number of retries used for various operations
- the entry points and low-level routines that FileCore needs to access the disc controllers and associated hardware.

Except for the low-level entry points and routines (which are for the use of FileCore only) all of these are described below.

ADFS_DiscOp (SWI &40240)

Calls FileCore_DiscOp

On entry

See FileCore_DiscOp (SWI &40540)

On exit

See FileCore_DiscOp (SWI &40540)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_DiscOp (SWI &40540), after first setting R8 to point to the FileCore instantiation private word for ADFS.

This call is functionally identical to FileCore_DiscOp (SWI &40540).

Related SWIs

FileCore_DiscOp (SWI &40540)

Related vectors

None

ADFS_HDC (SWI &40241)

Sets the address of an alternative hard disc controller

On entry

R2 = address of alternative hard disc controller
R3 = address of poll location for IRQ/DRQ
R4 = bits for IRQ/DRQ
R5 = address to enable IRQ/DRQ
R6 = bits to enable IRQ/DRQ

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets up the address of the hard disc controller to be used by the ADFS. For instance, an expansion card can supply an alternative controller to the one normally used.

The polling and interrupt sense is done using:

```
LDRB  Rn, [poll location]
TST   Rn, [poll bits]
```

The IRQ/DRQ must be 1 when active.

Related SWIs

None

Related vectors

None

ADFS_Drives (SWI &40242)

Calls FileCore_Drives

On entry

See FileCore_Drives (SWI &40542)

On exit

See FileCore_Drives (SWI &40542)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_Drives (SWI &40542), after first setting R8 to point to the FileCore instantiation private word for ADFS.

This call is functionally identical to FileCore_Drives (SWI &40542).

Related SWIs

FileCore_Drives (SWI &40542)

Related vectors

None

ADFS_FreeSpace (SWI &40243)

Calls FileCore_FreeSpace

On entry

See FileCore_FreeSpace (SWI &40543)

On exit

See FileCore_FreeSpace (SWI &40543)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_FreeSpace (SWI &40543), after first setting R8 to point to the FileCore instantiation private word for ADFS.

This call is functionally identical to FileCore_FreeSpace (SWI &40543).

Related SWIs

FileCore_FreeSpace (SWI &40543)

Related vectors

None

ADFS_Retries (SWI &40244)

Sets the number of retries used for various operations

On entry

R0 = mask of bits to change
R1 = new values of bits to change

On exit

R0 preserved
R1 = R0 AND entry value of R1
R2 = old value of retry word
R3 = new value of retry word

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets the number of retries used by writing to the retry word. The format of this word is:

Byte	Number of retries for
0	hard disc read/write sector
1	floppy disc read/write sector
2	floppy disc mount (per copy of the disc map)
3	verify after *Format, before sector is considered a defect

The new value is calculated as follows:

(old value AND NOT R0) EOR (R1 AND R0)

Related SWIs

None

Related vectors

None

ADFS_DescribeDisc (SWI &40245)

Calls FileCore_DescribeDisc

On entry

See FileCore_DescribeDisc (SWI &40545)

On exit

See FileCore_DescribeDisc (SWI &40545)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_DescribeDisc (SWI &40545), after first setting R8 to point to the FileCore instantiation private word for ADFS.

This call is functionally identical to FileCore_DescribeDisc (SWI &40545).

Related SWIs

FileCore_DescribeDisc (SWI &40545)

Related vectors

None

*ADFS

Selects the Advanced Disc Filing System.

Syntax

```
*ADFS
```

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.

Related commands

*RAM, *Net

*Configure ADFSbuffers

Sets the number of ADFS file buffers.

Syntax

```
*Configure ADFSbuffers <n>
```

Parameters

<n> number of buffers

Use

*Configure ADFSBuffers sets the number of 1024 byte file buffers reserved for ADFS in order to speed up operations on open files. A value of n=1 reserves the default number of buffers appropriate to the RAM size supplied; a value of n=0 disables fast buffering on open files.

Example

```
*Configure ADFSbuffers 8
```

*Configure ADFSDirCache

Reserves an area of RAM for the directory cache.

Syntax

```
*Configure ADFSDirCache <size>[K]
```

Parameters

`<size>` kilobytes of memory reserved

Use

*Configure ADFSDirCache reserves an area of memory 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 selects a default value which depends on RAM size.

Example

```
*Configure ADFSDirCache 16K
```

*Configure Drive

Sets the drive selected at power on.

Syntax

```
*Configure Drive <n>
```

Parameters

<n> drive number

Use

*Configure Drive sets the number of the drive which will automatically be selected on 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 number of floppy disc drives recognised at power on.

Syntax

```
*Configure Floppies <n>
```

Parameters

<n> 0 to 4

Use

*Configure Floppies sets the number of floppy disc drives the computer will recognise. The default value is 1.

Example

```
*Configure Floppies 0
```

Related commands

*Configure HardDiscs

*Configure HardDiscs

Sets the number of hard disc drives recognised at power on.

Syntax

```
*Configure HardDiscs <n>
```

Parameters

<n> 0 to 2

Use

*Configure HardDiscs sets the number of hard disc drives the computer will recognise. The default value depends on the model number of the computer (for example, an Archimededes 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 Step

Sets the step rate of the floppy disc drive.

Syntax

```
*Configure Step <n> [<drive>]
```

Parameters

<n>	step time in milliseconds
<drive>	0 to 3

Use

*Configure Step sets the floppy disc drive step rate to <n>, the step time in milliseconds. If the parameter <drive> number 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.

Example

```
*Configure Step 3
```

*Format

Prepares a new floppy disc for use, or erases a used disc for re-use.

Syntax

```
*Format <drive> [L|D|E] [Y]
```

Parameters

`<drive>` the number of the disc drive, from 0 to 3

`L|D|E` the type of format required, selected from:

L	640K	47-entry directories	old map	all ADFS
D	800K	77-entry directories	old map	Arthur 1.2
E	800K	77-entry directories	new map	RISC OS

`Y` no prompt for confirmation

Use

The default is E-format. This format offers improved handling of file fragmentation on the disc and therefore does not need to be periodically compacted (see the *Compact command).

Example

`*Format 0` Formats to default E-format.

`*Format 0 L` Formats the disc in drive 0 for use with ADFS on the BBC Master range of computers.

Related commands

*Compact

RamFS

Introduction

RamFS is the RAM Filing System. It is a module that, together with FileSwitch and FileCore, provides a RAM-based filing system.

Most of the facilities that you will use with RamFS are in fact provided by FileCore and FileSwitch, and you should read the chapters on those modules in conjunction with this one.

Overview

RamFS is a module that provides the hardware-dependent part of a RAM-based filing system. It uses FileCore, and so conforms to the standards for a module that does so; see the chapter entitled *FileCore* for details.

It provides:

- a * Command to select itself (*RamFS)
- four SWIs that give access to corresponding FileCore SWIs
- the entry points and low-level routines that FileCore needs to access the RAM-based filing system.

Except for the low-level entry points and routines (which are for the use of FileCore only) all of these are described below.

RamFS_DiscOp (SWI &40780)

Calls FileCore_DiscOp

On entry

See FileCore_DiscOp (SWI &40540)

On exit

See FileCore_DiscOp (SWI &40540)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_DiscOp (SWI &40540), after first setting R8 to point to the FileCore instantiation private word for RamFS.

This call is functionally identical to FileCore_DiscOp (SWI &40540).

Related SWIs

FileCore_DiscOp (SWI &40540)

Related vectors

None

RamFS_Drives (SWI &40782)

Calls FileCore_Drives

On entry

See FileCore_Drives (SWI &40542)

On exit

See FileCore_Drives (SWI &40542)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_Drives (SWI &40542), after first setting R8 to point to the FileCore instantiation private word for RamFS.

This call is functionally identical to FileCore_Drives (SWI &40542).

Related SWIs

FileCore_Drives (SWI &40542)

Related vectors

None

RamFS_FreeSpace (SWI &40783)

Calls FileCore_FreeSpace

On entry

See FileCore_FreeSpace (SWI &40543)

On exit

See FileCore_FreeSpace (SWI &40543)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_FreeSpace (SWI &40543), after first setting R8 to point to the FileCore instantiation private word for RamFS.

This call is functionally identical to FileCore_FreeSpace (SWI &40543).

Related SWIs

FileCore_FreeSpace (SWI &40543)

Related vectors

None

RamFS_DescribeDisc (SWI &40785)

Calls FileCore_DescribeDisc

On entry

See FileCore_DescribeDisc (SWI &40545)

On exit

See FileCore_DescribeDisc (SWI &40545)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_DescribeDisc (SWI &40545), after first setting R8 to point to the FileCore instantiation private word for RamFS.

This call is functionally identical to FileCore_DescribeDisc (SWI &40545).

Related SWIs

FileCore_DescribeDisc (SWI &40545)

Related vectors

None

*Ram

Selects the RAM filing system if RamFS is configured.

Syntax

```
*Ram
```

Related commands

*ADFS, *Net

NetFS

Introduction

The NetFS is a filing system that allows you to access and use remote file server machines, using Acorn's Econet network. In common with other filing systems it uses the FileSwitch module, and so when you are using the NetFS you can use any of the commands that FileSwitch provides.

The NetFS module takes the commands that you give to it, either directly or via FileSwitch, and converts them to file server commands. These commands are then sent to the file server using the standard protocol of Econet. The file server then acts on the files or directories that it stores.

Much of the above is transparent to the user, and in general to use file servers you do not need to know file server protocols, or how data is sent over the Econet. For advanced work, you can communicate directly with file servers. If you do need to know more about file server and Econet protocols, you should see:

- the chapter entitled *Econet*
- the *Econet Advanced User Guide*, available from your Acorn supplier

Overview

The NetFS software provides a filing system for RISC OS. To do this it communicates via the Econet with a file server; the file server stores the files and keeps track of them in its directories, as well as providing authenticated access. The NetFS software translates the user's requests that emerge from FileSwitch into one or more file server commands. These commands are then sent to the file server where they act on the files or directories stored there.

The NetFS software is designed to hold information about each file server that it is logged on to and to use this information when communicating with the file server. There are also some extra commands provided by the NetFS software that communicate directly with the file server.

All communication with the file server is done using the interfaces provided by Econet. Basic communication with a file server involves you transmitting a command to it, and then receiving a reply. Either or both of these may contain your data: for instance when you create a directory the name you supply is sent to the file server, where as when you read the name of the current disc that name is sent back to you. Most commands however send things in both directions. The NetFS software knows all the formats and requirements of the file server and presents these to the user, via FileSwitch.

The other commands (those that do not involve files or directories directly) are accessed via star commands. These commands are only available when NetFS is the current filing system.

There are three commands related to access control: *Logon, *Pass, and *Bye. Two commands are to do with selecting file servers: *FS, and *ListFS. The *Free command provides information about the amount of free space remaining on each of the discs of a file server. The two commands *Mount and *SDisc are identical; the former is provided for compatability with ADFS, the latter for compatability with existing network software (ANFS and NFS).

Technical Details

Naming

As well as supplying a filing system name as part of a file name (such as 'Net:&.Fred), you can supply as part of the filing system name the name or number of a file server: for example 'Net#253:&.Fred' or 'Net#Maths:Program'. This will cause the file to be found (or saved, or whatever) on the given file server. If a name is quoted, you must currently be logged on to that file server. If a number is given then you must be logged on to the resulting file server; if only part of the number is given then it will be defaulted against the current file server number.

Timeouts

The dynamics of communication are controlled by several timeouts.

The values used by NetFS for the TransmitCount, TransmitDelay, and ReceiveDelay are more fully explained in the chapter entitled *Econet*. These are the values used for all normal communication with the file server.

Before attempting to log on to a file server, NetFS tries the immediate operation MachinePeek to the file server. This operation uses a second set of values: the MachinePeekCount and the MachinePeekDelay. If this operation fails, the error "Station not present" is generated. The reason for this is that stations must respond to MachinePeek. You can therefore determine quite quickly if the destination machine is actually present on the network, without having to wait the long time required for a normal transmission to timeout and report "Station not listening".

The last value used is called the BroadcastDelay; this is the amount of time for which NetFS will wait for a file server to respond to the broadcast for names of file servers. If the named file server has not responded within that time the error "Station name not found" will be returned.

Direct access to file servers

To provide access to those functions not provided as part of the FileSwitch interface, or as one of the command interfaces provided directly by NetFS, there are a pair of SWI calls.

The first of these (SWI NetFS_DoFSOp) provides communication with the current file server, and the second (SWI NetFS_DoFSOpToGivenFS) to any file server to which the NetFS software is logged on.

- The function (in R0) is an indication to the file server what it should do. You will find documentation of the file server functions in the *Econet Advanced User Guide* (part number 412,019).
- The buffer contains the data to be sent to the file server. Econet's five byte header (Reply port, Function, URD, CSD, CSL) is prepended to the buffer during transmission. When a reception occurs Econet's two byte header is stripped off before the returned data is placed in the buffer.

NetFS_ReadFSNumber (SWI &40040)

Returns the full station number of your current file server

On entry	—
On exit	R0 = station number R1 = net number
Interrupts	Interrupt status is unaltered Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	SWI is re-entrant
Use	This call returns the full station number of your current file server.
Related SWIs	NetFS_SetFSNumber (SWI &40041), NetFS_ReadFSName (SWI &40042)
Related vectors	None

NetFS_SetFSNumber (SWI &40041)

Sets the full station number used as the current file server

On entry

R0 = station number
R1 = net number

On exit

—

Interrupts

Interrupts may be enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the full station number used by NetFS as the current file server.

Related SWIs

NetFS_ReadFSNumber (SWI &40040), NetFS_SetFSName (SWI &40043)

Related vectors

None

NetFS_ReadFSName (SWI &40042)

Reads the name of the your current file server

On entry

R1 = pointer to buffer
R2 = size of buffer in bytes

On exit

R0 = pointer to buffer
R1 = pointer to the terminating null of the string in the buffer
R2 = amount of buffer left, in bytes

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call reads the name of your current file server.

Related SWIs

NetFS_ReadFSNumber (SWI &40040), NetFS_SetFSName (SWI &40043)

Related vectors

None

NetFS_SetFSName (SWI &40043)

Sets by name the file server used as your current one

On entry

R0 = pointer to buffer

On exit

—

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets by name the file server used as your current one.

Related SWIs

NetFS_SetFSNumber (SWI &40041), NetFS_ReadFSName (SWI &40042)

Related vectors

None

NetFS_ReadCurrentContext (SWI &40044)

Unimplemented

On entry	—
On exit	R0 - R2 corrupted
Interrupts	Interrupt status is unaltered Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	SWI is re-entrant
Use	This call is unimplemented, and returns immediately to the caller.
Related SWIs	NetFS_SetCurrentContext (SWI &40045)
Related vectors	None

NetFS_SetCurrent Context (SWI &40045)

Unimplemented

On entry	—
On exit	All registers preserved
Interrupts	Interrupt status is unaltered Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	SWI is re-entrant
Use	This call is unimplemented, and returns immediately to the caller, with all registers preserved.
Related SWIs	NetFS_ReadCurrentContext (SWI &40044)
Related vectors	None

NetFS_ReadFSTimeouts (SWI &40046)

Reads the current values for timeouts used by NetFS

On entry

—

On exit

R0 = transmit count
R1 = transmit delay in centiseconds
R2 = machine peek count
R3 = machine peek delay in centiseconds
R4 = receive delay in centiseconds
R5 = broadcast delay in centiseconds

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call reads the current values for timeouts used by NetFS when communicating with the file server.

Related SWIs

NetFS_SetFSTimeouts (SWI &40047)

Related vectors

None

NetFS_SetFSTimeouts (SWI &40047)

Sets the current values for timeouts used by NetFS

On entry

R0 = transmit count
R1 = transmit delay in centiseconds
R2 = machine peek count
R3 = machine peek delay in centiseconds
R4 = receive delay in centiseconds
R5 = broadcast delay in centiseconds

On exit

—

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

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

Related SWIs

NetFS_ReadFSTimeouts (SWI &40046)

Related vectors

None

NetFS_DoFSOp (SWI &40048)

Commands the current file server to perform an operation

On entry

R0 = file server function
R1 = pointer to buffer
R2 = number of bytes to send to file server from buffer
R3 = size of buffer in bytes

On exit

R0 = return condition given by file server
R3 = number of bytes placed in buffer by file server

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call commands the file server to perform an operation, as specified by the file server function passed in R0. For further details of these functions, the data they need to be passed in the buffer, and the data they return in the buffer, you should see the *Econet Advanced User Guide* or the documentation for your file server.

The buffer must be large enough to hold the data that the file server returns.

Errors returned by the file server are copied into NetFS's workspace and adjusted to be like a normal RISC OS error – R0 points to the error, and the V bit is set. Any further use of NetFS may overwrite this error, so you should copy it into your own workspace before you call NetFS again, either directly or indirectly. (For example, character input or output may call NetFS, as you may be using an exec or spool file.)

Related SWIs

NetFS_DoFSOpToGivenFS (SWI &4004C)

Related vectors

None

NetFS_EnumerateFSList (SWI &40049)

Lists all file servers to which the NetFS software is currently logged on

On entry

R0 = offset of first item to read in file server list
R1 = pointer to buffer
R2 = size of buffer in bytes
R3 = number of file server names to read from list

On exit

R0 = offset of next item to read (–1 if finished)
R3 = number of file server names read

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call lists all the entries in the list of file servers to which the NetFS software is currently logged on. This is the same as the list you would get by using the `*FS` command with no parameters.

The entries are returned as 20 byte blocks in the buffer:

Offset	Contents
0	Station number
1	Network number
2	Zero
3	Disc name
19	Zero

The order of the list is not significant, save that if you are logged on to your current file server it will be returned last.

Related SWIs

NetFS_EnumerateFS (SWI &4004A)

Related vectors

None

NetFS_EnumerateFSCache (SWI &4004A)

Lists all file servers of which the NetFS software currently knows

On entry

R0 = offset of first item to read in file server list
R1 = pointer to buffer
R2 = size of buffer in bytes
R3 = number of file server names to read from list

On exit

R0 = offset of next item to read (-1 if finished)
R3 = number of file server names read

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call lists all the entries in a list of file servers which the NetFS software holds internally. This list is used by the NetFS software to resolve file server names, and is the same as the list you would get by using the `*ListFS` command.

The entries are returned as 20 byte blocks in the buffer:

Offset	Contents
0	Station number
1	Network number
2	Drive number
3	Disc name
19	Zero

They are returned in alphabetical order.

Related SWIs

NetFS_EnumerateFSList (SWI &40049)

Related vectors

None

NetFS_ConvertDate (SWI &4004B)

Converts a file server time and date to a RISC OS time and date

On entry

R0 = pointer to file server format time and date (5 bytes)
R1 = pointer to 5 byte buffer

On exit

R1 is preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call converts a file server format time and date to a time and date in the internal format used by RISC OS (centiseconds since 00:00:00 on 1/1/1900).

The file server format is:

Byte	Bits	Meaning
0	0 - 4	Day of month (1 - 31)
	5 - 7	High bits of year (offset from 1980, 0 - 127)
1	0 - 3	Month of year (1 - 12)
	4 - 7	Low bits of year (offset from 1980, 0 - 127)
2	0 - 4	Hours (0 - 23)
	5 - 7	Unused
3	0 - 5	Minutes (0 - 59)
	6, 7	Unused
4	0 - 5	Seconds (0 - 59)
	6, 7	Unused

Related SWIs

OS_ConvertStandardDateAndTime (SWI &C0),
OS_ConvertDateAndTime (SWI &C1)

Related vectors

None

NetFS_DoFSOpToGivenFS (SWI &4004C)

Commands a given file server to perform an operation

On entry

R0 = file server function
R1 = pointer to buffer
R2 = number of bytes to send to file server from buffer
R3 = size of buffer in bytes
R4 = station number
R5 = network number

On exit

R0 = return condition given by file server
R3 = number of bytes placed in buffer by file server

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call commands the given file server to perform an operation, as specified by the file server function passed in R0. For further details of these functions, the data they need to be passed in the buffer, and the data they return in the buffer, you should see the *Econet Advanced User Guide* or the documentation for your file server.

The buffer must be large enough to hold the data that the file server returns.

Errors returned by the file server are copied into NetFS's workspace and adjusted to be like a normal RISC OS error – R0 points to the error, and the V bit is set. Any further use of NetFS may overwrite this error, so you should copy it into your own workspace before you call NetFS again, either directly or indirectly. (For example, character input or output may call NetFS, as you may be using an exec or spool file.)

Related SWIs

NetFS_DoFSOp (SWI &40048)

Related vectors

None

*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 your use of a file server, closing all open files and directories.

Example

```
*Bye 49.254
```

```
*Bye :fs
```

Related commands

*Shutdown, *Shut, *Logon

*Configure FS

Sets the default file server for the Network Filing System.

Syntax

```
*Configure FS <nnn>.<sss> | <name>
```

Parameters

<nnn>	network number
<sss>	station number
<name>	station name

Use

*Configure FS sets the network file server 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.

This option can also be set from the desktop, using the Configure application.

Example

```
*Configure FS Server1
```

Related commands

*Configure PS, *Configure FileSystem, *I Am, *Logon.

*Configure Lib

Defines how the library is selected by NetFS after logon.

Syntax

```
*Configure Lib [0 | 1]
```

Parameters

0 or 1

Use

*Configure Lib defines how the library is selected by NetFS after logon.

When NetFS logs on to a file server, the file server searches for $.Library on drives 0 - <maxdrive> of the file server, in that order. It passes the first match back to NetFS as the library to be used. If it does not match this directory then it instead passes back $ on the lowest numbered physical disc.

- If 0 is used as the parameter, then NetFS uses the library directory returned by the file server.
- If 1 is used as the parameter, then NetFS searches for $.ArthurLib on drives 0 - <maxdrive> of the file server, in that order. The first match is used by NetFS as the library. If it does not find a match, then it uses the library directory returned by the file server.

Example

```
*Configure Lib 0
```

*Free

Displays file server free space.

Syntax

```
*Free [:<file server>] [<user name>]
```

Parameters

`<file server>`	file server name or number – defaults to the current file server
`<user name>`	issued by the network manager

Use

*Free displays your current remaining free space as well as the total free space for the disc. If a user name is given, the free space belonging to that user name is printed out. If no user is given, then the current user's free space is displayed.

Example

```
*Free :Business William
Disc name       Drive  Bytes free
                       Bytes used

Business            0     3 438 592
                         30 967 808
---------------------------------
User free space             185 007
```

*FS

Restores the file server's previous context.

Syntax

```
*FS [[:]<file server>]
```

Parameters

`<file server>` the file server name or number – defaults to the current file server

Use

*FS changes the currently selected file server, restoring your previous context (for example, the current directory on that file server). If no argument is supplied, your current file server number and name are printed out, followed by any non-current context.

Example

```
*FS 49.254
```

```
*FS :myFS
```

```
*FS
 59.254 "Business"
 4.254 "Accounts"
```

Related commands

*ListFS

*I am

Identical to *Logon (see below), except that *I am first selects the NetFS. There is therefore no need to type *Net before the *I am command.

*ListFS

Lists available file servers.

Syntax

```
*ListFS
```

Use

*ListFS displays a list of the file servers which NetFS is able to recognise.

Example

```
*ListFS
 1.254  :0  Finance1
 1.254  :1  Finance2
 6.246  :0  Production
```

Related commands

*FS

*Logon

Logs you on to a network file server.

Syntax

```
*Logon [[:]<station number>|:<file server name>]
       <user name> [[:<CR>] <password>]
```

Parameters

`<station number>`	a number specifying the station you wish to log on to
`<file server name>`	the name of the station you want to log on to
`<user name>`	issued by the network manager
`<password>`	controlled by user

Use

*Logon enables you to use a file server. If you give neither a station number nor a file server name, then this command logs you on to the current file server. Your user name and password are checked by the file server against the password file, to allow access. You must select NetFS before typing *Logon (this is not necessary with the *I am command).

Example

```
*Net
*Logon :fs guest
```

Related commands

*I am

*Mount

Selects the user root directory, the current selected directory and the library.

Syntax

```
*Mount [:]<disc spec>
```

Parameters

`<disc spec>` the name of the disc to be mounted

Use

Within NetFS, *Mount enables you to set a user root directory on a specified disc, as well as your currently selected directory and library.

Example

```
*Mount fs
```

Related commands

*SDisc is a synonym. It is not possible to dismount (*Dismount) a network file server.

*Net

Selects the Network Filing System as the current filing system.

Syntax

```
*Net
```

Related commands

*ADFS, *RAM

*Pass

Allows you to change your password.

Syntax

```
*Pass [<old password> [<new password>]]
```

Parameters

`<old password>`	the user's existing password (if any)
`<new password>`	the new password (if any) that the user wishes to assign

Use

*Pass enables you to change the password, knowledge of which allows unrestricted access to your network files. 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 either of the prompts. A password may not be more than six characters long.

Examples

```
*Pass
```

`Old password: ----`	User types pail.
`New password: ------`	User types bucket.
`*Pass bucket`	User enters command again, this time giving existing password as parameter.
`New password:`	User presses Return, leaving himself with no password.

*SDisc

Selects the user root directory, the current selected directory and the library.

Syntax

```
*SDisc [:]<disc spec>
```

Parameters

`<disc spec>` the name of the disc to be mounted

Use

Within NetFS, *SDisc enables you to set a user root directory on a specified disc, as well as your currently selected directory and library.

Example

```
*SDisc fs
```

Related commands

*Mount is a synonym. It is not possible to dismount (*Dismount) a network file server.

Example program

The following program fragments are examples of how you might use file server operations by calling NetFS_DoFSOp:

```
ReadFileServerVersion
        MOV     r0, #25                 ; Command
        ADR     r1, Buffer
        MOV     r2, 0                   ; Nothing to send
        MOV     r3, #(?Buffer - 1)      ; Lots to receive
        SWI     XNetFS_DoFSOp
        BVS     Error
        MOV     r0, #0                  ; Terminate string returned
        STRB    r0, [ r1, r3 ]          ; One byte past the return size
        MOV     r0, r1
        SWI     XOS_Write0              ; Print it
        BVS     Error


PrintStationNumberOfUser                ; User name pointed to by R0
        ADR     r1, Buffer
        MOV     r2, #0                  ; Initial value of index
Loop    LDRB    r3, [ r0 ], #1
        CMP     r3, #" "                ; Check for termination
        MOVLT   r3, #13                 ; Translate to what the FS wants
        STRB    r3, [ r1, r2 ]          ; Copy into transmit buffer
        ADD     r2, r2, #1              ; Update index, and size to send
        BGT     Loop
        MOV     r0, #24                 ; Command
        MOV     r3, #?Buffer
        SWI     XNetFS_DoFSOp
        BVS     Error
        LDRB    r3, [ r1, #1 ]          ; Pickup station number
        LDRB    r4, [ r1, #2 ]          ; Pickup network number
        STMFD   r13!, { r3, r4}         ; Deposit in stack frame
        MOV     r0, r13                 ; Pointer to value for conversion
        MOV     r2, #?Buffer            ; Destination size
        SWI     XOS_ConvertNetStation
        ADD     r13, r13, #8            ; Dispose stack frame
        SWIVC   XOS_Write0              ; Display output
        SWIVC   XOS_NewLine
        BVS     Error
```

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 Filewitch, and converts them to printer server commands. These commands are then sent to the printer server using the standard protocol of Econet. The printer server then acts on the commands and files that it is sent. It handles their spooling, and manages its (locally) connected printer.

Much of the above is transparent to the user, and in general to use printer servers you do not need to know printer server protocols, or how data is sent over the Econet. If you do need to know more about printer server and Econet protocols, you should see:

- the chapter entitled *Econet*
- the *Econet Advanced User Guide*, available from your Acorn supplier

Technical Details

Naming

The network printing system is actually a filing system, and as such you can use it by giving its name as part of a file name. For example:

```
*Save NetPrint:Fred A000 +14C3
```

However, with current implementations the file name is ignored, and the 'NetPrint:' part is used to send the data to the network printer. As well as save operations, the NetPrint filing system can also open files and take data. This means that the operating system can spool to NetPrint:. This is discussed in more detail in the chapter entitled *System devices*.

The current printer server

Whenever you open or save a file onto NetPrint: the current printer server is used. This printer server has a default value which is stored in CMOS RAM, and you can set the current value using a star command. You can also override the current value by supplying the printer server number as part of the file name. For example:

```
NetPrint#234:
```

This example would send the print to the printer server at station 234. As usual you can specify a full network number. For example:

```
Netprint#2.235:
```

Also, since printer servers can be named, you can supply the printer name rather than the number. For example:

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

Operations supported

The NetPrint filing system supports the OS_File Save operation and the OS_Find OpenOut operation, as well as OS_BPut and OS_GBPB writes (but not backwards).

Linking NetPrint to *FX 5 4 and VDU 2

There are system variables that connect the VDU print streams to files; an example of this is the default value set up by NetPrint upon its initialisation. This is `PrinterType$4`, and its value is `NetPrint:`. You could change this value to indicate a particular printer:

```
NetPrint#Epson:
```

and set up another variable to contain a different value:

```
PrinterType$3 = NetPrint#2.235
```

so that you can swap between printers with a *FX command. For example:

```
*FX 5 4
*FX 5 3
```

Timeouts

The dynamics of communication are controlled by several timeouts.

The values used by NetPrint for the TransmitCount, TransmitDelay, and ReceiveDelay are more fully explained in the chapter entitled *Econet*. These are the values used for all normal communication with the printer server.

Before attempting to connect to a printer server, NetPrint tries the immediate operation MachinePeek to the printer server. This operation uses a second set of values: the MachinePeekCount and the MachinePeekDelay. If this operation fails, the error "Station not present" is generated. The reason for this is that stations must respond to MachinePeek. You can therefore determine quite quickly if the destination machine is actually present on the network, without having to wait the long time required for a normal transmission to timeout and report "Station not listening".

The last value used is called the BroadcastDelay; this is the amount of time for which NetPrint will wait for a printer server to respond to the broadcast with the name of the printer server. If the named printer server has not responded within that time the error "No free printer server of this type" will be returned.

NetPrint_ReadPSNumber (SWI &40200)

Returns the full station number of your current printer server

On entry

—

On exit

R0 = station number
R1 = net number

Interrupts

Interrupts status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call returns the full station number of your current printer server.

Related SWIs

NetPrint_SetPSNumber (SWI &40201),
NetPrint_ReadPSName (SW &40202)

Related vectors

None

NetPrint_SetPSNumber (SWI &40201)

Sets the full station number used as the current printer server

On entry

R0 = station number
R1 = net number

On exit

—

Interrupts

Interrupts may be enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the full station number used by NetPrint as your current printer server.

Related SWIs

NetPrint_ReadPSNumber (SWI &40200),
NetPrint_SetPSName (SWI &40203)

Related vectors

None

NetPrint_ReadPSName (SWI &40202)

Reads the name of your current printer server

On entry

R1 = pointer to buffer
R2 = size of buffer in bytes

On exit

R0 = pointer to buffer
R1 = pointer to the terminating null of the string in the buffer
R2 = amount of buffer left, in bytes

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call reads the name of your current printer server.

Related SWIs

NetPrint_ReadPSNumber (SWI &40200),
NetPrint_SetPSName (SWI &40203)

Related vectors

None

NetPrint_SetPSName (SWI &40203)

Sets by name the printer server used as your current one

On entry

R0 = pointer to buffer

On exit

—

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets by name the printer server used as your current one.

Related SWIs

NetPrint_SetPSNumber (SWI &40201),
NetPrint_ReadPSName (SWI &40202)

Related vectors

None

NetPrint_ReadPSTimeouts (SWI &40204)

Reads the current values for timeouts used by NetPrint

On entry

—

On exit

R0 = transmit count
R1 = transmit delay in centiseconds
R2 = machine peek count
R3 = machine peek delay in centiseconds
R4 = receive delay in centiseconds
R5 = broadcast delay in centiseconds

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call reads the current values for timeouts used by NetPrint when communicating with the printer server.

Related SWIs

NetPrint_SetPSTimeouts (SWI &40205)

Related vectors

None

NetPrint_SetPSTimeouts (SWI &40205)

Sets the current values for timeouts used by NetPrint

On entry

R0 = transmit count
R1 = transmit delay in centiseconds
R2 = machine peek count
R3 = machine peek delay in centiseconds
R4 = receive delay in centiseconds
R5 = broadcast delay in centiseconds

On exit

—

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call sets the current values for timeouts used by NetPrint when communicating with the printer server.

Related SWIs

NetPrint_ReadPSTimeouts (SWI &40204)

Related vectors

None

*Configure PS

Selects the network printer server number or name at power on.

Syntax

```
*Configure PS <nnn>.<sss> | <name>
```

Parameters

<nnn>	network number
<sss>	station number
<name>	station name

Use

*Configure PS selects the number or name of the printer server within the NetFS.

You do not need to be logged on to a file server to use a printer server.

Example

```
*Configure PS Laser1
```

*PS

Changes the printer server to be used.

Syntax

```
*PS <printer server>
```

Parameters

`<printer server>` the name or station number of the printer server

Use

*PS changes the currently selected printer server, checking that the new one exists. The new printer server will be used next time you print to the net printer.

Example

```
*PS 49.254
```

```
*PS myPS
```

Related commands

*SetPS, *Configure PS

*SetPS

Changes the printer server to be used.

Syntax

```
*SetPS <printer server>
```

Parameters

`<printer server>` the name or station number of the printer server.

Use

*SetPS changes the currently selected printer server. This command only changes the stored name or number of the current printer server. No check is made that the printer server exists, or is available, until the next time you print to the network printer. It is only then that an error might be generated.

Example

```
*SetPS 49.254
```

```
*SetPS myPS
```

Related commands

*PS, *Configure PS

DeskFS

Introduction

DeskFS is a ROM based filing system that provides system resources for the Desktop. The Desktop uses the system variable Wimp$Path to find these system resources; by default its value is `DeskFS:` . You can change where the Desktop looks for these system resources by changing the value of Wimp$Path.

DeskFS provides a single * Command to select the filing system, described overleaf.

*DeskFS

Selects the desktop filing system

Syntax

```
*DeskFS
```

Parameters

None

Use

*DeskFS selects the desktop filing system. This is a ROM based filing system used to store system resources for the Desktop module.

Example

```
*DeskFS
```

Related commands

*Ram, *ADFS, *Net

Related SWIs

None

Related vectors

None

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
`serial:`	the serial port
`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
`serial:`	the serial port
`printer:`	the printer
`netprint:`	the network printer driver (provided by the NetPrint module)
`null:`	the 'null device', which swallows all output

An error is given if the specified system device is not present; for example, if the SystemDevices module is not present..

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: or serial:; the program reading them 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: sytem device is more sophisticated that other ones. As well as using it in place of file names, you can also use it with certain commands that normally use the name of a filing system. For example, to display the currently selected network printer and a list of available network printers.

```
*Cat netprint:
```

printer:

The printer: device allows various special fields, to refer to the different types of printers. These are:

- printer#sink: and printer#null: are synonyms
- printer#parallel: and printer#centronics: are synonyms
- printer#serial: and printer#rs423: are synonyms
- printer#user:
- printer#<n>: refers to printer type <n> , where <n> is in the range 0 - 255.

You can only open one file on printer: at once; if you try to open a second one FileSwitch returns a handle of 0. (Future versions of RISC OS may instead return an error.) If you try to save data to printer: as a whole file while another file is open, printer: returns an error

Other output devices

You can open as many files as you wish on the other output devices, which are:

null:, vdu:, serial: 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 rawvdu characters go straight to the VDU drivers.

Part 4 - The Window Manager

The Window Manager

Introduction

This chapter describes the Window Manager. It provides the facilities you need to write applications that work in the Desktop windowing environment that RISC OS provides.

The Window Manager is an important part of RISC OS because:

- it provides a simple to use graphical interface, that makes your applications more accessible to a wider range of users
- it also provides the means for you to make your applications run in a multi-tasking environment, so they can interact with each other, and with other software.

This chapter also gives guidelines on how your applications should behave so that they are consistent with other RISC OS applications. This should make it easier for users to learn how to use your software, as they will already be familiar with the necessary techniques.

You will find it benefits both you and other programmers if you make all your applications run under the Window Manager (and in a consistent manner), since this will lead to a much richer RISC OS environment.

Overview

w Manager is designed to simplify the task of producing programs to run under a WIMP (Windows, Icons, Menus and Pointer) environment. The manager itself is usually referred to as the Wimp. Programs that run under the Wimp are often called tasks, because they are operating under a multi-tasking environment. In this section, the words task, program and application should be treated as synonyms.

An immediately recognisable feature of Wimp programs is their use of overlapping rectangular windows on the screen. These are used to implement a 'desktop' metaphor, where the windows represent documents on a desk. The responsibility of drawing and maintaining these windows is shared between the application(s) and the Window Manager.

The Wimp co-operates with the task in keeping the screen display correct by telling the task when something needs to be redrawn. Thus, the task needs to make as few intelligent decisions as possible. It merely has to respond appropriately to the messages it receives from the Wimp, in addition to performing its own processing (using the routines supplied to perform window operations).

Very often, much of the work of keeping a window's contents up to date can be delegated to the Wimp. This is especially true if a program takes advantage of icons. An icon is a rectangular area in a window whose contents can be text, a sprite, both, or user-drawn graphics. In the first three cases, the Wimp can maintain the icon automatically, even to the point of performing text input without the application's intervention.

Menus also form an important part of WIMP-based programs. RISC OS Wimp menus are pop-up. That is, they can be made to appear when the user clicks on the appropriate mouse button – the middle Menu button. This is an alternative to the menu bar approach, where an area of the screen is dedicated to providing a fixed set of menu headers. In a multi-tasking environment, pop-up menus are much more useable. Further, they can be context-sensitive, ie the menu that pops up is appropriate to the mouse pointer position when the Menu button was pressed.

The Wimp provides support for nested menus, where one menu entry can lead to another menu, to any desired depth. Moreover, the 'leaf' of a menu structure can be a general window, not just a fixed text item. This allows for very flexible selections to be made from menus.

A very powerful feature of the RISC OS Wimp is its support for co-operative multi-tasking. Several programs can be active at once. They gain control on return from the Wimp's polling routine, which is described below. There is normally no pre-emption. Pre-emption means the removal of control from a task at arbitrary times, without its prior knowledge. With polling, a task only relinquishes control when it chooses, so for the system to work, tasks must be well behaved. This means they must not spend too much time between polling, otherwise other tasks will be prevented from running. However, it is possible to enforce pre-emption for non-Wimp tasks, by running them in for example, the edit application's task window.

To allow several applications to run at once, the Wimp must also perform memory management. This allows each application to 'see' a standard address space starting at &8000 whenever it has control. As far as a task is concerned, it is the only user of the application workspace. The amount of workspace that a task has is settable before it starts up. A program does not therefore have to be written with multi-tasking in mind. A task that does everything correctly will work whether it is the only program running, or one of several.

Communication between tasks is possible. In fact, it is often necessary, as the Task Manager sometimes needs to 'talk' to the programs it is controlling. The Wimp implements a general and very powerful message-passing scheme. Messages are used to inform tasks of such events as screen mode and palette changes, and to implement a general purpose file transfer facility.

The next section gives an overview of the major components of the RISC OS Window Manager.

Technical details

Polling

Central to any program running under the Wimp environment is its polling loop. Wimp programs are event driven. This means that instead of the program directing the user through various steps, the program waits for the user to control it. It responds to events. An event is a message sent to a task by the Wimp (or by another task). Events are usually generated in response to the user performing some action, such as clicking a mouse button, moving the pointer, selecting a menu item, etc. Inter-task ('user') messages are also passed through the polling loop.

An application calls the routine Wimp_Poll (SWI &400C7) to find out which events, if any, are pending for it. This routine returns a number giving the event type, and some event-specific information in a parameter block supplied by the caller. One event is Null_Reason_Code (0), which means nothing in particular needs to be done. The program can use this event to perform any background processing.

In very broad terms, Wimp applications will have the following (simplified) structure:

```
SYS"Wimp_Initialise"                        Tell the Wimp about the task
finished = FALSE : DIM blk 255              Get block for Wimp_Poll
REPEAT
  SYS"Wimp_Poll",0,blk TO eventCode         Get the event code to process
   CASE eventCode OF
    WHEN 0:...                              Do Null_Reason_Code
      WHEN 1:...                            Do Redraw_Window_Request
      ...                                   etc.
  ENDCASE
UNTIL finished
SYS"Wimp_CloseDown"                         Tell Wimp we've finished
```

Currently, event codes in the range 0 to 19 are returned, though not all of these are used. A fully specified Wimp program will have `WHEN` (or equivalent) routines to deal with most of them.

Some of the event types are fairly esoteric and can be ignored by many programs. It is very important that tasks do not complain about unrecognised event codes; they should simply ignore them or better, avoid receiving them in the first place.

When calling Wimp_Poll, the program can mask out certain events if it does not want to hear about them at the moment. For example, if the program doesn't need to know about the pointer leaving or entering a window, it could mask out these events. This makes the whole system more efficient, as the Wimp will not bother to pass control to a task which will simply ignore the event. Some events are unmaskable, eg an application must respond to Open_Window_Request.

As noted above, events are usually generated internally by the Wimp. However, a user task may also send messages, which result in Wimp_Poll events being generated at the destination task. For example, the Madness application moves all of the windows around the screen by sending an Open_Window_Request message to their owners. A more useful use of messages is the data transfer protocol. Most messages sent between tasks are of type User_Message_xxx (17, 18 and 19). See the entry for Wimp_SendMessage (SWI &400E7) for details of these.

All of the event types are described in the entry for Wimp_Poll in the section on **SWI Calls**, along with descriptions of how the application should respond to them.

Mouse buttons

The Wimp system works with a three-button mouse, and since it is important that all tasks use the mouse in as consistent a manner as possible, it has been decided that the buttons shall be used as follows:

lefthand button	Select
middle button	Menu
righthand button	Adjust

The interpretation of which button should do what depends on the circumstances, but broadly speaking the Select button is used to make new selections, while the Adjust button is used to alter existing selections, or to add selections to existing ones. Often, where Select performs a certain task,

Adjust under the same circumstances will perform a variation on it. See the descriptions of controls in the section Window system areas for examples of this.

Various parts of the Wimp enforce the interpretations given above for the mouse buttons. For example, icons may be programmed to respond in various ways to clicks with the the Adjust and Select buttons, by setting their button type. On the other hand, a click on the Menu button is always reported in exactly the same way, regardless of where it occurs, as a Mouse_Click event with the button state set to 2. This is to encourage all programs to interpret a click on the middle button in the same way – as a request to open a menu.

Layout of windows

Windows consist of a visible area, in which the task can draw graphics, and a surrounding 'system' area, comprising a Title Bar, scroll bar indicators and so on. The task does not normally draw directly in this area, except the Title Bar. The visible area provides a window into a larger region, called the work area. You can imagine the work area to be the complete document you are working with, and the visible area a window into this.

There are, therefore, two sets of coordinates to deal with when setting up a window. The visible area coordinates determine where the window will appear on the screen and its size. These are given in terms of OS graphics units, with the origin in its default position at the bottom left of the screen.

Then there are the work area coordinates. These give the minimum and maximum x and y coordinate of the whole document. The limits of the work area are sometimes called its extent. The work area is specified when a window is created, but can be altered using the Wimp_SetExtent (SWI &400D7) call.

Between the work area coordinates and the visible area coordinates is a final pair which join the two together. These are the scroll offsets. They indicate which part of the work area is shown by the visible area – this is called the visible work area.

The scroll offsets give the coordinates of the pixel in the work area which is displayed at the top lefthand corner of the visible region. Suppose the visible region shows the very top left of the work area. Then the x scroll position would be 'work area x min', and the y scroll position would be 'work area y max'.

It is common to define the work area such that its origin (0,0) is at the top left of the document. This means that all x scroll offsets are positive (as you can only ever be on or to the right of the work area origin), and all y offsets are zero or negative (as you can only ever be on or below the work area origin).

To summarise, let's consider which part of the work area will be visible, and where it will appear on the screen, for a typical set of coordinates.

The following give the total document size:

```
work_area_x_min = 0
work_area_y_min = -5000
work_area_x_max = 1000
work_area_y_max = 0
```

The document is therefore 1000 units wide by 5000 high, with the work area origin at the top left of the document. The following give the window's position on the screen and its size:

```
visible_area_x_min = 200
visible_area_y_min = 100
visible_area_x_max = 500
visible_area_y_max = 400
```

This gives a window 300 units wide by 300 high. It is positioned about 1/6th (200/1280) of the way across and 1/10th (100/1024) of the way up a typical 1280 by 1024 unit display.

The following determine which part of the work area is displayed:

```
scroll_offset_x = 250
scroll_offset_y = -400
```

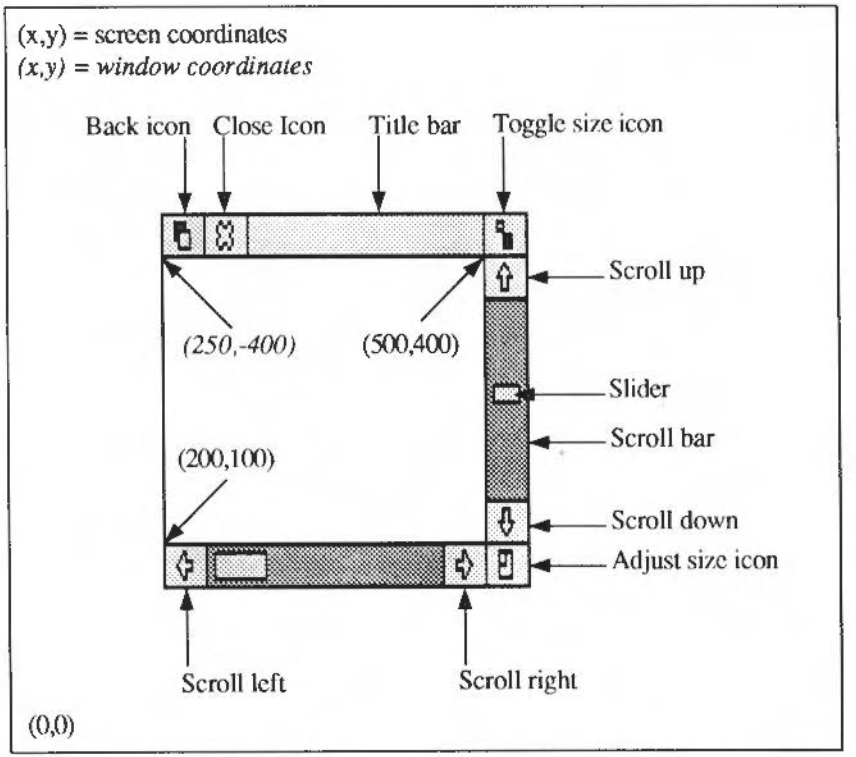
Thus the pixel at the top left of the window represents the point (250,-400) in the work area. This is shown on the screen at coordinates (200,400).

Combining the above bits of information, we can work out what portion of the work area is visible. By definition, the minimum x coordinate and the maximum y coordinate of the visible work area are just the scroll offsets. The maximum x and minimum y can then be derived by adding the width and subtracting the height respectively of the displayed window:

visible_work_area_min_x = scroll_offset_x = 250
visible_work_area_min_y = scroll_offset_y – height = –700
visible_work_area_max_x = scroll_offset_x + width = 550
visible_work_area_max_y = scroll_offset_y = –400

Thus on the screen at coordinates (200,100) - (500,400) would be a 300 pixel-square window showing the visible work area (250,–700) - (550,–400). Moreover, the Sliders drawn by the system have a length proportional to the area that the window displays. The horizontal Slider would therefore occupy about 300/1000 = 0.3 of the horizontal scroll bar, and the vertical one would occupy 300/5000 = 0.06 of the scroll bar.

All this is shown in the diagram below.



A commonly required calculation is one which gives the coordinates of a point in the work area of a window, given a screen position (eg where a mouse button click occurred). This mapping obviously depends on the window's screen position and its scroll offsets. The first step is to find the work area pixel that would be displayed at the screen origin, then add the given screen coordinates to this.

The formula below generalises this:

```
work area x = screen x + (scroll_offset_x-visible_work_area_min_x)
work area y = screen y + (scroll_offset_y-visible_work_area_max_y)
```

However, it is easier to envisage this as:

```
work area x = screen x – (visible_work_area_min_x-scroll_offset_x)
work area y = screen y – (visible_work_area_max_y-scroll_offset_y)
```

since (visible_min_x-scroll_x, visible_max_y-scroll_y) are equal to the coordinates of the origin of the work area on the screen.

Generally, when this calculation is needed, the scroll offsets and visible work area coordinates are available (eg having been returned from Wimp_Poll). Even if they are not, a call to Wimp_GetWindowState (SWI &400CB) will secure the information.

In addition to the coordinates described above, several other attributes have to be set when a window is created. These are described in detail in the entry on Wimp_CreateWindow (SWI &400C1), but are summarised below.

Windows can overlap on the screen. In order to determine which windows obscure which, the Wimp maintains 'depth' as well as positional information. We say that there is a window stack. The window at the top of the stack obscures all others that occupy the same space on the screen; the one on the bottom of the stack is obscured by any other at the same coordinates.

Certain mouse operations alter a window's depth in the stack. A click with Select on the Title Bar (see below) brings the window to the top. Similarly you can give a window a Back icon, which, when clicked on, will send the window to the bottom of the stack. On opening a window, you can determine its depth in the stack by specifying the window that it must appear behind. Alternatively you can give its depth absolutely as 'top' or 'bottom'.

There are several colours used in drawing a window. For harmonious operation with other applications, several of these have been standardised: you should set the Title Bar colours, the scroll bar inner and outer colours and highlighted title colour to the values given in the section entitled *Application notes*, unless you have some good reason not to. On the other hand, the work area colours (which are set for you before an update or redraw) can be assigned any values required.

One 32-bit word of the window block contains flags. These control many of its attributes: which control icons it should have, whether it's movable, whether Scroll_Request events should be generated etc. Another word of flags control the appearance of the Title Bar, and yet another word set the button type of the work area. Both of these are actually icon attributes, the Title Bar being treated like an icon in many ways.

Finally there are miscellaneous properties such as the sprite area address to use for icon sprites, the minimum size of the window, and the icon data for the Title Bar.

Appended to the window definition are any initial icons that it owns. Further icons can be added using the call Wimp_CreateIcon (SWI &400C2).

Window system areas

The window illustrated above has a fully defined system area – it has all of the available controls. The control areas, going clockwise from the top-left corner are described below. Where the effects of using Select and Adjust on them are different, this is noted.

Back icon

A click on this icon causes the window to be moved to the back of the window stack, making it the 'least visible' one. A Redraw_Window_Request event is issued to any applications which have windows that were obscured by it and are now visible.

Close icon

A click on this icon should cause the window to be closed, or rather the Wimp generates a Close_Window_Request event and it is then up to the application to respond with a Wimp_CloseWindow (SWI &400C6) call. (Or not, if it has good reason not to.) Using Adjust on the Close icon of a Filer window closes the directory display, but opens its parent directory. When a window is closed, the Wimp will issue Redraw_Window_Requests to those windows which were obscured by it and are now visible.

Title bar

This contains the name of the window, which is set when the window is created. Dragging the Title Bar causes the whole window to be dragged. If Select is used for the drag, the window is also brought to the top; Adjust leaves it at the same depth. The Title Bar has many of the attributes of an icon (font type, indirection, centring etc). If the whole window is being dragged (and not just its outline), each movement will generate an Open_Window_Request for it, and Redraw_Window_Requests to windows that become unobscured.

Toggle Size icon

A click in this icon toggles the window between its maximum size and the last user-set size. An Open_Window_Request event is generated to ask the application to update the work region of the resized window. The maximum size of a window depends on its work area extent and the size of the screen. Again, using Select uncovers the window; Adjust leaves it at the same place in the stack. As usual, if the change in window size renders previously obscured window visible, Redraw_Window_Requests will be generated for them. When the window is toggled back to its small size, it goes back to its previous depth in the stack.

Vertical scroll bar

Although this is one object as far as the window definition is concerned, there are five regions within it. They are: the scroll up arrow, the page up area (above the Slider), the Slider, the page down area, and the scroll down arrow.

If the user clicks on one of the arrows with Select, the scroll offset for the window is adjusted by 32 units in the appropriate direction. Using Adjust scrolls in the reverse direction. Holding down either button causes the scrolling to auto-repeat. A click in the page up/down region adjusts the scroll offsets by the height of the window work area, with Adjust again giving the reverse effect from Select. An Open_Window_Request is generated to update the scrolled window.

If the window had one of the Scroll_Request flags set when it was created, a click in one of the arrows or page up/down areas causes a Scroll_Request event to be generated instead. The application can decide how much to scroll and call Wimp_OpenWindow (SWI &400C5) to update its contents.

Finally, the Slider may be dragged to set the scroll offsets to any position in the work area. The Open_Window_Request events are returned either continuously or when the drag finishes, depending on the state of the Wimp drag configuration bits.

All scroll operations leave the window's depth unaltered.

Adjust Size icon

Dragging on this icon causes the window to be resized. The limits of the new window size are determined by the work area extent and the minimum size given when the window was created. Depending on the state of the Wimp drag configuration flags the Wimp generates either continuous Open_Window_Requests (and possibly Redraw_Window_Requests for other windows) or a single one at the end of the drag. Select brings the window to the top; Adjust leaves it at the same depth.

Horizontal scroll bar

This is exactly equivalent to the vertical scroll bar described above. For 'up' read 'right' and for 'down' read 'left'. ie. scroll up increases the y-scroll offset, while scroll right increases the x-scroll offset.

When a window is created, its control regions can be defined in one of two ways. The 'old' way is to use certain flags which specify in a limited fashion which of the regions should be present and which are omitted. The 'new' method uses one flag per control, and is much easier to use. The old way was used in Arthur, while the new is only available in RISC OS.

Redrawing windows

As mentioned above, the Wimp and the application cooperate to ensure that the windows on the screen remain up to date. The Wimp can't do all of the work, as it does not always know what the contents of a window should be.

When the task receives the reason code Redraw_Window_Request from Wimp_Poll, it should enter a loop of the following form:

```
REM blk is the Wimp_Poll block
SYS"Wimp_RedrawWindow",,blk TO flag
WHILE flag
      Redraw contents of the appropriate window
      SYS"Wimp_GetRectangle",,blk TO flag
ENDWHILE
Return to polling loop
```

When a window has to be redrawn, often only part of it needs to be updated. The Wimp splits this area into a series of non-overlapping rectangles. The rectangles are returned as x0,y0,x1,y1 where (x0,y0) is inclusive and (x1,y1) is exclusive. This applies to all boxes, eg. icons, work area, etc. The `WHILE` loop above is used to obtain all the rectangles so that they can be redrawn. The Wimp automatically sets the graphics clipping window to the rectangle to be redrawn. The task can take a simplistic view, and redraw its whole window

contents each time round the loop, relying on the graphics window to clip the unwanted parts out. Alternatively, and much more efficiently, it can inspect the graphics window coordinates (which are returned by Wimp_RedrawWindow (SWI &400C8) and Wimp_GetRectangle (SWI &400CA)) and only draw the contents of that particular region.

The areas to be redrawn are automatically cleared (to the window's background colour) by the Wimp. The task must determine what part of the workspace area is to be redrawn using the visible area coordinates and the current scroll offsets.

When redrawing a window's contents, you should normally use the overwrite GCOL action. You should use EOR mode when redrawing any currently dragged object. EOR mode is also useful when updating the window contents, such as dragging lines in Draw. As a rule, the contents of the document should not use EOR mode.

You should not use block operations such as Wimp_BlockCopy (SWI &400EB) within the redraw or update loop, only outside it to move an area of workspace. These restrictions allow you to use the same code to draw the window contents and to print the document. If you use, for example, exclusive-OR plotting or block moves during the redraw these won't work on, say, a PostScript printer driver.

Updating windows

When a task wants to update a window's contents, it must not simply update the appropriate area of the screen. This is because the task does not know which other windows overlap the one to be updated, so it could overwrite their contents. As with all window operations, it must be done with the Wimp's co-operation. There are two possible approaches. The program can:

- call Wimp_ForceRedraw (SWI &400D1) so Wimp subsequently returns a Redraw_Window_Request, or
- call Wimp_UpdateWindow, and perform appropriate operations.

In both cases, you provide the window handle and the coordinates of the rectangular area of the work area to be updated. The Wimp works out which areas of this rectangle are visible, and marks them as invalid. If you use the first method, the Wimp will subsequently return a Redraw_Window_Request from Wimp_Poll, which you should respond to as already described. In the second case, a list of rectangles to be redrawn is returned immediately.

When Wimp_ForceRedraw is used, the Wimp clears the update area automatically. This should therefore be used when a permanent change has occurred in the window's contents, eg a paragraph has been reformatted in an editor. When you call Wimp_UpdateWindow (SWI &400C9), no such clearing takes place. This makes this call more suitable for temporary changes to the window, eg when dragging objects or 'rubber-banding' in graphics programs.

It is simpler to use Wimp_ForceRedraw since, once it has been called, the task just returns to the central loop, from where the Redraw_Window_Request will be received. The code to handle this must already be present for the program to work at all. On the other hand, the second method is much quicker as the redrawing is performed immediately. Also, you can keep the original contents, using EOR to update part of the rectangle; for example, when dragging a line.

Icons and sprites

As mentioned earlier, an icon is a rectangular area of a window's workspace. Icons can be created at the same time as a window, by appending their definitions to a window block. Alternatively, you can create new icons as needed by calling Wimp_CreateIcon. A third possibility is to plot 'virtual' icons during a redraw or update loop using Wimp_PlotIcon (SWI &400E2). The advantage of this last technique is that the icons plotted don't occupy permanent storage.

Icons have handles that are unique within their parent window. Thus an icon is totally defined by a window/icon handle pair. User icon handles start from zero; the system areas of windows have negative icon numbers when returned by Wimp_GetPointerInfo (SWI &400CF).

The contents of an icon can be anything that the programmer desires. The Wimp provides a lot of help with this. It will perform automatic redrawing of icons whose contents are text strings, sprites, or both. Moreover, text icons can be writeable, that is, the Wimp will deal with user input to the icon, and also handle certain editing functions such as Delete and left and right cursor movements.

Below is an overview of the information supplied when the program defines an icon. For a detailed description, see Wimp_CreateIcon (SWI &400C2) below.

Bounding box

Four coordinates define the rectangle that the icon occupies in the window's workspace. The Wimp uses this region when detecting mouse clicks or movements over the icon, when filling the icon background (if any) and drawing the icon border (if any).

Icon flags

This single word contains much of the information that make icon handling so flexible. It indicates:

- whether the icon contains text, a sprite, or both
- for text icons, the text colours, whether the font is anti-aliased or not (and the font handle or colours), and the alignment of text within the font bounding box
- for sprite icons, whether to draw the icon half size
- whether the icon has a border and/or a filled background
- whether the application has to help redraw the icon's contents
- whether the icon is indirected
- the button type of the icon
- the exclusive selection group (ESG) of the icon, and how to handle Adjust-type selections of this icon
- whether to shade the icon so that it can't be selected.

Indirected icons use the last twelve bytes of the icon definition in a different way from non-indirected ones; see below.

The button type of an icon determines how the Wimp will deal with mouse movements and clicks over the icon. There are 16 possible types. Examples are: ignore all movements/clicks; report single clicks, double clicks and drags; select the icon on a single click; make the icon writeable, and so on.

When Select is used to select an icon, its selected bit is set regardless of its previous state, and it is highlighted. When Adjust is used, its selected bit is toggled, de-selecting it if it was previously highlighted, and vice versa.

When an icon is selected, the Wimp indicates this visually by inverting the colours that are used to draw its text and/or sprite. Selecting an icon causes all other icons in its exclusive selection group to be de-selected. The ESG is

in the range 0 to 31. Zero is special; this puts the icon in a group of its own, so selecting the icon will not affect any other icons, but each selection actually toggles its state.

Imagine a window has three icons with ESG=1. Only one of these can be selected at once: the selection (or toggling by Adjust) of one automatically cancels the other two. However, if the icon has its adjust bit set, then using Adjust to toggle the icon's state will not have any affect on the other icons in the same ESG.

When the icon's shaded bit is set, the Wimp draws the icon in a 'subdued' way, to indicate that it can't be selected. This also prevents selection by clicking.

Icon flags occur in other contexts. A window definition uses the button type bits to determine its work area's button type. The rest of the bits (with some restrictions) are used to determine the appearance of a window's Title Bar. Finally menu items have icon flags to determine their appearance.

Icon data

The last 12 bytes of an icon definition are used in two different ways. If the icon is not indirected, these are used to hold a 12-byte text string. This is the text to be displayed for a text icon, the name of the sprite for a sprite icon, and both of these things for a text and sprite icon. Clearly the last is not very useful; it is unlikely that you will want to display an icon called `sm!arcpaint` along with the text `sm!arcpaint`.

If the icon button type is writeable, clicking on the icon will position the caret at the nearest character and you can type into the icon, modifying the 12-byte text.

Indirected icons overcome the limitations of standard icons. Text can be more than 12 bytes long; the sprite in a text plus sprite icon can have a different name from the text displayed; sprite-only indirected icons can have a different sprite area pointer from their window; writeable icons can have validation strings defining the acceptable characters, and anti-aliased text can have colours other than the default white foreground/black background.

The twelve data bytes of an indirected icon are interpreted as three words: a pointer to the icon text or icon sprite, a pointer to the validation string or sprite control block, and the maximum length of the icon text.

Icon sprites

The sprites that are used in icons can come from any source: the system sprite pool, the Wimp sprite pool, or a totally independent user area. The use of the system sprites is not recommended as certain operations (such as scaling and colour translation) can't be performed on them. Wimp sprites are useful for obtaining standard shapes without duplicating them for each application. User sprites are used when private sprites are required that aren't available in the Wimp sprite area.

The Wimp sprite area is accessed by specifying a sprite area control block pointer of +1 in a window definition or indirected icon data word. There are actually two parts to the area, a permanent part held in ROM, and a transient, expandable area held in the RMA. The call Wimp_SpriteOp (SWI &400E9) allows automatic access to Wimp sprites by name. This is read-only access. The only operation allowed on Wimp sprites that changes them is the MergeSpriteFile reason code (11), or the equivalent `*IconSprites` command. These add further sprites to the Wimp area, expanding the RMA if necessary.

Below is a BASIC program to save the ROM sprites to a file. You can then use Paint to examine the sprites it contains.

```
SYS "Wimp_BaseOfSprites" TO rom
SYS "OS_SpriteOp",&10C,rom,"WSprites"
```

Amongst the 47 (RISC OS 2.00) ROM-based sprites are standard file-type icons (and half size versions of most of them), standard icon bar devices (printers, disk drives etc), common button types (radio buttons, option buttons) and the default pointer shape.

Pop-up menus

The Wimp provides a way in which a task can define multi-level menu structures. By multi-level we mean that a menu item may have a submenu. The user activates this by moving the pointer over the right-arrow that indicates a submenu. The new menu is opened automatically, the Wimp keeping track of the 'selection so far'.

The application usually activates a menu by calling Wimp_CreateMenu (SWI &400D4) in response to a Mouse_Click event of the appropriate type. It passes a pointer to a data structure that describes the list of menu items. Each of those items contains a pointer to its submenu, if required.

The click of the Menu button while the pointer is over a window is always reported, regardless of button types. You can use the window and icon handles to create a menu which accords to the context of the click. For example, the Filer varies its menu according to the current file selection (or pointer position if there is none).

When the user makes his or her menu choice by clicking on any of the mouse buttons while over an item, another event, Menu_Selection, is generated. The application responds to this by decoding the selected menu item(s) and performing appropriate actions.

Because menus can have a complex hierarchical structure (as opposed to the simple single level menus on some systems) a call Wimp_DecodeMenu (SWI &400D5) is provided to help translate the selection made into a textual form.

Just as icons can be made writeable, menu items can have that property too. This makes it very easy to obtain input from the user while a menu is open.

Menus are not restricted to text-only items. A leaf item (ie the last in a chain of selections) may be a window, which in turn contains a complete dialogue box. And of course, such windows can have as many icons as required, displaying sprites, text prompts, writeable icon fields etc.

It could be annoying that choosing an item from deep within a menu structure causes the whole menu to disappear. For example, the user might be experimenting with different selections from a colour menu, and he doesn't necessarily want to perform the whole menu operation again each time he clicks the mouse. To overcome this, selections made using the Adjust button do not cancel the menu. The Wimp supports this directly, but needs some co-operation from the application to make it work.

Finally, because the Wimp can inform a task when a submenu is being opened, the menu tree can be built dynamically, according to the selections that have gone before.

Dialogue boxes

There is no direct way of setting up 'dialogue' boxes under the Wimp. However, because icons can be handled in very versatile ways, it is quite straightforward to set up windows which act as dialogue boxes. The Wimp can be made to deal with button clicks within the window, for example automatically highlighting icons.

Another feature of the Wimp which is useful in dialogue boxes is exclusive selection groups mentioned above, where a highlighted icon is automatically de-highlighted if another icon from the same group is selected. This provides a 'radio button' facility, like the waveband selector on some radios.

Also, because writeable icons are available, it is a simple matter to input text supplied by the user, again with the Wimp doing most of the work. If required, the task can restrict the movement of the mouse to within the dialogue box, by defining a mouse rectangle (using the pointer OS_Word &15 described in the chapter entitled *VDU drivers*) which encloses the box. This ensures that the user can perform no other task until he or she responds to the dialogue box. The task should always reset the mouse rectangle to the whole screen once the dialogue is over. Also, open_window_requests for the dialogue box should cause the box to be reset. Note that usually the pointer is not restricted. The dialogue box is deleted if you click outside it.

Keyboard input and text handling

A task running under the Wimp should perform all of its input using the Wimp_Poll routine, rather than calling OS_ReadC or OS_Byte &81 directly. It is permissible for a program to scan the keyboard using the –ve inkey OS_Bytes. Further details are given in the chapter entitled *Character Output*.

The input focus

One window has what is termed the 'input focus'. For example, the main text window of an editor might be the current input window, and its system area is highlighted by the Wimp to show this. (A flag can also be read by the program to see if it has the input focus.) The input window or icon also has a caret (vertical bar text cursor) to show the current input position.

A window gains the input focus if it has a writeable icon over which the user clicks with Select or Adjust. The caret is positioned and sized automatically by the Wimp in this case. It uses a height of 40 OS units for the system font.

Alternatively, the program can gain the input focus explicitly by calling Wimp_SetCaretPosition (SWI &400D2). This displays a caret of a specified height and colour at the position specified in the given window and, optionally, icon. If the icon is a writeable one, the Wimp can automatically calculate the position and height from the index into the text, if required.

Generally Wimp_SetCaretPosition is called in response to a mouse click over a window's work area. The position within the window must be calculated using the pointer position, the window's screen position, and the current scroll offsets.

Wimp_SetCaretPosition causes a couple of events to occur if the input window actually changes: Gain_Caret and Lose_Caret. This enables tasks to respond to the change in caret position (and possibly the task that owns it) by updating their window contents appropriately. This is especially true if an application is drawing its own caret and not relying on the Wimp's vertical bar. Note that the Wimp's caret is automatically maintained by the Wimp in Wimp_RedrawWindow, so you don't have to redraw it yourself.

Key presses

If the insertion point is within a writeable icon, then many key presses are handled by the Wimp. The icon text is updated, and for certain cursor keys, the caret position and index within the string are updated. Other key presses, and all keys when the input focus is not in a writeable icon, must be dealt with by the application itself.

A program gets to know about key presses through the Wimp_Poll Key_Pressed event. The data returned gives the standard caret information plus the code of the key pressed. It is up to the application to determine how the key-press is handled. There are certain standard operations for use in dialogue boxes, eg cursor down means go to the next item, but generally it will very much depend on what the application is doing.

Function and 'hot' keys

Among the keys that the Wimp cannot respond to automatically are the function keys F1 to F12. These are passed to the application as special codes with bit 8 set (ie in the range 256 - 511). If the application can deal with function keys, it should process the key press appropriately. If not, it should pass the key back to the Wimp with the call Wimp_ProcessKey (SWI &400DC).

If a function key is passed back to the Wimp in this way and the input focus belongs to a writeable icon, the Wimp will expand the function key definition and insert (as much as possible of) the string into the icon.

In general, a program should always pass back key presses it doesn't understand to the Wimp. This allows the writing of programs which are activated by 'hot keys', eg a screen dump that occurs when Print (F0) is

pressed. Keys passed to Wimp_ProcessKey are passed (through the Key_Pressed event) to tasks whose windows have the 'grab hot keys' bit set. They are called in the order they appear on the window stack, topmost first.

If a program can act on a hot key, it should perform its magic task and return via Wimp_Poll. If it doesn't recognise that particular key, it should pass it to the next grab-hot-keys window in the stack by calling Wimp_ProcessKey before it next calls Wimp_Poll.

Note that the caret position returned by Wimp_Poll is appropriate to the caret position, so it may not correspond to the window with the grab-hot-keys bit set. Also, note that all potential hot key grabbers take priority over icon soft key expansion, and that you should not process a key and hand it back to the Wimp. This could lead to user-confusion.

If the only raison d'etre of a window is to allow its creator to grab hot keys, ie if it will never appear, it should be created and opened off the screen (with a large negative x position). To allow this, its window flags bit 6 should be set.

The Escape key

One of the Wimp's start-up actions (the first time Wimp_Initialise (SWI &400C0) is called) is to make the Escape key return ASCII 27. It does this by issuing an OS_Byte with R0=229, R1=1, R2=0. Thus no Escape conditions or (RISC OS) events are normally generated. The task that has the input focus can respond to ASCII 27 in any way it wants.

If you want to allow the user to interrupt the program by pressing Escape during a long operation, you can re-enable it using OS_Byte with R0=229, R1=0, R2=0. The following restrictions must be observed. Escapes must only be enabled between calls to Wimp_Poll, ie you **must not** call that routine with Escape enabled. This is very important. If you detect an Escape, you must disable it before calling the Wimp again and then clear it using OS_Byte with R0=124.

Even if no Escape occurs, you should still disable it before you next call Wimp_Poll; it is a good idea to call OS_Byte with R0=124 just after disabling Escapes.

It is also a good idea to display the Hourglass pointer during long-winded operations, preferably with the percentage of completion if this is possible. The user is less likely to try to interrupt if they can see that the operation is progressing. Note that you should not attempt to change the pointer while the hourglass is still showing.

When Wimp_CloseDown (SWI &400DD) is called for the last time (ie. when the last task finishes), the Wimp restores the Escape key to its previous state, along with all the other settings it changed (function keys, cursor keys etc.)

Changing the pointer shape

You should not use the standard OS_Words and OS_Bytes to control the pointer shape under the Wimp. Instead, use the call Wimp_SpriteOp (SWI &400E9) with R0 = 36 (SetPointerShape). This programs the pointer shape from a sprite definition, performing scaling and colour translation if required. Pointer sprites have names of the form `ptr_xxxxx`. The standard arrow shape is held in the Wimp ROM sprite area and is called `ptr_default`.

The call Wimp_SetPointerShape (SWI &400D8) which was available before RISC OS version 2.00 should no longer be used, although it is still provided for compatibility.

Pointer shape 1 is used by the Wimp as its default arrow pointer. Any program wishing to use a different shape must use shape 2, and program the pixels appropriately using the above call. Do not use logical colour 2 in pointer sprites, as this is unavailable in very high resolution modes. Shapes 3 and 4 are used by utilities such as the Hourglass module which changes the pointer shape under interrupts. For information about the SWIs supported by this module, refer to the chapter entitled *Hourglass*.

Note that when changing the pointer shape, it is recommended that the pointer palette is also reset. This is held in the sprite. Also, each sprite should have its own palette.

A task should only change the pointer when it is within the work area of one of its windows. The Wimp_Poll routine returns two reason codes for detecting this: Pointer_Entering_Window and Pointer_Leaving_Window (5 and 4 respectively). Whenever the first code is received, the task can change the pointer to shape 2 for as long the pointer stays within the window. On receiving the second code, the task should reset the pointer to shape 1. The best way to achieve this is to use the *Pointer command.

Tasks should trap Message_ModeChange, as a mode change resets the pointer to its default shape. If, on a mode change, the task thinks that it 'owns' the pointer, ie it is over one of the task's windows, it should re-program the pointer shape, if required.

Mode independence

An important aspect of Wimp-based applications is that they do not depend for their operation on a particular screen mode. A corollary of this is that they should not explicitly change display attributes such as mode or colours. The motivation for this rule is to ensure that many separate tasks can be active without mutual interference.

To help programs operate in a consistent manner regardless of, say, the number of screen colours, the Wimp provides a variety of utility functions, such as colour translation and the scaling of sprites and text. In fact many of these features are provided by other parts of RISC OS, but are given Wimp calls to facilitate a more uniform interface.

Colour handling

The Wimp's model of the display centres on the 16-colour modes. There are 16 Wimp colours defined, listed below. In other modes, the Wimp performs a mapping between these standard colours and those which are actually available. When setting colours for graphics (including VDU 5 text), or anti-aliased fonts, the application specifies standard colours to the appropriate Wimp routine, which translates them and generates the necessary VDU calls.

Here are the standard colours, and their usages:

0 - 7	grey scale from white (0) to black (7) colour 1 is icon bar and scroll bar inner colour colour 2 is standard window title background colour colour 3 is the scroll bar outer colour colour 4 is the desktop background colour
8	dark blue
9	yellow
10	green
11	red
12	cream, window title background for input focus owner
13	army green
14	orange
15	light blue

In non-16 colour modes, these standard colours are represented as follows:

2-colour modes	logical colour 0 is set to Wimp colour 0, ie white logical colour 1 is set to Wimp colour 7, ie black
0	logical colour 0
1 - 6	decreasing brightness stippled patterns
7	logical colour 1
8 - 15	logical colour 0 or 1, whichever is closer to standard colour's brightness level
4-colour modes	logical colour 0 is set to Wimp colour 0, ie white logical colour 1 is set to Wimp colour 2, ie light grey logical colour 2 is set to Wimp colour 4, ie dark grey logical colour 3 is set to Wimp colour 7, ie black
0 - 15	set to the logical colour closest in brightness to the standard one
256-colour modes	the default palette is used
0 - 15	set to the closest colour to the standard one obtainable

As an example of the use of colour translation, if you were to set the graphics colour to 2 in a two-colour mode, using Wimp_SetColour (SWI &400E6), then the Wimp would actually set up an ECF pattern (number 4 is used) to be a lightish stippled pattern, and issue a GCOL to make ECF 4 the current graphics colour. On the other hand, in a 256-colour mode it would calculate the GCOL and TINT which gives the closest match to the standard light grey, and issue the appropriate VDUs.

In 256-colour modes, exact representations of the Wimp colours 0 - 7 (the grey scale) are available, but only approximate (albeit pretty close) representations of Wimp colours 8 - 15 can be obtained.

The Wimp utilises its colour translation mechanism in the following circumstances:

- when using the colours given in a window's definition, unless bit 10 of the window flags is set. In this case, the colour is used directly. NB in a 256-colour mode an untranslated colour is given as `%cccccctt`, ie bits 0 - 1 give bits 6 - 7 of the `TINT` and bits 2 - 7 give bits 0 - 5 of the `GCOL`.

- when using the colours in an icon's definition. Text colours are translated, except that the stippled patterns can't be used in two-colour modes. Sprites are plotted using the OS_SpriteOp PutSpriteScaled reason code with an appropriate colour table and scaling factors.
- when using the text caret colour, unless translation is overridden.

If you want to override the Wimp's translation of colours, you can use the the ColourTrans module and PutSpriteScaled to perform more sophisticated colour matching. The Draw and Paint applications do this.

System font handling

The system font is the standard 8 by 8 pixel character set. It is used by OS_WriteC text printing codes. Under the Wimp, the system font is defined to be 16 units wide by 32 OS units high. This is true regardless of the actual screen resolution. The consequence of this is that system font characters are the same physical size, independent of the screen mode.

To obtain the appropriate sizing of characters, the Wimp uses the VDU driver's ability to scale characters printed in VDU 5 mode. Thus in mode 4, where a pixel is 4 OS units wide, system font characters are only four pixels wide, to maintain their 16 OS unit width. Similarly in 512-line modes, characters are plotted double height to give them the same appearance as in mode 12.

Dragging boxes

One of the recognisable features of most window systems is the ability to 'drag' items around the screen. The RISC OS Wimp is no exception, and provides extensive facilities for dragging objects.

Icons and window work areas can be given a button type which causes the Wimp to detect drag operations automatically. A 'drag' is defined as the Select or Adjust button being pressed for longer than about 0.2s. Alternatively, if the user clicks and then moves the mouse outside the icon rectangle before releasing, this also counts as a drag. The result is that a Mouse_Click event is returned by Wimp_Poll. Note that before a drag event is generated, the application will also be informed of the initial click, and the drag could in turned be followed by a double click event, depending on the button type.

The call Wimp_DragBox (SWI &400D0) initiates a dragging operation. The user supplies the initial position and size of the box to be dragged, and a 'parent' rectangle within which the dragging must be confined. Normally, the initial position of the box will be such that the mouse pointer is positioned somewhere within the box. However, this is not mandatory; the Wimp, while performing the dragging, ensures that the relative positions of the pointer and the box remain constant.

There are two main types of drag operation: system and user. System types work on a given window, and drag its size, position or scroll offsets. These drags are normally performed automatically if the window has the appropriate control icon (eg a Title Bar to drag its position). However, you might want to allow a non-titled window to be moved, or a window without a Adjust Size icon to be resized; the system drag types cater for this sort of operation.

User drag boxes can be fixed size, where the whole of the box is moved along with the pointer, or variable sized, where the topleft of the box is fixed, and the bottom-right moves with the pointer. (The fixed and movable corners can be varied by specifying the box's top-left and bottomright coordinates in the reverse order.) The Wimp displays the drag box using dashed lines whose dash pattern changes cyclically.

There is an 'invisible' type of drag box. In this case, the mouse is simply constrained to the parent rectangle, which must be a single window, and the initial box coordinates are ignored. It is up to the task to draw the object being dragged. This usually involves setting a 'dragging' flag in the main poll loop, and the use of Wimp_UpdateWindow (SWI &400C9). The task must also ensure that the dragged object is redrawn if a Redraw_Window_Request is issued, and enable Null reason codes and use them to perform tracking.

Finally, a program can arrange for the Wimp to call its own machine code routines during dragging, for the ultimate in flexibility. This enables the program to drag any object it likes, so long as it can draw it and then remove it without affecting the background. In this case, the object can go outside the window. The Wimp will ask for it to be removed at the appropriate times.

In all cases, the task is notified when the drag operation ends (when the user releases all mouse buttons) by Wimp_Poll returning the reason code User_Drag_Box.

Tool windows and 'panes'

A pane is a window which is 'fixed' to another window, but has different properties from it. For example, consider a drawing program. You might have a scrollable, movable main window for the drawing area. This is called the tool window. On the left edge of this might be a fixed window which contains icons for the various drawing options. This lefthand window (the pane) always moves with the main window, but does not have scroll bars, or any other control areas.

Dealing with panes is really entirely up to the task program. However, there are one or two things to bear in mind when using them. If a tool window is closed, all of its panes must be closed too. Similarly, when a tool window is opened (an Open_Window_Request is received), the task must inspect the coordinates of the main window returned by the Wimp, and use them to open the pane in the appropriate position.

One bit in a window's definition is used to tell the Wimp that this is a pane. This is used by the Wimp in two circumstances:

- if the pane gets the input focus, the tool window is highlighted
- when toggling the tool window size, the Wimp must treat panes as transparent.

There are various optimisations that can be used. If you open the windows in the right order, unnecessary redraws can be avoided.

Memory management

Part of the Wimp's job is to manage the system's memory resources. There are several areas: the screen, system sprites, fonts, the RMA, application space etc. Many of these are controllable through the Task Manager's bar display. The user can drag, say, the font cache bar to set the desired size.

The remainder, when all of the other requirements have been met, is called the free pool. The Wimp can 'grab' memory from this to increase another area's size, or to start a new application, and extend it when another area is made smaller, or an application terminates. Because the allocation of memory is always under the user's control, he or she can make most of the decisions concerned with effective utilisation.

Two important bars in the Task Manager's display are the 'Free' and 'Next' ones. These give respectively the size of the free memory pool, and the amount of memory that will be given to the next application. They can be

dragged to give the desired effect. For example, the user can decrease the RAM disc slot to increase the 'Free' size, which will in turn allow another resource, eg the screen size, to be increased. This is only used if the task doesn't issue an explicit *WimpSlot command, though most will do so.

Using the memory mapping capabilities of the MEMC chip, the Wimp can make all applications' memory appear to start at address &8000. This is called logical memory, and is all the application need worry about. Logical memory is mapped via the MEMC into the physical memory of the machine. The smallest unit of mapping is called a page, and its size is typically 8K or 32K bytes. Before giving control to a task through Wimp_Poll, the Wimp ensures that the correct pages of physical memory are mapped into the application workspace at address &8000.

In general, then, the application need not concern itself with memory allocation. However, there are times when direct interaction between a task and the Wimp's allocation is desirable. For example, a program may need a certain minimum amount of memory to operate correctly. Conversely, when running an application might decide that it doesn't need all of the memory that was allocated to it, and give some back.

The SWI Wimp_SlotSize (SWI &400EC) allows the size of the current task's memory and the 'Next' slot to be read or altered. See the description of that call for details of its entry and exit parameters and examples of its use. The command *WimpSlot uses the call.

A program may need a large amount of memory for a temporary buffer. Just as it is possible to claim the screen memory using OS_ClaimScreenMemory, a program can call Wimp_ClaimFreeMemory (SWI &400EE) to obtain exclusive use of the Wimp's free pool. Only programs executing in SVC (supervisor) mode can make use of this memory, as it is protected against user-mode access. Furthermore, while the memory is claimed, the Wimp cannot dynamically alter the size of other areas, so programs should not 'hog' it for extended periods (ie across calls to Wimp_Poll).

Finally, just as built-in resources such as RMA size and sprite area size are alterable by dragging their respective bars, the Task Manager allows the user to perform the same operation on task bars. This is only possible with the task's cooperation. When a task starts up, the Task Manager asks it, by sending a message, if it will allow dynamic sizing of its memory allocation. If

the program responds, the Task Manager will allow dragging of its bar, otherwise it won't. See the Wimp_SendMessage (SWI &400E7) action code Message_SetSlot for details.

Template files

To facilitate the creation of windows, a 'template editor', called FormEd, has been written for the Wimp system. This allows you to use the mouse to design your own window layouts, and position icons as required. An extensive set of hierarchical menus provides a neat way of setting up all the relevant characteristics of the various windows and icons.

Once a window 'template' has been designed, it can be given an identifier (not necessarily the same as the window title) and saved in a template file along with any other templates which have been set up and identified. The Wimp provides a Wimp_OpenTemplate (SWI &400D9) call, which makes it very simple for a task, on start up, to load a set of window definitions. The task can load a named template from the file, which can then be passed straight to Wimp_CreateWindow (SWI &400C1), or it can look for a wildcarded name, calling Wimp_LoadTemplate (SWI &400DB) repeatedly for each match found.

Many of the templates used by the system are resident in ROM. They are held in the files DeskFS:Templates.*, where * is Filer, NetFiler, Palette, Switcher or Wimp. You can base your own templates on these by loading a DeskFS: file into the template editor (FormEd – available with Release 3 of the Acorn C Compiler), modifying it and re-saving it in your own file. For example, the palette utility template file contains the 'Save as' dialogue box, which all applications should use (with a change of sprite name).

It is also possible to override the system's use of the ROM template files by setting the Wimp$Path variable. This contains a comma-separated list of prefixes, usually directory names, in which the Wimp will search for the directory Templates when opening template files. Its default value is DeskFS:, but you could change it to, say, ADFS::MyDisc.,DeskFS: to make it look for modified, disc-resident versions of the standard template files first. Note that directory names must end in a dot.

There are two issues associated with the loading of window templates from a file. These concern the allocation of external resources:

- resolving references to indirected icons
- resolving references to anti-aliased font handles.

In the first case, what happens is that the relevant indirected icon data is saved in the template file. When the template is loaded in, the task must provide a pointer to some free workspace where the Wimp can put the data, and redirect the relevant pointers to it. The workspace pointer will be updated on exit from the call to Wimp_LoadTemplate. If there is not enough room, an error is reported (the task must also provide a pointer to the end of the workspace). Having loaded the template, the program can inspect the icon block to determine where the indirected data has been put.

The issue concerning font handles is more difficult to solve. The template file provides the binding from its internal font handles to the appropriate font names and sizes. In addition, the Wimp must also have some way of telling the task which font handles it actually bound the font references to when the template was loaded. This is so the task can call Font_LoseFont as required when the window is deleted (or alternatively, when the task terminates).

To resolve this, the task must provide a pointer to a 256-byte array of font 'reference counts' when calling Wimp_LoadTemplate. Each element must be initialised to zero before the first call. Font handles received by the Wimp when calling Font_FindFont are used as indices into the array. Element i is incremented each time font handle i is returned.

So, when Load_Template returns, the array contains a count of how many times each font handle was allocated. On closing the window or terminating, the program must scan the array and call Font_LoseFont the given number of times for non-zero entries. As with icon pointers, the program can find out the actual font handles used by examining the window block returned by Wimp_LoadTemplate.

It is up to the programmer to decide whether it is sufficient to provide just one array of font reference counts, so that the fonts can be closed only when all the windows are deleted (or the task terminates), or whether a separate array is needed for each window. Of course, considerable space optimisations could be made in the latter case if the array were scanned on exit from Wimp_LoadTemplate and converted to a more compact form.

If a task is confident that its templates do not contain references to anti-aliased fonts, then the array pointer can be null, in which case the Wimp reports an error if any font references are encountered.

Note that if anti-aliased fonts are used, the program must also rescan its fonts when Message_ModeChange is received. This involves calling Font_ReadDefn for each relevant font handle, changing to the correct xy resolution, and calling Font_FindFont again. The new font handle can be put back in the window using Wimp_SetIconState.

Relocatable module tasks

A program using the Wimp can be loaded from disc into the application memory (&8000), or may be a relocatable module resident in the RMA (relocatable module area). In the main, Wimp tasks of both varieties work in the same way and have similar structures. However, module tasks must additionally cope with service calls generated at various times by the Wimp. They must also to be able to terminate when asked to, eg during an *RMTidy operation.

In this section we describe the special requirements of module tasks, but not how to write modules from scratch. See the chapter entitled *Modules* for details. You may also like to read the sections on Wimp_Initialise (SWI &400C0) and Wimp_CloseDown (SWI &400DD) before going over the listings below.

Much of the following is concerned with service call handling. A general, and very important, aspect of this is register usage. A module service handler can modify registers R0 - R6 that have been explicitly stated to be return parameters for each individual service call. However, these registers should not be modified, except to produce a particular effect as defined below. Badly behaved service code which does not adhere to this can produce bugs which are very difficult to track down and cause the system to fail in unpredictable ways.

Task initialisation

Tasks are started using a * Command. This is decoded by the module's command table and the appropriate code to handle the command is called automatically. This is standard module code, and looks like this:

```
;This is pointed to by the entry for the module's * Command
myCommandCode
        STMFD   SP!, {LR};Save the link register
        MOV     R2, R0          ;R2 points at command tail
        ADR     R1, titleStr;R1 points at title string of module
        MOV     R0, #2          ;Module 'Enter' reason code
        SWI     XOS_Module;Enter the module as a language
        LDMFD   SP!, {PC};Return (in case that failed)
WIMP_VER * 200
titleStr
        DCB     "MyModule",0;as returned by *Modules
        ALIGN
TASK  DCB"TASK"


;This is the module's language entry point
startCode
        LDR     R12, [R12];Get workspace pointer
                                claimed in Init entry
        LDR     R0, taskHandle
        TEQ     R0, #0          ;Are we already running?
        LDRGT   R1, TASK ;Yes, so close down first
        SWIGT   XWimp_CloseDown
        MOVGT   R0, #0          ;Mark as inactive
        STRGT   R0, taskHandle
;Now claim any workspace etc. required before initing the Wimp
;...
;If all goes well, we end up here
        MOV     R0, #WIMP_VER;(re)start the task
        LDR     R1, TASK
        ADR     R2, titleStr
        SWI     XWimp_Initialise
        BVS     startupFailed;Tidy up and exit if something went wrong
        STR     R1, taskHandle;Save the non-zero handle
...
```

Thus when the user enters the appropriate * Command, the module is started as a language and the start code is called using the word at offset 0 in the module header. It is entered in user mode with interrupts enabled, and R12 pointing at its private word.

On entry, the task checks to see if it is already active. If it is, it closes down (to avoid running as two tasks at once). It also resets its `taskHandle` variable to indicate that it is inactive. It then performs any necessary pre-Wimp_Initialise code, such as claiming workspace from the RMA. If this succeeds, it calls Wimp_Initialise and saves the returned task handle.

Service Calls

The next section describes those service calls that are of particular relevance to you when you are writing modules to run under the Window Manager. The remaining service calls that RISC OS provides are documented in the chapter entitled *Modules*.

Service_Memory (Service Call &11)

Memory controller about to be remapped

On entry

R0 = current active object pointer (CAO)
R1 = Service_Memory

On exit

R1 = 0 to prevent re-mapping taking place

Use

This is issued when the contents-addressable memory in the memory controller is about to be remapped, which alters the memory map of the machine. You should claim this call if you don't want the remapping to take place.

A module will initially be given the current slot size for its application workspace starting at &8000. However, modules do not generally need this area, as they use the RMA for workspace. Therefore, when a task calls Wimp_Initialise, the Wimp inspects the CAO. If this is within application workspace, the Wimp does nothing. However, if the CAO is outside of application space (a module's CAO is its base address in the RMA or ROM), the Wimp will reduce the current slot size to zero automatically, except as described below.

Some modules, notably BASIC, do require application workspace. Therefore the Wimp makes this service call just before returning the application space to its free pool. A task can object to the remapping taking place by claiming the call. The Wimp will then leave the application space as it is.

Service_StartWimp (Service Call &49)

Start up any resident module tasks using Wimp_StartTask

On entry

R1 = Service_StartWimp

On exit

R1 = 0 to claim call
R0 = pointer to * Command to start module

Use

The Desktop will try to start up any resident module tasks when it is called (using *Desktop or by making the task the start-up language). It does this by issuing a service call Service_StartWimp (&49). If this call is claimed, the Desktop starts the task by passing the * Command returned by the module to Wimp_StartTask. It then issues the service again, and repeats this until no-one claims it.

A module's service call handler should deal with this reason code as follows:

```
serviceCode
        LDR     R12, [R12]      ;Load workspace pointer
        STMFD   SP!, {LR}       ;Save link and make R14 available
        TEQ     R1, #Service_StartWimp;Is it service &49?
        BEQ     startWimp       ;Yes
        ...                             ;Otherwise try other services
        LDMFD   SP!, {PC}       ;Return


startWimp
        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 `startWimp` 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 StartWimp service. If the task fails to start, this will still be -1 the next time the Wimp issues a Service_StartWimp, and so it will not claim the service.

Service_StartedWimp (Service Call &4A)

Service_Reset (Service Call &27)

Request to task modules to set `taskHandle` variable to zero

On entry

R1 = Service_StartedWimp or Service_Reset

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_StartWimp would never be claimed). In order to avoid this, the two service calls above should be recognised by task modules. On either of them, the task handle should be set to zero:

```
serviceCode
        STMFD   sp!, {R14}
        LDR     R12, [R12]              ;Get workspace pointer
...
        TEQ     R1, #Service_StartedWimp;Service &4A?
        BEQ     Service_StartedWimp
tryServiceReset
        TEQ     R1, #Service_Reset;Reset reason code?
        MOVEQ   R14, #0                 ;Yes, so zero handle
        STREQ   R14, taskHandle
        LDMFD   SP!, {PC}               ;Return
...

        LDR     R14, taskHandle  ;taskHandle = -1?
        CMN     R14, #1
        MOVEQ   R14, #0                 ;Yes, so zero it
        STREQ   R14, taskHandle
        LDMFD SP!, {PC}                 ;Return
```

Service_StartedWimp is issued when the last of the resident modules has been started, and Service_Reset is issued whenever the computer is soft reset.

Closing down

Generally a module task will terminate itself in the usual fashion by calling Wimp_CloseDown just before it calls OS_Exit. This might be in response to a Quit selection from a menu, or after a Message_Quit has been received. Modules also have finalisation entry point, and Wimp_CloseDown should be called from within this:

```
finalCode
        STMFD   sp!, {R14}
        LDR     R12, [R12]      ;Get workspace pointer
        LDR     R0, taskHandle;Check task is active
        TEQ     R0, #0
        LDRGT   R1, TASK        ;If so, close it down
        SWIGT   XWimp_CloseDown
        MOV     R1, #0                  ;always mark it as inactive
        STR     R1, taskHandle
;perform general finalisation code, possibly according
;to the value of R10 (fatality indicator).
        LDMFD   sp!, {PC}       ;Return with V and R0 intact in
                                ;case an error occurred
```

It is important that when Wimp_CloseDown is called from the finalise code, the task handle is quoted, as the module may not necessarily be the currently active Wimp task. Additionally, whenever Wimp_CloseDown is called, even outside of the finalisation code, the `taskHandle` variable should be cleared to zero.

Service_WimpCloseDown (Service Call &53)

Notification that the Window manager is about to close down a task

On entry

R0 = 0 if Wimp_CloseDown called (i) or
R0 > 0 if Wimp_Initialise called in task's domain (ii)
R2 = handle of task being closed down, (i) and (ii)

On exit

R0 preserved (i) or (ii), or set to error pointer (ii)

Use

The Wimp passes this service around when someone calls Wimp_CloseDown. Usually a task knows that it has called Wimp_CloseDown, so this might not appear to be particularly informative. However, there are a couple of situations where the Wimp actually makes the call on a task's behalf. It is on these occasions that the service is useful.

- If a task calls OS_Exit without having called Wimp_CloseDown first, the Wimp does so on the task's behalf. This can arise when an error is generated that is not trapped by the task's error handler. The Wimp will report the error, then call OS_Exit for the task. The task should perform the operations it would have performed if it had called Wimp_CloseDown itself, and return preserving all registers. It must not call Wimp_CloseDown.
- A task might call Wimp_Initialise from within the same domain as the currently active task. For example, if a program allows the user to issue a * Command, the user might use it to try to start another Wimp task. The Wimp will try to close down the original task before starting the new one by issuing this service with R0>0.

If the original task does not want to be closed down, it should alter R0 so that it contains the pointer to a standard error block. The text `Wimp is currently active` is regarded as a suitable message. (The task should compare the handle in R2 to its own to ensure that it is the task that is being asked to die.) The call should not be claimed, in order to allow others to receive the service, and R0 should not be altered except to point to an error.

If, on return from the service, R0 points to an error, the Wimp will return this to the new task trying to start up (it will also set the V flag). Thus, if the task is detecting errors correctly, it will abort its attempt to start up and call OS_Exit. This will happen if, for example, you try to start the Draw application from within a task window.

Service_ReportError (Service Call &57)

Request to suspend trapping of VDU output so an error can be displayed

On entry

R0 = 0 (window closing) or 1 (window opening)

On exit

—

Use

This service is provided so that certain tasks which usually trap VDU output (eg the VDU module) can be asked to suspend their activities temporarily while an error window is displayed.

If the state of the trapping module is 'active' and the service call is received with R0=1, the module should stop trapping and set its state to 'suspended'. Similarly, if the state is suspended and the service is received with R0=0, the error window has disappeared and the module should re-enter the active state.

By taking note of this call, tasks running in an Edit window allow the standard filing system 'up-call' mechanism to continue operating, whereby users are asked to insert discs which the Filer cannot find in a drive.

Service_MouseTrap (Service Call &52)

The Wimp has detected a significant mouse movement

On entry

R0 = mouse x coordinate
R1 = Service_MouseTrap
R2 = button state (from OS_Mouse)
R3 = time of mouse event (from OS_ReadMonotonicTime)
R4 = mouse y coordinate (NB R1 is already being used!)

On exit

—

Use

It is possible to write programs which record changes in the mouse button state and pointer position. The recording can be played back later to simulate the effect of a human manipulating the mouse. This is very useful for setting up unattended demonstrations.

To save memory or disc space, such programs usually only record the mouse position when the button state changes, or after a certain time interval, eg ten times a second. Some Wimp events are dependent on on a change of mouse position, not button state. It is therefore possible for a mouse recorder program to miss a critical mouse movement if it doesn't happen to choose the correct time to make its recording. The replay will then give different results from the original.

Service_MouseTrap is designed to overcome the problem. Whenever the Wimp detects a significant mouse movement, eg the pointer moving over a submenu right arrow, it issues this call. A mouse recorder should include the data in its output, in addition to any other mouse movements and button events that it would ordinarily log.

Programs which react to particular mouse movements (eg certain types of dragging) should themselves generate this event, where there is no mouse button transition.

A mouse recorder program should also trap INKEY of positive and negative numbers.

Service_StartFiler (Service Call &4B)

Request to filing system modules to start up

On entry

R0 = Filer's `taskHandle`
R1 = Service_StartFiler

On exit

R1 = 0 to claim call
R0 = pointer to * Command to start module

Use

In order to ensure that filing system modules are not started up without the Filer module, they are started by a different mechanism. Rather than responding to the Service_StartWimp service call, they wait for the Filer module to start them up, using Service_StartFiler. The Filer behaves in a similar way to the Desktop, issuing the Service_StartFiler service call, followed by Wimp_StartTask, if the service call is claimed.

The Filer will try to start up any resident filing system module tasks when it is started (by responding to Service_StartWimp). It does this by issuing a service call Service_StartFiler (&4B).

If this call is claimed, the Filer starts the task by passing the * Command returned by the module to Wimp_StartTask. It then issues the service again, and repeats this until no-one claims it.

A module's service call handler should deal with this reason code as follows:

```
serviceCode
        LDR     R12, [R12]          ;Load workspace pointer
        STMFD   SP!, {LR}           ;Save link and make R14 available
        TEQ     R1, #Service_StartFiler;Is it service &4B?
        BEQ     startFiler          ;Yes
        ...                                 ;Otherwise try other services
        LDMFD   SP!, {PC}           ;Return

startFiler
        LDR     R14, taskHandle;Get task handle from workspace
        TEQ     R14, #0             ;Am I already active?
        MOVEQ   R14, #-1            ;No, so init handle to -1
        STREQ   R14, taskHandle         ;R12 relative
        ADREQ   R0, myCommand       ;Point R0 at command to start task
        MOVEQ   R1, #0                      ;(see earlier) and claim the service
        LDMFD   SP!, {PC}           ;Return
```

Note that the `taskHandle` word of the module's workspace must be zero before the task has been started. This word should therefore be cleared in the module's initialisation code. If the task is not already running, the StartFiler code should set the handle to –1, load the address of a command that can be used to start the module, and claim the call. Otherwise (if `taskHandle` is non-zero) it should ignore the call.

The automatic start-up process is made slightly more complex by the necessity to deal elegantly with errors that occur while a module is trying to start up. If the appropriate code is not executed, the Desktop can get into an infinite loop of trying to initialise unsuccessful modules.

This is avoided by the task setting its handle to –1 when it claims the StartFiler service. If the task fails to start, this will still be –1 the next time the Filer issues a Service_StartFiler, and so it will not claim the service.

Note that the Filer passes its own `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 task modules to set `taskHandle` variable to zero

On entry

R1 = Service_StartedFiler or Serice_Reset

On exit

Module's `taskHandle` variable set to zero

Use

A task which failed to initialise would have its taskHandle variable stuck at the value –1, which would prevent it from ever starting again (as Service_StartFiler would never be claimed). In order to avoid this, the two service calls should be recognised by the filing system task modules. On either of them, the task handle should be set to zero:

```
serviceCode
...
        TEQ     R1, #Service_StartedFiler;Service &4C?
        BNE     tryServiceReset  ;No
        LDR     R14, taskHandle  ;taskHandle = -1?
        CMN     R14, #1
        MOVEQ   R14, #0                  ;Yes, so zero it
        STREQ   R14, taskHandle
        LDMFD SP!, {PC}          ;Return

tryServiceReset
        TEQ     R1, #Service_Reset;Reset reason code?
        MOVEQ   R14, #0                  ;Yes, so zero handle
        STREQ   R14, taskHandle
        LDMFD   SP!, {PC}                ;Return
...
```

Service_StartedFiler is issued when the last of the resident filing system task modules has been started, and Service_Reset is issued whenever the computer is soft reset.

Service_FilerDying (Service Call &4F)

Notification that the Filer module is about to close down

On entry

R1 = Service_FilerDying

On exit

Module's `taskHandle` variable set to zero

Use

If the Filer module task is closed down (eg if the module is *RMKilled, or the Filer task is quitted from the TaskManager window) the Filer module tries to ensure that all the other filing system tasks are also closed down, by issuing this service call.

On receipt of this service call, a filing system task should check to see if it is active and if it is, it should close itself down by calling Wimp_CloseDown as follows:

```
serviceCode
...
        TEQ     R1, #Service_FilerDying
        BNE     try next
        STMFD SP!, {R0-R1, R14}
        LDR     R0, taskHandle          ; in workspace
        CMP     R0, #0
        MOVNE   R14, #0
        STRNE   R14, taskHandle
        LDRGT R1, taskid
        SWIGT XWimp_CloseDown
        LDMFD SP!, {R0-R1, PC}^         ; can't return errors from service calls

trynext
...
taskid DCB "TASK"                       ; word-aligned
```

SWI Calls

In the following section, we list all of the SWI calls provided by the Window Manager module. It is possible to make some generalisations about the routines, though there are inevitably exceptions:

- R0 is often used to hold or return a handle, be it task, window or icon.
- All Wimp calls do not preserve R0.
- Other registers are preserved unless used to return results.
- Flags are preserved unless overflow is set on exit.
- R1 is used as a pointer to information blocks, eg window definitions, icon definitions, Wimp_Poll blocks.
- The contents of a Wimp_Poll block are usually correctly set up for the most obvious routine to call for the returned reason code. For example, for an Open_Window_Request, the block will contain the information that Wimp_OpenWindow requires.
- All Wimp routines should not be executed with IRQs enabled due to the re-entrancy problems which may occur.
- Wimp routines may be called in User or SVC mode, except for Wimp_Poll, Wimp_PollIdle and Wimp_StartTask. These may only be called in User mode, as they rely on call-backs for their operation.
- As the Wimp uses the CallBack handler to do task swaps, it is not possible for a task to change the CallBack handler under interrupts. However language libraries can use the CallBack handler by setting it up when they start and using OS_SetCallBack (SWI &1B)

The following SWIs can only operate on windows owned by the task that is active when the call is made, and will report the error `Access to window denied` if an attempt is made to access another task's window:

Wimp_CreateIcon	except in the icon bar
Wimp_DeleteWindow	
Wimp_DeleteIcon	except in the icon bar
Wimp_OpenWindow	send Open_Window_Request instead
Wimp_CloseWindow	send Close_Window_Request instead
Wimp_RedrawWindow	
Wimp_SetIconState	
Wimp_UpdateWindow	

Wimp_GetRectangle
Wimp_SetExtent
Wimp_BlockCopy

This also means that a task cannot access its own windows unless it is a 'foreground' process, ie it has not gained control by means of an interrupt routine, or is inside its module Terminate entry.

Wimp_Initialise (SWI &400C0)

On entry

R0 = last Wimp version number known to task * 100 (ie at least 200)
R1 = "TASK" (low byte = "T", high byte = "K", ie &4B534154)
R2 = pointer to short description of task, for use in Task Manager display

On exit

R0 = current Wimp version number*100
R1 = task handle

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call registers a task with the Wimp, and must be called once only when the task starts up. The following is done when the first task starts up and when a 'grubby' task exits (ie a task that starts from and returns to the Dektop but does not use it) and there are more tasks running.

- redefines soft characters &80 to &85 and &88 to &8B for the window system
- programs function, cursor, Tab and Escape key statuses, remembering their previous settings
- issues *Pointer to initialise the mouse and pointer system
- uses Wimp_SetMode to set the mode to the configured WimpMode, or to the last mode the Wimp used if this is different
- sets up the palette.

Related SWIs

None

Related vectors

None

Wimp_CreateWindow (SWI &400C1)

On entry

R1 = pointer to window block

On exit

R0 = window handle

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call tells the Wimp what the characteristics of a window are. You should subsequently call Wimp_OpenWindow (SWI &400C5) to add it to the list of active windows (ones that are to be displayed). The format of a window block is as follows:

R1+0	visible area minimum x coordinate (inclusive)
R1+4	visible area minimum y coordinate (inclusive)
R1+8	visible area maximum x coordinate (exclusive)
R1+12	visible area maximum y coordinate (exclusive)
R1+16	scroll x offset relative to work area origin
R1+20	scroll y offset relative to work area origin
R1+24	handle to open window behind (–1 means top, –2 means bottom)
R1+28	window flags – see below
R1+32	title foreground and window frame colour – &FF means that the window has no control area or frame
R1+33	title background colour
R1+34	work area foreground colour
R1+35	work area background colour – &FF means 'transparent', so the Wimp won't clear the rectangles during a redraw operation

R1+36	scroll bar outer colour
R1+37	scroll bar inner (Slider) colour
R1+38	title background colour when highlighted for input focus
R1+39	reserved – must be 0
R1+40	work area minimum x coordinate
R1+44	work area minimum y coordinate
R1+48	work area maximum x coordinate
R1+52	work area maximum y coordinate
R1+56	Title Bar icon flags – see below
R1+60	work area flags giving button type – see below
R1+64	sprite area control block pointer (+1 for Wimp sprite area)
R1+68	minimum width of window; NB two-byte quantities
R1+70	minimum height of window 0,0 means use title width instead
R1+72	title data – see below
R1+84	number of icons in initial definition (can be 0)
R1+88	icon blocks, 32 bytes each – see Wimp_CreateIcon (SWI &400C2)

Note that the entries from R1+0 to R1+24 are not used unless Wimp_GetWindowState is called.

Fields requiring further explanation are:

Window flags

Window flags and status information are held in the word at offsets +28 to +31.

Bit	Meaning when set
0	*window has a Title Bar
1	window is moveable, ie it can be dragged by the user
2	*window has a vertical scroll bar
3	*window has a horizontal scroll bar
4	window can be redrawn entirely by the Wimp, ie there are no user graphics in the work area. Redraw window requests won't be generated if this bit is set
5	window is a pane, ie it is on top of a tool window
6	window can be opened (or dragged) outside the screen area
7	*window has no Back icons or Close icons
8	a Scroll_Request event is returned when a mouse button is clicked on one of the arrow icons (with auto-repeat) or in the outer scroll bar region (no auto-repeat)

9 as above but no auto-repeat on the arrow icons
10 treat the window colours given as GCOL numbers instead of standard Wimp colours. This allows access to colours 0 - 254 in 256-colour modes (255 always has a special meaning)
11 don't allow any other windows to be opened below this one (used by the icon bar, and the backdrop for pre-RISC OS style applications)
12 generate events for 'hot keys' passed back through Wimp_ProcessKey if the window is open
13 - 15 reserved; must be 0

Flags marked * are old-style control icon flags. You should use bits 24 to 31 in preference.

The five bits bits below are set by the Wimp and may be read using Wimp_GetWindowState (SWI &400CB).

16 window is open
17 window is fully visible, ie not covered at all
18 window has been toggled to full size
19 the current Open_Window_Request was caused by a click on the Toggle Size icon
20 window has the input focus

21 - 23 reserved; must be 0

The eight bits below provide an alternative way of determining which control icons a window has when it is created. If bit 31 is set, bits 24 to 30 determine the presence of one system icon, otherwise the 'old style' control icon flags noted above are used.

24 window has a Back icon
25 window has a Close icon
26 window has a Title Bar
27 window has a Toggle Size icon
28 window has a vertical scroll bar
29 window has a Adjust Size icon
30 window has a horizontal scroll bar
31 use bits 24 - 30 to determine the control icons, otherwise use bits 0, 2, 3 and 7

A window may only have a quit and/or Back icon if it has a Title Bar, and a if it has one or two scroll bars. A Toggle Size icon needs a vertical scroll bar or a Title Bar. We recommend that new applications use the bit 31 set method of determining the control icons.

Bits 24 to 30 are also returned by Wimp_GetWindowState, updated to reflect what actually happened, so you can use this to ensure that the control icons used by the Wimp are as specified when the window was created, ie it was a valid specification.

Title bar flags

Title bar flags are held in the four bytes +56 to +59 of a window block. They correspond to the icon flags used in an icon block, described under Wimp_CreateIcon below. They determine how the contents of the Title Bar are derived and displayed. Note the following differences from proper icon flags though:

- the Title Bar always has a border, ie bit 2 is ignored
- the title background is filled, ie bit 5 is ignored
- the Wimp redraws the title, ie bit 7 is ignored
- any flags to do with button types, ESGs and selections are ignored. Dragging on the Title Bar always drags the window.
- if an anti-aliased font, or sprite, is used, you should bear in mind that the height of the Title Bar is fixed at 44 OS units, or 36 if you subtract the top and bottom frame lines. Thus only font sizes of about 10 to 12 points can be accommodated, and fairly small sprites. Also remember that lines will vary in width according to the screen mode used
- bits 24 - 31 (when used as text colours) are ignored; the Title Bar colours are given in other window definition bytes

So, the title may be text or a sprite, may be indirected (but not writeable), use normal or anti-aliased text, and may be positioned within the Title Bar as required.

Title data

Title data is held in the twelve bytes at +72 to +83 of a window block. It has the same interpretation as the icon data bytes described under Wimp_CreateIcon. In summary:

- if text, then up to 12 bytes of text including a terminating control code

- if a sprite, then the name of the sprite (12 bytes)
- if the Title Bar is indirected, then the following three words: a pointer to a buffer containing the text, a pointer to a validation string (-1 if none), and the length of the buffer.

See the section on icon data under Wimp_CreateIcon (SWI &400C2) for more details.

Window button types

The word at offset +26 in a window block is used to determine the 'button type' of the work area. Only bits 12 to 15 of this word are used. The 16 possible button types are much as described in the section on icon creation below. Note though that there is no concept of a window's work area being 'selected' by the Wimp; the user is simply informed of button clicks through the Mouse_Click event.

Note that as stated previously, the button type only determines how Select and Adjust are handled; Menu is always reported. The interpretations of the button types for windows then are:

Bits 12 - 15	Meaning
0	ignore all clicks
1	notify task continually while pointer is over the work area
2	click notifies task (auto-repeat)
3	click notifies task (once only)
4	release over the work area notifies task
5	double click notifies task
6	as 3, but can also drag (returns button state * 16)
7	as 4, but can also drag (returns button state * 16)
8	as 5, but can also drag (returns button state * 16)
9	as 3
10	click returns button state*256 drag returns button state*16 double click returns button state*1
11	click returns button state drag returns button state*16
12 - 14	reserved
15	mouse clicks cause the window to gain the input focus.

Icons

The handles of any icons defined in this call are numbered from zero upwards, in the same order that they appear in the block. For details of the 32-byte definitions, see the next section.

Note: the Wimp_CreateWindow call may produce a `Bad work area extent` error if the visible area and scroll offsets combine to give a visible work area that does not lie totally within the work area extent.

Related SWIs

None

Related vectors

None

Wimp_CreateIcon (SWI &400C2)

On entry

R1 = pointer to block

On exit

R0 = icon handle (unique within that window)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The block contains the following:

R1+0	window handle (or –2 left of icon bar, –1 for right)
R1+4	icon block

where an icon block is defined as:

+0	minimum x coordinate of icon bounding box
+4	minimum y coordinate of icon bounding box
+8	maximum x coordinate of icon bounding box
+12	maximum y coordinate of icon bounding box
+16	icon flags
+20	12 bytes of icon data

Icon blocks are also used in the Wimp_CreateWindow block and returned by Wimp_GetWindowInfo (SWI &400CC).

This call tells the Wimp what the characteristics of an icon are. Once you have defined the icon, you can only make these changes to it:

- you can change its flags using the call Wimp_SetIconState (SWI &400CD).

- you can change indirected text. The icon must then be redrawn using the call Wimp_SetIconState, leaving the flags unchanged if necessary.
- you can change its text if its button type is 15 (writeable). The Wimp does this for you automatically, handling the caret positioning and text updating. For further details, see the sections on Wimp_SetCaretPosition (SWI &400D2), Wimp_GetCaretPosition (SWI &400D3), and the Wimp_Poll Key_Pressed event.

The window handle at R1+0 may be an application window, or –2 for the left half of the icon bar (devices), or –1 for the right half of the icon bar (applications). Note that creating an icon on the icon bar may cause other icons to 'shuffle', changing their x coordinates.

The bounding box coordinates are given relative to the window's work area origin, except that the horizontal offset may be applied to an icon created on the icon bar. Note that if an icon is writeable, the icon bounding box determines how much of the string is displayed at once. Typing into the icon or moving the caret left or right can cause the string to scroll within this box. The buffer length entry in the icon data determines the maximum number of characters that can be entered into a writeable icon. One character is used for the terminator.

Note that icon strings can be terminated by any character from 0 to 31, and are preserved during editing operations by the Wimp. However, in template files, the terminator must be 13 (Return).

Icon flags

As noted earlier, subsets of these flags are used in Wimp_CreateWindow blocks to control how the contents of a window's Title Bar is defined, and the button type bits are used to determine how clicks within a window's work area are processed.

The full list of flags for a proper icon is:.

Bit	Meaning when set
0	icon contains text
1	icon is a sprite
2	icon has a border
3	contents centred horizontally within the box
4	contents centred vertically within the box
5	icon has a filled background

6	text is an anti-aliased font (affects meaning of bits 24 - 31)
7	icon requires task's help to be redrawn
8	icon data is indirected
9	text is right-justified within the box
10	if selected with Adjust don't cancel others in the same ESG
11	display the sprite (if any) at half size
12 - 15	icon button type
16 - 20	exclusive selection group (ESG, 0 - 31)
21	icon is selected by the user and is inverted
22	icon cannot be selected by the mouse pointer; it is shaded
23	icon has been deleted
24 - 27	foreground colour of icon (if bit 6 is cleared)
28 - 31	background colour of icon (if bit 6 is cleared)
or	
24 - 31	font handle (if bit 6 is set). Font colours may be passed in an indirected icon's validation string.

Icon button types

These are much the same as window button types. However, icons can be 'selected' (inverted) by the Wimp automatically, so there are some additional effects to those already described for windows:

0	ignore mouse clicks or movements over the icon (except Menu)
1	notify task continuously while pointer is over this icon
2	click notifies task (auto-repeat)
3	click notifies task (once only)
4	click selects the icon; release over the icon notifies task; moving the pointer away deselects the icon
5	click selects; double click notifies task
6	as 3, but can also drag (returns button state * 16)
7	as 4, but can also drag (returns button state * 16) and moving away from the icon doesn't deselect it
8	as 5, but can also drag (returns button state * 16)
9	pointer over icon selects; moving away from icon deselects; click over icon notifies task ('menu' icon)
10	click returns button state*256 , drag returns button state*16 double click returns button state*1
11	click selects icon and returns button state drag returns button state*16
12 - 13	reserved
14	clicks cause the icon to gain the caret and its parent window to become the input focus and can also drag (writeable icon). For

example, this is used by the FormEd application

15 clicks cause the icon to gain the caret and its parent window to become the input focus (writeable icon)

All the above return Mouse_Click events (6), where the button state is:

Bit	Meaning when set
0	Adjust pressed
1	Menu pressed
2	Select pressed, or combination of above

A drag is initiated by the button being held down for more than about a fifth of a second. A double click is reported if the button is clicked twice in one second and the second click is within 16 OS units of the first. Note that button types which report double clicks will also report the initial click first.

Icon data

The icon data at +20 to +31 is interpreted according to the settings of three of the icon flags. The three bits are Indirected (bit 8), Sprite (bit 1) and Text (bit 0). The eight possible combinations and the eight interpretations of the icon data are:

IST	Meaning of 12 bytes/3 words
000	non-indirected, non-sprite, non-text icon
	+20 icon data not used in this case
001	non-indirected, text-only icon
	+20 the text string to be used for the icon, control-terminated
010	non-indirected, sprite-only icon
	+20 the sprite name to be for the icon, control-terminated
011	non-indirected, text plus sprite icon
	+20 the text and sprite name to be used – not especially useful
100	indirected, non-sprite, non-text icon
	+20 icon data not used in this case
101	indirected, text-only icon
	+20 pointer to text buffer
	+24 pointer to validation string – see below
	+28 buffer length
110	indirected, sprite-only icon
	+20 pointer to sprite or to sprite name; see +28
	+24 pointer to sprite control block, +1 for Wimp sprite area

+28 0 if [+20] is a sprite pointer, length if it's a sprite name pointer

111 indirected, text plus sprite icon
- +20 pointer to text buffer
- +24 pointer to validation string, which can contain sprite name
- +28 buffer length

Note that the icon bar's sprite area pointer is set to +1, so icons there use Wimp sprites. If you want to put an icon on the icon bar that isn't from the Wimp area, you must use an indirected sprite-only icon, type 110 above.

In Wimp 2.00 it is not possible to set the caret in the icon bar, so writeable icons should not be used.

Validation Strings

An indirected text icon can have a validation string which is used to pass further information to the Wimp, such as what characters can be inserted directly into the string and which should be passed to the user via the Key_Pressed event for processing by the application. The syntax of a validation string is:

- validation-string ::= command { ; command }*
- command ::= a allow-spec | d char | f hex-digit hex-digit | | {decimal-number} | s text-string {,text-string}
- allow-spec ::= { char-spec }* { ~ { char-spec }* }*
- char-spec ::= char | char-char
- char ::= \~ | \; | \\ | \~ | any character other than – ;

The spaces in the above definition are for clarity only, and a validation string will normally have no spaces in it.

In simple terms, a validation string consists of a series of 'commands', each starting with a single letter and separated from the following command by a semicolon. { }* means zero or more of the thing inside the { }. The following commands are available:

A command

The (A)llow command tells the Wimp which characters are to be allowed in the icon. Characters are inserted into the string if:

- a key is typed by the user

- the key returns a character code in the range 32 - 255
- the input focus is inside the icon
- the validation string allows the character within the string.

Otherwise:

- control keys such as the arrow keys and Delete are automatically dealt with by the Wimp
- other keys are returned to the task via the Key_Pressed event.

Each char-spec in the 'allow' string specifies a character or range of characters; the ~ character toggles whether they are included or excluded from the icon text string:

`A0-9a-z~dpu`	allows the digits 0 - 9 and the lower-case letters a - z, except for 'd', 'p' and 'u'

If the first character following the A command is a ~ all normal characters are initially included:

`A~0-9`	allows all characters except for the digits 0 - 9

If you use any of the four special characters - ; ~\ in a char-spec you must precede them with a backslash \:

`A~\-\;\~\\`	allows all characters except the four special ones - ; ~ \

D command

The (D)isplay command is used for password icons to avoid onlookers seeing what is typed. It is followed by a character that is used to echo all allowed characters:

`D*`	displays the password as a row of asterisks

Note that if the character is any of the four 'special' characters above, you must precede it by a \:

`D\-`	displays the password as a row of dashes

F command

The (F)ont colours command is used to specify the foreground and background colours used in text icons with an anti-aliased font. The F is followed by two hexadecimal digits, which specify the background and foreground Wimp colours respectively:

```
Fa3          sets background to 10 (&a hex), and foreground to 3.
```

This command uses the call Wimp_SetFontColours (SWI &400F3). If you do not use this command, the colours 0 and 7 (black on white) are used by default.

L command

The (L)ine spacing command is used to tell the Wimp that a text icon may be formatted. If the text is too wide for the icon it is split over several lines. You should follow the L with a decimal number giving the vertical spacing between lines of text in OS units – if omitted, the default used is 40 units. (A system font character is 32 OS units high.)

The current version of RISC OS ignores the number following the L, so no number can be specified. However, this option may be implemented in future versions of RISC OS.

This option can only be used with icons which are horizontally and vertically centred, and do not contain an anti-aliased font. The icon must not be writeable, since the caret would not be positioned correctly inside it.

S command

The (S)prite name command is used to give a text and sprite icon a different sprite name from the text it contains, for example `Sfile_abc`. No space should follow the S, and the sprite name should be no more than 12 characters long.

If a second name is given, separated from the first by a comma, this is used when the icon is highlighted. If it is omitted, the sprite is highlighted by plotting it with its original colours exclusive-OR'ed with the icon foreground colour.

Text plus sprite icons

If an icon has both its text and sprite bits (0 and 1) set, then it will contain both objects. The text must be indirected, so that the validation string can be used to give the sprite name(s) to use (see the S command above).

Three flags in the icon flags are used to determine the relative positions of the text and sprite. These are the Horizontal, Vertical and Right justified bits (3, 4, and 9 respectively). The eight possible combinations of these bits, and how they position the sprite and text within the icon bounding box, are as follows:

HVR	Horizontal	Vertical
000	text and sprite left justified	text at bottom, sprite at top
001	text and sprite right justified	text at bottom, sprite at top
010	sprite at left, text +6 units right of it	text and sprite centred
011	text at left, sprite at right	text and sprite centred
100	text and sprite centred	text at bottom, sprite at top
101	text and sprite centred	text at top, sprite at bottom
110	text and sprite centred (text on top)	text and sprite centred
111	text at right, sprite at left	text and sprite centred

The following points should be noted about text plus sprite icons:

- the text part can be writeable, but every time a key is pressed the sprite will be redrawn and so can flicker
- the text part of the icon always has its background filled
- if the text uses an anti-aliased font, the icon should not have a filled background, as the drawing of the text's background will obscure the sprite
- as usual, the whole of the icon area is used to delimit mouse clicks or movements over the icon, so clicks cannot be associated separately with the text and sprite (so clicking over the sprite would still cause the text of a writeable icon to gain the caret)

An important use of this type of icon is displaying a text plus sprite pair in the icon bar.

Related SWIs

None

Related vectors

None

Wimp_DeleteWindow (SWI &400C3)

On entry

R1 = pointer to block

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The block contains the following:

R1+0 window handle

This call closes the specified window if it is still open, and then removes the definition of the window and of all the icons within it. The memory used is re-allocated, except for the indirected data, which is in the task's own workspace.

Related SWIs

None

Related vectors

None

Wimp_DeleteIcon (SWI &400C4)

On entry

R1 = pointer to block

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The block contains the following:

R1+0	window handle (–1 or –2 for icon bar)
R1+4	icon handle

This call removes the definition of the specified icon. If the icon is not the last one in its window's list it is marked as deleted, so that the handles of the other icons within the window are not altered. If the icon is the last one in the list, the memory is reallocated.

Note: this call does not affect the screen. You must make a call to Wimp_ForceRedraw (SWI &400D1) to remove the icon(s) deleted, passing a bounding box containing the icons.

Related SWIs

None

Related vectors

None

Wimp_OpenWindow (SWI &400C5)

On entry

R1 = pointer to block

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The block contains the following:

R1+0	window handle
R1+4	visible area minimum x coordinate
R1+8	visible area minimum y coordinate
R1+12	visible area maximum x coordinate
R1+16	visible area maximum y coordinate
R1+20	scroll x offset relative to work area origin
R1+24	scroll y offset relative to work area origin
R1+28	handle to open window behind (–1 means top of window stack, –2 means bottom)

This call updates the list of active windows (ones that are to be displayed). The window may either be a new one being displayed for the first time, or an already open one that has had its parameters altered.

Note that coordinates (x0,y0,x1,y1,scroll x,scroll y) are ALL rounded down to whole numbers of pixels. This also happens on a mode change automatically.

Related SWIs

None

Related vectors

None

Wimp_CloseWindow (SWI &400C6)

On entry

R1 = pointer to block

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The block contains the following:

R1+0 window handle

This call removes the specified window from the active list; it is no longer marked as one to be displayed. The Wimp will issue redraw requests to other windows that were previously obscured by the closed one.

Related SWIs

None

Related vectors

None

Wimp_Poll (SWI &400C7)

On entry

R0 = mask
R1 = pointer to 256 byte block (used for return data)

On exit

R0 = reason code
R1 = pointer to block (data depends on reason code returned)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call checks to see whether certain events have occurred, and oversees such things as screen updating, keyboard and mouse handling, and menu selections. You must call it in the main loop of any program you write to run under the Wimp, and provide handlers for each reason code it can return.

The following reason codes may be returned:

Code	Reason
0	Null_Reason_Code
1	Redraw_Window_Request
2	Open_Window_Request
3	Close_Window_Request
4	Pointer_Leaving_Window
5	Pointer_Entering_Window
6	Mouse_Click
7	User_Drag_Box
8	Key_Pressed
9	Menu_Selection
10	Scroll_Request

11	Lose_Caret
12	Gain_Caret
13 - 16	Reserved
17	User_Message
18	User_Message_Recorded
19	User_Message_Acknowledge

The highest priority are types 17 - 19, then 1 - 6,8,9. The remaining reason codes are next and the lowest priority type is 0.

You can disable some of the reason codes; they are neither checked for nor returned, and need not have handlers provided. You must do this for as many codes as possible, especially the Null_Reason_Code, if your task is to run efficiently under the Wimp. Some of the remaining reason codes can be temporarily queued to prevent their return at times when they would otherwise interfere with the task running. Both the above are done by setting bits in the mask passed in R0:

Bit	Meaning when set
0	do not return Null_Reason_Code
1	do not return Redraw_Window_Request; queue for later handling
2 - 3	must be 0
4	do not return Pointer_Leaving_Window
5	do not return Pointer_Entering_Window
6	do not return Mouse_Click; queue for later handling
7	must be 0
8	do not return Key_Pressed; queue for later handling
9 - 10	must be 0
11	do not return Lose_Caret
12	do not return Gain_Caret
13 - 16	must be 0
17	do not return User_Message
18	do not return User_Message_Recorded
19	do not return User_Message_Acknowledge
20 - 31	must be 0

Note that the bits which are marked 'queue for later handling' above stop the Wimp from proceeding ie. it stops all other tasks too.

As you can see, certain events cannot be masked out and the task must always be prepared to handle them. Each reason code has one Wimp SWI that is most likely to be called in response. The block returned by Wimp_Poll is formatted ready to be passed directly to this call.

The reason codes are as follows:

Null_Reason_Code 0

This reason code is returned when none of the others are applicable. It should be masked out whenever possible to minimise the overheads incurred by the Wimp, so it doesn't have to set-up the task's memory and return control to it, only to find the task isn't interested anyway.

Redraw_Window_Request 1

The returned block contains:

R1+0	window handle

This reason code indicates that some of the window is out of date and needs redrawing. You should call Wimp_RedrawWindow (SWI &400C8) using the returned block, and then call Wimp_GetRectangle (SWI &400CA) as necessary. See their entries for further details and a scheme of the code required.

Open_Window_Request 2

The returned block contains:

R1+0	window handle
R1+4	visible area minimum x coordinate
R1+8	visible area minimum y coordinate
R1+12	visible area maximum x coordinate
R1+16	visible area maximum y coordinate
R1+20	scroll x offset relative to work area origin
R1+24	scroll y offset relative to work area origin
R1+28	handle to open window behind (–1 means top of window stack, –2 means bottom)

This reason code is returned as a result of the Adjust Size icon or the Title Bar of a window being selected, or as a result of the scroll bars being dragged to a new position. The dragging process is performed by the Wimp itself before it returns this reason code to the task.

Following detection, the Wimp sets five bits that determine the action on the window. These bits can be read using Wimp_GetWindowState (SWI &400CB) – refer to Wimp_CreateWindow (SWI &400C1) for more information.

You should call Wimp_OpenWindow (SWI &400C5) using the returned block and also call it for any pane windows that are attached to this one, using the coordinates in the block to determine the pane's position.

Close_Window_ Request 3

The returned block contains:

R1+0 window handle

This reason code is returned when you click with the mouse on the Close icon of a window.

You should normally call Wimp_CloseWindow (SWI &400C6) using the returned block. You may also need to issue further calls of Wimp_CloseWindow to close any dependent windows, eg panes. However, if you do not want to close the window immediately, you could open an error box, or ask the user for confirmation.

Programs such as Edit conventionally open the directory which holds the edited file if its window is closed using the Adjust button. This is done by calling Wimp_GetPointerInfo when the Close_Window_Request is received, and performing the appropriate action.

Pointer_Leaving_ Window 4

The returned block contains:

R1+0 window handle

This reason code is returned when the pointer has left a window's visible work area. You might use it to make the pointer revert to its default shape when it is no longer over your window's work area. However, it is not recommended that you use it to make dialogue boxes disappear as soon as the mouse pointer leaves them.

Note that this event doesn't only occur when the pointer leaves the window's visible work area, but whenever the window stops being the most visible thing under the pointer. So, for example, popping up a menu at the pointer position would cause this event.

Pointer_Entering_Window 5

The returned block contains:

R1+0	window handle

This reason code is returned when the pointer has moved onto a window. You might use it to bring a window to the top as soon as the pointer enters its work area, or to change the pointer shape when it over the visible work area.

As with the previous event type, Pointer_Entering_Window doesn't just happen when the pointer is physically moved into a window's visible work area. It could occur because a menu is removed or a window is closed, revealing a new uppermost window.

Mouse_Click 6

The returned block contains:

R1+0	mouse x (screen coordinates – not window relative)
R1+4	mouse y
R1+8	buttons (depending on window/icon button type)
R1+12	window handle (–1 for background, –2 for icon bar)
R1+16	icon handle (–1 for work area background)

This reason code is returned when:

- the state of the mouse buttons has changed, and
- the conditions of the button type have been met, and
- the Wimp does not automatically deal with the change in some other way.

For example:

- if an icon has button type 6, a click with Select will generate this event with buttons=4, whereas a drag with Adjust will give buttons=1 followed by another event with buttons=16
- if the change took place over a window's Close icon, this reason code will not be returned as Close_Window_Request is used instead
- a click on the Menu button is always reported with buttons=2

The window and icon handles indicate which window and icon the mouse pointer was over when the button change took place. Operations such as highlighting an icon when it is selected and the cancellation of the other

selections in the same ESG are all done automatically by the Wimp. See the section on button types in Wimp_CreateIcon (SWI &400C2) for details of the various icon button modes and mouse return codes.

User_Drag_Box 7

The returned block contains:

R1+0	drag box minimum x coordinate (inclusive)
R1+4	drag box minimum y coordinate (inclusive)
R1+8	drag box maximum x coordinate (exclusive)
R1+12	drag box maximum y coordinate (exclusive)

This reason code is returned when you release all the mouse buttons to finish a User_Drag operation. The block contains the final position of the drag box.

A user drag operation starts when the task calls Wimp_DragBox with a drag type of 5 to 11, usually in response to a drag code returned in a Mouse_Click event.

During the user drag operation (particularly with drag type 7), you may wish to keep track of the pointer position. To do this, call Wimp_GetPointerInfo (SWI &400CF) each time you receive a null event from Wimp_Poll. You can use the coordinates returned to redraw the dragged object (using Wimp_UpdateWindow (SWI &400C9) of course).

When this reason code is returned the drag is over; you should then stop reading the pointer information and, if appropriate, redraw the dragged object in its final position.

Key_Pressed 8

The returned block contains:

R1+0	window handle with input focus
R1+4	icon handle (–1 if none)
R1+8	x-offset of caret (relative to window origin)
R1+12	y-offset of caret (relative to window origin)
R1+16	caret height and flags (see Wimp_SetCaretPosition)
R1+20	index of caret into string (undefined if not in an icon)
R1+24	character code of key pressed (NB this is a word, not a byte)

This reason code is returned to tell a task that a key has been pressed while the input focus belonged to one of its windows. The task should process the key if possible. Otherwise the task should pass it to Wimp_ProcessKey (SWI &400DC) so that other tasks can then intercept 'hot key' codes.

If the caret is inside a writeable icon, the Wimp automatically processes the keys listed below, and does not generate an event:

Printable characters	are inserted into the text, if there is room, and the icon is redrawn
Delete, <-|	delete character to left of caret
Copy	delete character to right of caret
<-	move left one character
->	move right one character
Shift Copy	delete word (forwards)
Shift <-	move left one word (returns &19C if at left of line)
Shift ->	move right one word (returns &19D if at right of line)
Ctrl Copy	delete forwards to end of line
Ctrl <-	move to left end of line
Ctrl ->	move to right end of line

'Printed characters' are those printable ones whose codes are in the ranges &20 - &7E and &80 - &FF.

Clashes could occur between top-bit-set characters (obtained by pressing Alt plus ASCII code on the keypad) and special key codes. The Wimp avoids any such ambiguities by mapping the special keys to these values:

Key	Alone	+Shift	+Ctrl	+Ctrl Shift
Escape	&1B	&1B	&1B	&1B
Print (F0)	&180	&190	&1A0	&1B0
F1 - F9	&181 - 189	&191 - 199	&1A1 - 1A9	&1B1 - 1B9
Tab	&18A	&19A	&1AA	&1BA
Copy	&18B	&19B	&1AB	&1BB
left arrow	&18C	&19C	&1AC	&1BC
right arrow	&18D	&19D	&1AD	&1BD
down arrow	&18E	&19E	&1AE	&1BE
up arrow	&18F	&19F	&1AF	&1BF
Page down	&19E	&18E	&1BE	&1AE
Page up	&19F	&18F	&1BF	&1AF
F10 - F12	&1CA - 1CC	&1DA - 1DC	&1EA - 1EC	&1FA - &1FC
Insert	&1CD	&1DD	&1ED	&1FD

These are set up by Wimp_Initialise. Tasks running under the Wimp are not allowed to change any of these settings. Soft key expansions (outside of writeable icons) must be performed by the task accessing the key's expansion string using the `key$n` variables.

Menu_Selection 9

The returned block contains:

R1+0	item in main menu which was selected (starting from 0)
R1+4	item in first submenu which was selected
R1+8	item in second submenu which was selected
	.....
	terminated by –1

This reason code is returned when the user selects an item from a menu. Selections can be made by the user clicking on an item with any of the mouse buttons. Select and Menu are synonymous; Adjust has a slightly different effect, as discussed below. A press of Return inside a writeable menu item also generates this event (though not if it is pressed inside a writeable icon inside a menu dialogue box).

The values in the block indicate which item at each menu level was chosen, the first item in each menu being numbered 0. An entry of –1 terminates the list. No handle is used for menus, so the task must remember which menu it last used Wimp_CreateMenu (SWI &400D4) to open.

If the last item specified has submenus (ie was not a 'leaf' of the menu tree) then the command may be ambiguous, in which case the task should ignore it. If the command is clear, but not its parameters, then the task may ignore the command, use default parameters, or use the last parameters set, as is most appropriate.

There is a difference, from the user's point of view, between choosing an item with Select and Adjust. In the former case, the selection will also cancel the menu, causing it to be removed from the screen. In the latter case, the menu should stay on the screen (a persistent menu). The application achieves this as follows. Call Wimp_GetPointerInfo (SWI &400CF) to read the mouse button state, and save it. After decoding the menu selection and taking the appropriate action, examine the stored button state. If Select was pressed, just return to the polling loop.

If Adjust was down, however, re-encode the menu tree (reflecting any changes that the previous menu selection effected) and call Wimp_CreateMenu with the same menu tree pointer that was used to create the menu in the first place. The next time you call Wimp_Poll, the Wimp will spot the re-opened menu, and recreate it on the screen. It goes down the tree until the end of the tree is reached, or the tree fails to correspond to the previous one, or until a shaded item is reached.

Scroll_Request 10

The returned block contains:

R1+0	window handle
R1+4	visible area minimum x coordinate
R1+8	visible area minimum y coordinate
R1+12	visible area maximum x coordinate
R1+16	visible area maximum y coordinate
R1+20	scroll x offset relative to work area origin
R1+24	scroll y offset relative to work area origin
R1+28	handle to open window behind (–1 means top of the window stack, –2 means bottom)
R1+32	scroll x direction
R1+36	scroll y direction

The scroll directions have the following meanings:

Value	Meaning
–2	Page left/down (click in scroll bar outer area)
–1	Left/down (click on scroll arrow)
0	No change
+1	Right/up (click on scroll arrow)
+2	Page right/up (click in scroll bar outer area)

This reason code is returned if the user clicks in a scroll area of a window which has one of the 'Scroll_Request returned' bits set in its window flags. It returns the old scroll bar offsets and the direction of scrolling requested. The task should work out the new scroll offsets, store them in the scroll offsets (R1+20 and R1+24) of the returned block, and then call Wimp_OpenWindow (SWI &400C5).

Remember that the coordinates used for scroll offsets are in OS units. Therefore, if you want to make a click on one of the arrows scroll by, say, one pixel, you must scale the –1 or 1 returned in the event block by the appropriate factor for the current mode. For example, in !Edit the text is aligned with the bottom of the window when scrolling down, and subsequently moves down by one text line exactly. When scrolling up, the text is aligned with the top of the window.

Lose_Caret 11

This is returned when the window which owns the input focus has changed. That happens when Wimp_SetCaretPosition (SWI &400D2) is called, either explicitly, or implicitly by the user clicking on a button type 15 object. The event isn't generated if the input focus only changes position within the same window.

The event warns the task which had the caret (and which may well be retaining it) that something has changed. It can be used to remove a specialised text-position indicator which does not use the Wimp's caret, or its appearance could be altered to show this is where the caret would be if the window still had the input focus.

R1 points to a standard caret block:

R1+0	window handle that had the input focus (–1 if none)
R1+4	icon handle (–1 if none)
R1+8	x-offset of caret (relative to window origin)
R1+12	y-offset of caret (relative to window origin)
R1+16	caret height and flags (see Wimp_SetCaretPosition)
R1+20	index of caret into string (or –1 if not in a writeable icon)

Gain_Caret 12

This event is returned to the task which now has the caret, subsequent to a Wimp_SetCaretPosition. The block pointed to by R1 is the same as above, except that the window/icon handle is the caret's new owner.

Events 13 - 16: not used

The next three reason codes (17 - 19) are concerned with the receipt of user messages. Events of type 0 to 12 are normally sent directly from the Wimp to a task in response to some user action. The User_Message reason codes are more general purpose, and are sent from Wimp to task, or from task to task. See the description of Wimp_SendMessage (SWI &400E7) for more details about the sending of messages and of the various types of User_Message actions which are defined.

One message action that all tasks should act on is Message_Quit, which is broadcast by the Desktop when the user selects the Exit item from its menu.

User_Message 17

The returned block contains:

R1+0	size of block in bytes (20 - 256 in a multiple of four (ie. words))
R1+4	task handle of message sender
R1+8	my_ref – the sender's reference for this message
R1+12	your_ref – a previous message's my_ref, or 0 if this isn't a reply
R1+16	message action code
R1+20	message data (dependent on message action)
	...

This event is returned when another task has sent a message to the current task, to one of its windows, or to all tasks using a broadcast message. The action code field defines the meaning of the message, ie how the message data should be processed by the receiver.

If the message is not acknowledged (because the receiving task is no longer active, or just ignores it) then no further action is taken by the Wimp.

User_Message_ Recorded 18

The block has the same format as that described above under User_Message. The interpretation of the message action is the same, so the way in which the receiving task handles these two types should be identical. However, the way the Wimp responds differs if the message is not acknowledged.

The receiving task can acknowledge the message by calling Wimp_SendMessage with the reason code User_Message_Acknowledge (19) and the your_ref field set to the my_ref of the original. This will prevent the sender from receiving its original message back from the Wimp with the event type 19.

Another way to acknowledge a message (and prevent the Wimp returning it to the sender) is to send a reply message using reason code User_Message or User_Message_Acknowledge, again with the your_ref field set to the original message's my_ref.

Both types of acknowledgement must take place before the next call to Wimp_Poll.

User_Message_Acknowledge 19

The format of the block is as above. This event type is generated by the Wimp when a message sent with reason code User_Message_Recorded was not acknowledged or replied to by the receiver. The message in the block is identical to the one sent by the task in the first place.

Note that a task should ignore any messages it does not understand: it must not acknowledge messages as a matter of course. See Wimp_SendMessage (SWI &400E7) for details.

Related SWIs

None

Related vectors

None

Wimp_RedrawWindow (SWI &400C8)

On entry

R1 = pointer to block

On exit

R0 = 0 for no more to do, non-zero for update according to returned block

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The block contains the following:

R1+0	window handle
R1+4	visible area minimum x coordinate
R1+8	visible area minimum y coordinate
R1+12	visible area maximum x coordinate
R1+16	visible area maximum y coordinate
R1+20	scroll x offset relative to work area origin
R1+24	scroll y offset relative to work area origin
R1+28	current graphics window minimum x coordinate
R1+32	current graphics window minimum y coordinate
R1+36	current graphics window maximum x coordinate
R1+40	current graphics window maximum y coordinate

The window handle at +0 is set on entry, usually from the last call to Wimp_Poll; the rest of the block is filled in by Wimp_RedrawWindow.

Note that this SWI must be called as the first Wimp operation after the Wimp_Poll which returned a Redraw_Window_Request. This means that you cannot, for example, delete or create any other windows between the Wimp_Poll and the Wimp_RedrawWindow. If you need to do any special extra operations in your Wimp_Poll loop, do them just before calling Wimp_Poll, not afterwards.

This call is used to start a redraw of the parts of a window that are not up to date. These consist of a series of non-overlapping rectangles. Wimp_RedrawWindow draws the window outline, issues VDU 5, and then exits via Wimp_GetRectangle, which returns the coordinates of the first invalid rectangle (if any) of the work area, and clears it to the window's background colour, unless it's transparent. It also returns a flag saying whether there is anything to redraw.

The first four words are the position of the window's work area on the screen, ie they have the same meaning as those words in the Wimp_CreateWindow (SWI &400C1) and Wimp_OpenWindow (SWI &400C5) blocks.

The last four words describe an area within the visible work area in screen coordinates, not work area relative, possibly the whole thing if the window is not covered. The graphics clip window is set to the returned rectangle. A task could just redraw its entire work area each time a rectangle is returned. However, it is much more efficient if the task takes note of the graphics clip window co_ordinates and works out what it needs to draw.

By using these two sets of coordinates in conjunction with the scroll offsets, you can find the work area coordinates to be updated:

```
work x = screen x – (screen x0–scroll x)
work y = screen y – (screen y1–scroll y)
```

where:

```
screen x0 =  [R1+4]
screen y1 =  [R1+16]
scroll x =   [R1+20]
scroll y =   [R1+24]
```

The code used to redraw the window was outlined in the section **Redrawing windows**. The expressions above in parenthesis are the screen coordinates of the work area origin.

Related SWIs

None

Related vectors

None

Wimp_UpdateWindow (SWI &400C9)

On entry

R1 = pointer to block – see below

On exit

R0 and block as for Wimp_RedrawWindow (SWI &400C8)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The block contains the following on entry:

R1+0	window handle
R1+4	work area minimum x coordinate (inclusive)
R1+8	work area minimum y coordinate (inclusive)
R1+12	work area maximum x coordinate (exclusive)
R1+16	work area maximum y coordinate (exclusive)

This call is similar to Wimp_RedrawWindow. The differences are:

- not all of the window has to be updated; you specify the rectangle of interest in work area coordinates
- the rectangles to be updated are not cleared by the Wimp first
- this can be called at any time, not just in response to a Redraw_Window_Request event

The routine exits via Wimp_GetRectangle (SWI &400CA), which returns the coordinates of the first visible rectangle (if any) within the work area specified on entry.

The code for the task to update the window should follow this scheme:

```
SYS"Wimp_UpdateWindow",,blk TO more
WHILE more
  update the contents of the returned rectangle
  SYS"Wimp_GetRectangle",,blk TO flag
ENDWHILE
```

A common reason for calling this is to drag an item across a window. Another is to draw a user-defined text cursor instead of using the system one.

Related SWIs

None

Related vectors

None

Wimp_GetRectangle (SWI &400CA)

On entry

R1 = pointer to block

On exit

R0 and block as for Wimp_RedrawWindow (SWI &400C8)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The block contains the following on entry:

R1+0	window handle

This call is used repeatedly following a call of either Wimp_RedrawWindow or Wimp_UpdateWindow. It returns the details of the next rectangle of the work area to be drawn (if any). If the call follows an earlier call to Wimp_RedrawWindow, then the rectangle is also cleared to the background colour of the window. If however it follows a call to Wimp_UpdateWindow then the rectangle's contents are preserved.

VDU 5 is asserted as a mode change and in Wimp_RedrawWindow. If you use VDU 4 text in a window (which can only be done when you are sure that the character does not need to be clipped) you should reset to VDU 5 mode before calling Wimp_SetRectangle or Wimp_Poll.

Note that the window handle will be faulted by the Wimp if it differs from the one last used when Wimp_RedrawWindow or Wimp_UpdateWindow was called. This means that a task must draw the whole of a window before performing any other operations.

Related SWIs

None

Related vectors

None

Wimp_GetWindowState (SWI &400CB)

On entry

R1 = pointer to block

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The block contains the window handle on entry, and the following on exit:

R1+0	window handle
R1+4	visible area minimum x coordinate
R1+8	visible area minimum y coordinate
R1+12	visible area maximum x coordinate
R1+16	visible area maximum y coordinate
R1+20	scroll x offset relative to work area origin
R1+24	scroll y offset relative to work area origin
R1+28	handle of window in front of this one (or –1 if none)
R1+32	window flags – see Wimp_CreateWindow (SWI &400C1)

This call returns a summary of the given window's state.

You can usually find out the window's coordinates without using this call, since Wimp_GetRectangle returns the window coordinates anyway. This call is most useful for reading the window flags, for example to find out if a window is uncovered.

Related SWIs

None

Related vectors

None

Wimp_GetWindowInfo (SWI &400CC)

On entry

R1 = pointer to block

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The block contains the following on entry:

R1+0	window handle

The block contains the following on exit:

R1+0	window handle
R1+4	window block – see Wimp_CreateWindow (SWI &400C1) and Wimp_CreateIcon (SWI &400C2)

This call returns complete details of the given window's state, including any icons that were created after the window, using Wimp_CreateIcon.

Related SWIs

None

Related vectors

None

Wimp_SetIconState (SWI &400CD)

On entry

R1 = pointer to block

On exit

The icon's flags are updated

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The block contains the following:

R1+0	window handle (–1 or –2 for icon bar)
R1+4	icon handle
R1+8	EOR word
R1+12	clear word

This call sets the given icon's flag word as follows:

```
new-state = (old-state AND NOT clear-word) EOR EOR-word
```

The way each bit of the icon flags is affected is controlled by the state of the corresponding bits in the EOR word and the Clear word:

Setting of CE	Effect
00	preserve the bit's status
01	toggle the bit's state
10	clear the bit
11	set the bit

For example, say you wanted to change an icon's button type (bits 12 - 15) to 10 (%1010 binary). You would set the clear-bits to 1 and the EOR bits to the new value:

Clear = %1111000000000000
EOR = %1010000000000000

The screen is automatically updated if necessary, so the call can be used to reflect a change in a text icon's contents. If you change the justification of a text icon using this call, and the icon owns the caret, you should also call Wimp_SetCaretPosition (SWI &400D2) to make sure that it remains positioned in the text correctly.

Related SWIs

None

Related vectors

None

Wimp_GetIconState (SWI &400CE)

On entry

R1 = pointer to block

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

On entry the block contains the following:

R1+0	window handle
R1+4	icon handle

On exit the block contains the following:

R1+0	window handle
R1+4	icon handle
R1+8	32-byte icon block – see Wimp_CreateIcon (SWI &400C2)

This call returns details of the given icon's state.

If you want to search for an icon with particular flag settings (for example to find out which icon in a group has been selected), you should use Wimp_WhichIcon (SWI &400D6).

Related SWIs

None

Related vectors

None

Wimp_GetPointerInfo (SWI &400CF)

On entry

R1 = pointer to block

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

On exit the block contains the following:

R1+0	mouse x
R1+4	mouse y
R1+8	button state
R1+12	window handle (–1 for background, –2 for icon bar)
R1+16	icon handle (see below)

This call returns information about the position of the pointer and the instantaneous state of the mouse buttons. It enables the task to find out where the mouse pointer is independently of the buttons being pressed or released, for example for dragging purposes.

The mouse button state (returned in R1+8 to R1+11) can only have bits 0, 1 and 2 set:

Bit	Meaning if set
0	Righthand button pressed (Adjust)
1	Middle button pressed (Menu)
2	Lefthand button pressed (Select)

If the mouse is over a user window (window handle >=0) then the icon handle will be either a valid non-negative value for a user icon, or one of the following system values:

Value	Icon
–1	work area
–2	Back icon
–3	Close icon
–4	Title Bar
–5	Toggle Size icon
–6	scroll up arrow
–7	vertical scroll bar
–8	scroll down arrow
–9	Adjust Size icon
–10	scroll left arrow
–11	horizontal scroll bar
–12	scroll right arrow
–13	the outer window frame

Related SWIs

None

Related vectors

None

Wimp_DragBox (SWI &400D0)

On entry

R1 <= 0 to cancel drag operation, otherwise
R1 = pointer to block

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

On entry the block contains the following:

R1+0	window handle (for drag types 1 - 4 only)
R1+4	drag type
R1+8	minimum x coordinate of initial position of drag box
R1+12	minimum y coordinate of initial position of drag box
R1+16	maximum x coordinate of initial position of drag box
R1+20	maximum y coordinate of initial position of drag box
R1+24	minimum x coordinate of parent box (for types 5 - 11 only)
R1+28	minimum y coordinate of parent box (for types 5 - 11 only)
R1+32	maximum x coordinate of parent box (for types 5 - 11 only)
R1+36	maximum y coordinate of parent box (for types 5 - 11 only)
R1+40	R12 value for user routine (for types 8 - 11 only)
R1+44	address of draw box routine (for types 8 - 11 only)
R1+48	address of remove box routine (for types 8 - 11 only)
R1+52	address of move box routine, or <= 0 if there isn't one (for types 8 - 11 only)

The coordinates are passed as screen coordinates, ie. bottom-left inclusive and top-right exclusive.

This call initiates a dragging operation. It is typically called as a result of a Mouse_Click event which has reported a drag-type click (ie Select or Adjust held down for longer than about 1/5th of a second). A drag spans calls to Wimp_Poll, so the task must maintain information about what is being

dragged, etc. Usually the coordinates are not required until the final drag event occurs, at which point the Wimp returns them. Sometimes Wimp_GetPointerInfo should be called in Wimp_Poll null events to track the pointer (especially for type 7 below). A drag is terminated (and reported) when the user releases all of the mouse buttons.

The drag is confined to the 'parent box' specified, or to an area computed by the Wimp for types 1 - 4. The action depends on the drag type:

Drag type	Meaning
1	drag window position
2	drag window size
3	drag horizontal scroll bar
4	drag vertical scroll bar
5	drag fixed size 'rotating dash' box
6	drag rubber 'rotating dash' box
7	drag point (no Wimp-drawn dragged object)
8	drag fixed size user-drawn box
9	drag rubber user-drawn box
10	as 8 but don't cancel when buttons are released
11	as 9 but don't cancel when buttons are released

Types 1 - 4

These are the 'system' types since they relate to picking up a window, changing its size and scrolling it respectively. In these cases, the bounding box for pointer movement is worked out automatically by the Wimp. For example, type 2 drags are confined to the defined maximum and minimum sizes of the window.

Bits in the `WimpFlags` CMOS configuration parameter determine the way in which these drags update the screen. There are four bits, 0 - 3, corresponding to drag types 1 - 4. If the bit is clear, then dragging is indicated by a dashed outline box, similar to that used in types 5 and 6 below. An Open_Window_Request event is generated when the mouse button is released to allow the task to update appropriate parts of the dragged window. If the `WimpFlags` bit is set, continuous update is required, and Open_Window_Requests are generated for every mouse move.

These drag types are useful if you want to allow the user to, for example, pick up a window which does not have a Title Bar (and so is usually unmoveable). You could detect clicks in a region of within, say, 32 OS units from the top of the visible work area and instigate a drag type 1 when these occur.

Types 5 - 7

These are 'user' types, where the task decides what the significance of the dragging will be. In these cases you supply the coordinates of the parent box. The box being dragged is constrained to this area. For types 5 and 6 the initial box position is used to draw a box with a dashed border which cycles round.

For type 5 boxes, the relative positions of the mouse pointer and the box are kept constant, so moving the mouse moves the box too.

For type 6, the relative positions of the bottom right corner of the box and the pointer are kept constant, so moving the mouse will increase or decrease the size of the box. Generally you would arrange the initial box coordinates such that this corner is at or near the pointer position reported in the drag-click event. You can alter the moveable corner to the left by reversing the initial x coordinates, and to the top by reversing the initial y coordinates.

In the case of type 7, where there is no dashed box to be dragged, the initial drag box position is ignored and the mouse coordinates are constrained to the bounding box.

Types 8 - 11

These types give the maximum flexibility for dragging objects around the whole screen. Use drag type 7 and Wimp_UpdateWindow to drag an object within a window. They are, though, somewhat more complex to use than the previously described types.

First the application must provide the addresses of three routines which draw, remove and move the user's drag item (it doesn't have to be a box). If no move routine is supplied ([R1+52] <= 0), the Wimp will use the remove and draw routines to perform the operation.

Note that the user code must not be in application space, but in the RMA. This is because the Wimp doesn't know to page the task in when this code is required.

The user code is called under the following conditions:

On entry

SVC mode (so use X-type SWIs and save R14_SVC before hand)
R0 = new minimum x coordinate
R1 = new minimum y coordinate
R2 = new maximum x coordinate
R3 = new maximum y coordinate
R4 = old minimum x coordinate (for move routine only)
R5 = old minimum y coordinate (for move routine only)
R6 = old maximum x coordinate (for move routine only)
R7 = old maximum y coordinate (for move routine only)
R12 = value supplied in Wimp_DragBox call

On exit

R0 - R3 actual box coordinates (normally preserved from entry)

The user routines would draw, remove or just move (ie remove and redraw) their drag object according to the coordinates passed. These coordinates are derived by the Wimp from mouse movements.

The graphics window is also set up by the Wimp. The user routines must not change this, or draw outside it.

While these drags are taking place, the Wimp still performs its rotating dashed box code, so the routines can take advantage of this. Programming of the VDU dot-dash pattern is performed by the Wimp, so all the user routines have to do is call the appropriate dot-dash line `PLOT` codes.

The move routine has to deal with two cases: whether the box has moved or not. If the box has moved (ie R0 - R3 are not identical to R4 - R7), then the move routine must exclusive-OR once using the old coordinates to remove the box, then EOR again with the new coordinates to redraw it. If the box hasn't changed, the the Wimp will have programmed the dot-dash pattern so that a single EOR plot will give the desired shifting effect of the pattern, so this is what the routine should do.

Of course, the foregoing is only applicable to dragged objects which use the dash effect. If you are dragging, say, a sprite, then the move routine only has to do anything when the coordinates have changed, viz restore the background that the sprite overwrote, then save the new background and replot the sprite. When no move has taken place, the routine could do nothing (or change the sprite for an animation effect etc.)

Related SWIs	None
Related vectors	None

Wimp_ForceRedraw (SWI &400D1)

On entry

R0 = window handle (–1 means whole screen)
R1 = minimum x coordinate of area to redraw
R2 = minimum y coordinate of area to redraw
R3 = maximum x coordinate of area to redraw
R4 = maximum y coordinate of area to redraw

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call forces an area of a window or the screen to be marked as invalid, and to be redrawn later using Redraw_Window_Request events.

If R0 is –1 on entry, then R1 - R4 specify an area of the screen in absolute co_ordinates. If R0 is not –1, then it indicates a window handle, and R1 - R4 specify an area of the window relative to the window's work area origin.

This call could be used

- to reconstruct the screen if for some reason it has been corrupted
- to reinstate a particular area after, for example, an error box has been drawn over the top of it
- to redraw the screen after redefining one or more of the soft characters, which could affect any part of the screen.

Two strategies are possible when the task is required to change the contents of a window. These are:

- call this routine, which causes the specified area to be redrawn later
- call Wimp_UpdateWindow (SWI &400C9), followed by the necessary graphic operations (and calls to Wimp_GetRectangle (SWI &400CA)).

The second method is generally quicker, but involves more code.

Related SWIs

None

Related vectors

None

Wimp_SetCaretPosition (SWI &400D2)

On entry

R0 = window handle (–1 to turn off and disown the caret)
R1 = icon handle (–1 if none)
R2 = x-offset of caret (relative to work area origin)
R3 = y-offset of caret (relative to work area origin)
R4 = height of caret (if –1, then R2, R3, R4 are calculated from R0,R1,R5)
R5 = index into string (if –1, then R4, R5 are calculated from R0,R1,R2,R3
R2 and R3 are modified to exact position in icon)

On exit

R0 - R5 = preserved

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call removes the caret from its old position, sets up the data for its new position, and redraws it there. Subsequent calls to Wimp_RedrawWindow and Wimp_UpdateWindow will cause the caret to be automatically redrawn by the Wimp, unless it is marked as invisible.

R4 and R5 can only be set to –1 if the icon handle passed in R1 is non-negative.

Some of the values may be calculated:

- If R4 (the height) is –1, the Wimp calculates the x and y coordinates of the caret and its height (R2, R3, R4) from the data in R0, R1 and R5. This is only possible if R1 contains an icon handle.
- Similarly, if R5 (the index) is –1, the Wimp calculates the index into the string and the caret height (R4, R5) from R0~ - ~R3.

In each case, the height of the caret is determined from the bounding box of the font used in the icon (for the system font, a height of 40 OS units is used). The caret's coordinates refer to the pixel at the bottom of the vertical bar. Note that the icon's bounding box and whether it has an outline are also considered.

The font height also contains some flags. Its full description is:

bits 0 - 15 height in OS units (0 - 65535)
bits 16 - 23 colour (if bit 26 is set)

Bit	Meaning when set
24	use VDU 5-type caret, else use anti-aliased caret
25	the caret is invisible
26	use bits 16 - 23 for the colour, else caret is Wimp colour 11
27	bits 16 - 23 are untranslated, else they are a Wimp colour

If bit 27 is set, then bit 26 must be set and the caret is plotted by EORing the logical colour given in bits 16 - 23 onto the screen. For the 256-colour modes, bits 16 - 17 are bits 6 - 7 of the tint, and bits 18 - 23 are the colour.

If bit 27 is clear, then the caret is plotted such that the Wimp colour given (or colour 11) appears when the background is Wimp colour 0 (white). The Wimp achieves this by EORing the actual colour for Wimp colour 0 and the caret colour together, then EORing this onto the screen.

Esoteric note: to ensure that the caret is plotted in a given colour on a non-white background, you must do the following:

- use Wimp_ReadPalette (SWI &400E5) to obtain the real logical colours associated with your background and caret (byte 0 of the entries)
- EOR these together
- put the result in bits 16 - 23 and set bits 26 and 27

Related SWIs

None

Related vectors

None

Wimp_GetCaretPosition (SWI &400D3)

On entry

R1 = pointer to block

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call returns details of the caret's state. The block contains the following:

R1+0	window handle where caret is (–1 if none)
R1+4	icon handle (–1 if none)
R1+8	x-offset of caret (relative to work area origin)
R1+12	y-offset of caret (relative to work area origin)
R1+16	caret height and flags or –1 for not displayed
R1+20	index of caret into string (if in a writeable icon)

The height and flags returned at R1+16 are as described under Wimp_SetCaretPosition (SWI &400D2).

Related SWIs

None

Related vectors

None

Wimp_CreateMenu (SWI &400D4)

On entry

R1 = –1 means close any active menu, or
R1 = pointer to menu block
R2 = x coordinate of top-left corner of top level menu
R3 = y coordinate of top-left corner of top level menu

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The menu block contains the following:

R1+0	menu title (if a null string, then the menu is untitled)
R1+12	menu title foreground and frame colour
R1+13	menu title background colour
R1+14	menu work area foreground colour
R1+15	menu work area background colour
R1+16	width of following menu items
R1+20	height of following menu items
R1+24	vertical gap between items
R1+28	menu items (each 24 bytes):

bytes 0 - 3 menu flags:

Bit	Meaning when set
0	display a tick to the left of the item
1	dotted line following (separates sections)
2	item is writeable for text entry
3	generate a message when moving to

	the submenu
7	this is the last item in this menu
all others	not used; must be zero

bytes 4 - 7 submenu pointer (>= &8000) or window handle (1 - &7FFF) (–1 if none)
bytes 8 - 11 menu icon flags – as for a normal icon
bytes 12 - 23 menu icon data (12 bytes) – as for a normal icon

This call is used to create a menu structure. The top level menu is initially displayed by the Wimp. Having made this call, the task must return to its normal polling loop. While the task calls Wimp_Poll, the Wimp maintains the menu tree, until the user clicks with any of the mouse buttons. If the click was outside the menus, then the Wimp closes all the menus and behaves as if they had not been there. If the mouse is clicked inside a menu, then a Menu_Selection reason code is returned from Wimp_Poll, along with a list of selections.

Note that the menu structure must remain intact as long as the tree is open. The Wimp does not take a copy, but uses it directly.

Pressing Return while the caret is inside a writeable item is equivalent to pressing a mouse button, ie it selects that item.

A menu is basically a window whose work area is entirely covered by the menu items. The work area colour bytes at R1+14 and R1+15 are therefore not generally used unless the 'gap between items' is non-zero; they are overridden by the items' icons colours. The window has a Title Bar if the string at R1+0 is non-null, otherwise it is untitled. The maximum length of the title string is the smaller of 12 and (item-width DIV 16). ie. it cannot be indirected. It should be terminated by a control code if the length is less than 12.

The menu will be automatically given a vertical scroll bar if it is taller than the current screen mode.

A menu item is a text icon whose bounding box is derived from width and height given at R1+16 and R1+20. Thus all entries in a menu are the same size. They are arranged vertically and lie horizontally between a 'tick' icon on the left and an arrow (submenu indicator) icon on the right, if present.

The menu item flags can alter the appearance of each item, eg by telling the Wimp to display the tick, or a separating dashed line beneath it. To shade an item, set bit 22 of the icon flags.

If the submenu pointer for an item is not –1, then it points to a similar data structure describing a submenu. An arrow is displayed to the right of the menu item; if the user moves the mouse pointer over this, then the submenu automatically pops up. Generally, submenu titles are the same as the parent item's text, or can be a prompt like 'Name:'.

The submenu pointer can be a window handle instead. Such a window is known as a dialogue box or dbox for short. In this case, the window is opened (as if it were a menu) when the mouse pointer moves over the arrow. The first writeable icon in the window is given the input focus. You cannot close a menu window by clicking in it or pressing Return. Instead you should give it an 'OK' icon and treat clicks over that as a selection. The menu can then be closed using Wimp_CreateMenu with R1=–1.

If you want Return to make a selection, use the key-pressed event.

Cancelling a menu-window can be achieved by clicking outside of the menu structure, or by providing a 'Cancel' icon for the user to click on. In the first case, no Close_Window_Request is returned for the window; it is closed automatically by the Wimp.

When a menu window is closed, the caret is automatically given back to wherever it was before the window was opened.

Bit 3 of the menu flags changes the submenu behaviour. If it is set, then moving over the right arrow will cause a MenuWarning message to be generated. The application can respond as it sees fit, usually by calling Wimp_CreateSubMenu (SWI &400E8) to display the appropriate object. Note that in this case the submenu pointer in the menu structure does not have to be valid, but it is passed to the application in the message block anyway. The submenu pointer is important if Wimp_DecodeMenu will be used later on.

Many of the iconic properties of menu items can be controlled, using the icon flags word and icon data bytes. Below is a list of the aspects of an icon that a menu item may or may not exhibit:

- it can contain text. Indeed it must in order to be useful (bit 0 must be set)

- it can contain a sprite, but see note below
- it can have a border, but this isn't particularly useful
- the text is always centred vertically (bit 4 ignored), but the horizontal formatting bits (3 and 9) are used
- the background should be filled (bit 5 set)
- the text can be anti-aliased
- the item is drawn only by the Wimp (bit 7 ignored)
- the icon be indirected – useful for long writeable item strings
- the button type is always 9 and the ESG is always 0 (bits 12 - 20 ignored). Use the menu flags to make an item writeable.
- the selected bit (21) isn't readable as the icon is 'anonymous'. The task hears about the final selection through the Menu_Selection event
- the shaded bit (22) is useful for disabling certain items. However, such items' submenu arrows can't be followed, so you should only shade leaf items
- the deleted bit (23) is irrelevant
- the colours/font handle byte (bits 24 - 31) should be set as appropriate.

The icon data contains either the actual text (0 to 12 characters, control-code terminated if less than twelve) or the three indirected icon information words. A validation string can naturally be used for writeable items.

A menu item can only usefully contain a sprite if it is a sprite-only (no text) indirected icon. This allows for a sprite control block pointer to be given in the middle word of the icon data. Typically this is +1 for a Wimp sprite, or a valid user-area pointer.

If the task can create more than one menu, it must remember which menu is displayed, as the Wimp does not return this when a selection has been made. It must also scan down its data structure to determine which submenus the numbers relate to, before it can decide what action to take. Wimp_DecodeMenu (SWI &400D5) can help with this.

It is recommended that tasks use a 'shorthand' for defining menus, which is translated into the full form required by the Wimp when needed. But menus must be held in semi-permanent data structures once created, since the Wimp accesses them while menus are open.

Note that if a menu selection is made using Adjust, it is conventional for the application to keep the menu structure open afterwards. What happens is that the Wimp marks the menu tree temporarily when a selection is made. The application should call Wimp_GetPointerInfo to see if Adjust is pressed. If so, it should call Wimp_CreateMenu before returning to Wimp_Poll, which causes the tree to be re-opened in the same place.

The menu structure may be modified before re-opening, in which case any changes are noted by the Wimp, for example if menu entries become shaded. If the application does not call Wimp_CreateMenu, then the Wimp will delete the menu tree on the next call to Wimp_Poll, as the tree was marked temporary when the selection was made.

See the **Application Notes** section for information about the standard colours and sizes used for menus.

Related SWIs

None

Related vectors

None

Wimp_DecodeMenu (SWI &400D5)

On entry	R1 = pointer to menu data structure R2 = pointer to a list of menu selections R3 = pointer to a buffer to contain the answer
On exit	buffer updated to contain menu item text, separated by '.'s
Interrupts	Interrupts are not defined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This call converts a numerical list of menu selections to a string containing the text of each successive menu item, eg `Display.Small icons` for a typical Filer menu selection.
Related SWIs	None
Related vectors	None

Wimp_WhichIcon (SWI &400D6)

On entry

R0 = window handle
R1 = pointer to block to contain the list of icon handles
R2 = bit mask (bit set means consider this bit)
R3 = bit settings to match

On exit

block at R1 updated to contain a list of icon handle words, terminated by –1

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call compares the flag words of all of the icons belonging to the given window with the pattern given in R3. Each icon whose flags match has its handle added to the block pointed to by R1.

The mask in R2 is used to determine which bits are to be used in the comparison. The icon's handle is added to the list if (icon-flags AND bit-mask) = (bit-settings AND bit-mask). For example:

```
SYS "Wimp_WhichIcon",window,buffer,1<<21,1<<21
```

On exit a list of icon handles whose selected bit (21) is set will be in the buffer. Similarly, to see which is the first icon with ESG number 1 that is selected:

```
SYS "Wimp_WhichIcon",window,buffer,&003F0000,&00210000
```

!buffer now contains the handle of the required icon, or –1 if none is selected.

Related SWIs

None

Related vectors

None

Wimp_SetExtent (SWI &400D7)

On entry

R0 = window handle
R1 = pointer to block

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

On entry, the block contains:

R1 + 0	new work area minimum x
R1 + 4	new work area minimum y
R1 + 8	new work area maximum x
R1 + 12	new work area maximum y

This call sets the work area extent of the specified window, and usually causes the window's scroll bars to be redrawn (to reflect the new total size of window). The work area extent may not be changed so that any part of the visible work area lies outside the extent, so this call cannot change the current size of a window, or cause it to scroll.

It is usual to make this call when a document has been extended, eg by text being inserted into a word-processor.

Note that you must set the extent to be a whole number of pixels. If not, strange effects can occur, such as the pointer moving beyond its correct bounding box. If you do this, the Wimp automatically readjusts the extent on a mode change.

Related SWIs

None

Related vectors

None

Wimp_SetPointerShape (SWI &400D8)

On entry

R0 = shape number (0 for pointer off)
R1 = pointer to shape data (-1 for no change)
R2 = width in pixels (must be multiple of 4)
R3 = height in pixels
R4 = active point x offset from top-left in pixels
R5 = active point y offset from top-left in pixels

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The shape data is a series of bytes giving the pixel colours for the shape. Each row of the shape is given as a whole number of bytes (eg 3 bytes for a 12-pixel wide shape). Bytes are given in left to right order. The least significant two bits of each byte give the colour of the leftmost pixel in that group of four (ie it looks backwards as you write it down in binary).

In new programs, you should now use the call Wimp_SpriteOp (SWI &400E9) with R0=36 (SetPointerShape) instead of this one. The following principles still apply though.

This convention should be used when programming the pointer shape under the Wimp:

- shape 1 is the default arrow shape (set-up by *Pointer)
- to use an alternative, define and use shape 2
- when the pointer leaves the window where it was changed, it should be reset to shape 1.

The reason codes Pointer_Entering_Window and Pointer_Leaving_Window returned from Wimp_Poll are very useful for deciding when to reprogram the pointer shape.

If you want to use Wimp_SpriteOp for all pointer shape programming, and wish to avoid using *Pointer, you can use the Wimp sprite `ptr_default` to program the standard arrow shape. Note however that `ptr_default` does not have a palette, so you would have to reset the pointer palette too if your pointer shape changed it.

Related SWIs

None

Related vectors

None

Wimp_OpenTemplate (SWI &400D9)

On entry	R1 = pointer to template pathname to open
On exit	—
Interrupts	Interrupts are not defined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This causes the Wimp to open the template file given, and to read in some header information from the file. Only one template file may be open at a time; this is the one used by Wimp_LoadTemplate (SWI &400DB) when that SWI is called.
Related SWIs	None
Related vectors	None

Wimp_CloseTemplate (SWI &400DA)

On entry	—
On exit	—
Interrupts	Interrupts are not defined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This closes the currently open template file.
Related SWIs	None
Related vectors	None

Wimp_LoadTemplate (SWI &400DB)

On entry

R1 = pointer to user buffer for template
R2 = pointer to workspace for indirected icons
R3 = pointer to end of workspace
R4 = 256-byte font reference array (–1 for no fonts)
R5 = pointer to (wildcarded) name to match (must be 12 bytes word-aligned)
R6 = position to search from (0 for first call)

On exit

R2 = pointer to remaining workspace
R6 = position of next entry (0 if no match found)
The template is at R1
The font array is updated if fonts were used
The string at R5 is overwritten by the actual name (so at least 12 bytes must be available there)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The space required by the buffer passed in R1 is 88 bytes for the window, 32 bytes for each icon and room for all indirected data. This indirected data is then copied by the Wimp into the area pointed to by R2.

Window templates are created by the template creation utility (FormEd). They are stored in a file, and each template has a name associated with it. Because the search name may be wildcarded, it is possible to search for all templates of a given form (eg dialog*) by calling Wimp_LoadTemplate with R6=0 the first time, then using the value passed back for subsequent calls. R6 will be returned as 0 on the call after the last template is found. As the wildcarded name is overwritten by the actual one found, it must be re-initialised before every call and must be big enough to have the template name written into it.

The indirected icon workspace pointer is provided so that when the window definition is read into the buffer addressed by R1, its icon fields can be set correctly. An indirected icon's data is read from the file into the workspace addressed by R2, and the icon data pointer fields in the window definition are set appropriately. R2 is updated, and if it becomes greater than R3, a `Window definition won't fit` error is given.

The font reference count array is used to overcome the problem caused with dynamically allocated font handles. When a template file is created, font information such as size, font name etc is stored along with the font handle that was returned for the font in FormEd. When a template is subsequently loaded, the Wimp calls Font_FindFont and replaces references to the original font number with the new handle. It then increments the entry for that handle in the reference array. This array should be initialised to zero before the first call to Wimp_LoadTemplate.

When a window is deleted, for all font handles in the range 1 - 255 you should call Font_LoseFont the number of times given by that font's reference count. This implies that a separate 256-byte array is needed for each template loaded. However, this can be stored a lot more compactly (eg using font handle/count byte pairs) once the array has been set up by Wimp_LoadTemplate.

An alternative is to have a single reference count array for all the windows in the task, and only call Font_LoseFont the appropriate number of times for each handle when the task terminates.

Related SWIs

None

Related vectors

None

Wimp_ProcessKey (SWI &400DC)

On entry

R0 = character code

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call has two uses. The first is to make the Wimp return a Key_Pressed event as though the character code passed in R0 was typed by the user. It is useful in programs where a menu of characters corresponding to those not immediately available from the keyboard is presented to the user, and clicking on one of them causes the code to be entered as if typed.

The second use is to pass on a keypress that a task does not understand, so that other applications (with the 'hot key' window flag set) may act on on it. The key is passed (via the Key_Pressed event) to each eligible task in turn, from the top of the window stack down. It stops when a task fails to call Wimp_ProcessKey (because it recognises the key), or until the bottom window is reached.

For this to work, it is vital that a task always passes on unrecognised key presses using Wimp_ProcessKey. Conversely, if the program can act on the key stroke, it should not then call Wimp_ProcessKey, as this might result in a single key stroke causing several separate actions.

As a last resort, if no task acts on a function key press, the Wimp will expand the code into the appropriate function key string and insert it into the writeable icon that owns the caret, if any.

Related SWIs

None

Related vectors

None

Wimp_CloseDown (SWI &400DD)

On entry

R0 = task handle returned by Wimp_Initialise (only required if R1="TASK")
R1 = "TASK" (see Wimp_Initialise &400C0)

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call must be made immediately before the task terminates by calling OS_Exit. If this was the only extant task, the Wimp will reset the soft key and mode settings to their original values (ie as they were before Wimp_Initialise was first called). Any application memory used by the task will be returned to the Wimp's free pool.

If the task handle is not given, then the Wimp will close down the currently active task, ie the one which was the last to have control returned to it from Wimp_Poll. This is sufficient if the task is loaded in the application workspace (as opposed to being a relocatable module).

Module tasks should always pass their handle to Wimp_CloseDown, as there is no guarantee that the module in question is the active one at the time of the call. For example, a task module would be required to close down in its 'die' code, which may be called asynchronously without control passing to the module through Wimp_Poll.

A Wimp_CloseDown will cause the service call WimpCloseDown (&53) to be generated. See the section **Relocatable module tasks** for details.

Related SWIs

None

Related vectors

None

Wimp_StartTask (SWI &400DE)

On entry

R0 = pointer to * Command to be executed

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is used to start a 'child' task from within another program. The text pointed to by R0 on entry can be any * Command which will cause a Wimp program to be executed, eg `BASIC -quit myProg`.

The Wimp will create a new 'domain' or environment for the task and calls OS_CLI to execute the command. If the new task subsequently calls Wimp_Initialise and then Wimp_Poll, control will return to caller of Wimp_StartTask. Alternatively, control will return when the new task terminates through OS_Exit (which `QUIT` in BASIC calls).

This call is used by the Desktop and the Filer to start new tasks.

Note that you can only call this SWI:

- if you are already a 'live' Wimp task, and have gained control from Wimp_Initialise or Wimp_Poll.
- you are in USR mode.

Related SWIs

None

Related vectors

None

Wimp_ReportError (SWI &400DF)

On entry

R0 = pointer to standard error block, see below
R1 = flags, see below
R2 = pointer to application name for error window title (< 20 characters)

On exit

R1 = 0 if no key click, 1 if OK selected, 2 if `Cancel` selected

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The format of a standard error block is:

R0+0	error number
R0+4	zero-terminated error string

This call provides a built-in means for reporting errors that may occur during the running of a program. The error number and its text is pointed to by R0. The control code-terminated string pointed to by R2 is used in the Title Bar of the error window, optionally preceded by the text `Error from` .

The flags in R1 on entry have the following meanings:

Bit	Meaning when set
0	provide an OK box
1	provide a Cancel box
2	highlight Cancel (or OK if bit is cleared)
3	if the error is generated while a text-style window is open (eg within a call to Wimp_CommandWindow), then don't produce the prompt Press SPACE or click mouse to continue, but return immediately
4	don't prefix the application name with Error from in the error window's Title Bar
5	if neither box is clicked, return immediately with R1=0 and leave the error window open
6	select one of the boxes according to bits 0 and 1, close the window and return
7 - 31	reserved; must be 0

If neither bit 0 or 1 is set, an OK box is provided anyway. Bits 5 and 6 can be used to regain control while the error window is still open, for example to implement time-outs (for example, the disc insert box, which polls the disc drive to see if a disc has been inserted), or use keypresses to stand for clicks on either of the boxes. Note though that the Wimp should not be re-entered while an error window is open, so you should always call Wimp_ReportError with bit 6 of R1 set before you next call Wimp_Poll, if you are using bit 5 in this way.

Wimp_ReportError causes the service WimpReportError (&57) to be generated. See the section **Relocatable module tasks** for details.

Note that Escape currently always returns 1 (ie OK clicked), instead of whichever box is highlighted.

Related SWIs

None

Related vectors

None

Wimp_GetWindowOutline (SWI &400E0)

On entry

R1 = pointer to a five-word block

On exit

The block is updated

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

On entry, R1+0 contains the window handle; on exit the block is updated thus:

R1+0	window handle
R1+4	minimum x coordinate of window bounding box
R1+8	minimum y coordinate of window bounding box
R1+12	maximum x coordinate of window bounding box
R1+16	maximum y coordinate of window bounding box

The Wimp supplies the x0,y0 inclusive, x1, y1 exclusive coordinates of a rectangle which completely covers the specified window, including its border. This call is useful when you want, for example, to set a mouse rectangle to the same size as a window.

Note that this call will only work after a window is opened, not just created.

Related SWIs

None

Related vectors

None

Wimp_PollIdle (SWI &400E1)

On entry

R0 = mask (see Wimp_Poll)
R1 = pointer to 256 byte block (used for return data; see Wimp_Poll)
R2 = earliest time for return with Null_Reason_Code event

On exit

see Wimp_Poll (SWI &400C7)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call performs the same task as Wimp_Poll. However, the caller also specifies an OS_ReadMonotonicTime-type time on entry. The call will not return before then, unless there is a non-null event to be processed. Effectively the caller can 'sleep', not being woken up until the specified time has passed or until it has some action to perform. This gives more processing time to other tasks.

Having performed the appropriate action upon return, the task should add its 'time-increment'; (eg 100 for a one-second granularity clock) to the previous value it passed in R2 and call Wimp_PollIdle again.

Note that if the Wimp is suspended for a while (eg. the user goes into the command prompt) and then returns, it is possible for the current time to be much later than the 'earliest return' time.

For this reason, it is recommended that (for example) a clock task should cater for this by incorporating the following structure:

```
SYS"OS_ReadMonotonicTime" TO newtime
WHILE (newtime - oldtime) > 0
       oldtime=oldtime+100
ENDWHILE
REM Then pass oldtime to Wimp_PollIdle
```

Related SWIs	None
Related vectors	None

Wimp_PlotIcon (SWI &400E2)

On entry

R1 = pointer to an icon block (see below)

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call can be used to plot an icon in a window during a window redraw or update loop. The icon doesn't exist as part of the window's definition. Instead, the data to be used to plot the icon is passed explicitly through R1. The format of the block is the same as that used by Wimp_CreateIcon (SWI &400C2), except that there is no window handle associated with it (this being implicitly the window which is currently being redrawn or updated):

R1+0	minimum x coordinate of icon bounding box
R1+4	minimum y coordinate of icon bounding box
R1+8	maximum x coordinate of icon bounding box
R1+12	maximum y coordinate of icon bounding box
R1+16	icon flags
R1+20	icon data

See Wimp_CreateIcon for details about these fields.

Related SWIs

None

Related vectors

None

Wimp_SetMode (SWI &400E3)

On entry	R0 = mode number
On exit	—
Interrupts	Interrupts are not defined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This call changes the display mode used by the Wimp. It should not be used by applications (which should be able to work in any mode), unless absolutely necessary. Its main client is the palette utility, which allows the user to change mode as required. In addition to changing the mode this call resets the palette according to the number of colours in the new mode, reprograms the mouse pointer appropriately and re-allocates the screen memory to use the minimum required for this mode. In addition, the screen is rebuilt (by asking all tasks to redraw their windows) and tasks are informed of the change through a Wimp_Poll message. Notes: the new mode is remembered for the next time the Wimp is started, but does not affect the configured Wimp mode, so this will be used after a hard reset or power-up. If there is no active task when Wimp_SetMode is called, the mode change doesn't take place until Wimp_Initialise is next called.
Related SWIs	None
Related vectors	None

Wimp_SetPalette (SWI &400E4)

On entry

R1 = pointer to 20-word palette block

On exit

R1 = preserved

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The block pointed to by R1 contains the following on entry:

R1+0	Wimp colour 0 RGB value
R1+4	Wimp colour 1 RGB value
R1+8	Wimp colour 2 RGB value
...	...
R1+56	Wimp colour 14 RGB value
R1+60	Wimp colour 15 RGB value
R1+64	border colour RGB value
R1+68	pointer colour 1 RGB value
R1+72	pointer colour 2 RGB value
R1+76	pointer colour 3 RGB value

Each RGB value word has the format &BBGGRR00, ie. bits 0 - 7 are reserved, and should be 0, bits 8 - 15 are the red value, bits 16 - 23 the green and bits 24 - 31 the blue, as used in a VDU 19,l,16,r,g,b command. The call, whose main user is the palette utility, issues the appropriate palette VDU calls to reflect the new values given in the 20-word block. In modes other than 16-colour ones, a remapping of the Wimp's colour translation table may be required, necessitating a screen redraw. It is up to the user of Wimp_SetPalette to cause this to happen (the palette utility does). Tasks are informed of palette changes through a message event returned by Wimp_Poll.

Related SWIs

None

Related vectors

None

Wimp_ReadPalette (SWI &400E5)

On entry

R1 = pointer to 20-word palette block

On exit

R1 = preserved

'nterrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

'Use

The 20-word block is updated in the format described under Wimp_SetPalette (SWI &400E4). However, the bottom byte of the first 16 entries contains the logical colour number that is used for that Wimp colour. This is the same as the Wimp colour in 16-colour modes. In 256 colour modes, bits 0 and 1 are bits 6 and 7 of the tint, and bits 2 - 7 are the GCOL colour.

Applications can use this call to discover all of the current Wimp palette settings.

Related SWIs

None

Related vectors

None

Wimp_SetColour (SWI &400E6)

On entry

R0 = colour and GCOL action (see below)

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The format of R0 is as follows:

Bits	Meaning
0 - 3	Wimp colour
4 - 6	GCOL action
7	0 for foreground, 1 for background

This calls is used to set the current graphics foreground or background colour and action to one of the 16 standard Wimp colours. As described earlier, these map into ECF patterns in monochrome modes, four grey-level colours in four-colour modes, the available colours in 16-colour modes, and the closest approximation to the Wimp colours in 256-colour modes.

After the call to Wimp_SetColour, the appropriate GCOL, TINT and (in two-colour modes) ECF commands will have been issued. The Wimp uses ECF pattern 4 for its purposes.

Related SWIs

None

Related vectors

None

Wimp_SendMessage (SWI &400E7)

On entry

R0 = reason code (as returned by Wimp_Poll – often 17, 18 or 19)
R1 = pointer to message block
R2 = task handle of destination task, or

R2 = window handle; message sent to window's creator, or

R2 = –2 (icon bar) and
R3 = icon handle; message sent to icon's creator, or

R2 = 0; broadcast message, sent to all tasks, including the originator

On exit

R2 = task handle of destination task (except for broadcast messages)
the message is queued
the message block is updated (reason codes 17 and 18 only)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

All messages within the Wimp environment are generated using this call. The Wimp uses it internally to keep tasks informed about various events through their Wimp_Poll loop.

User tasks can also generate these types of message, with reason codes in the range 0 to 12. On entry, R1 should point to a block with the format described under Wimp_Poll (SWI &400C7). For example, if you send an Open_Window_Request to a task (R0=2), you should point R1 at a Wimp_OpenWindow (SWI &400C5) block.

More often though, Wimp_SendMessage is used by tasks to send events of type User_Message to one another. These differ from the 'system' types, in that the Wimp performs some special actions, eg filling in fields of the message block, and noting whether a reply has been received.

There are three variations, depending on the reason code in R0 on entry. The first two, User_Message and User_Message_Recorded (17 and 18), send a message to the destination task(s). The latter expects the message to be acknowledged or replied to, and if it isn't the Wimp returns the message to the sender. (See Wimp_Poll event codes 17, 18 and 19.)

Reason code User_Message_Acknowledge (19) is used to acknowledge the receipt of a message without actually generating an event at the destination task. The receiver copies the my_ref field of the message block into the your_ref field and returns the message using the task handle of the sender given in the message block. If you acknowledge a broadcast message, it is not passed on to any other tasks.

The format of a user message block is:

R1+0	length of block, 20 - 256 bytes, a whole number of words
R1+4	not used on entry
R1+8	not used on entry
R1+12	your_ref (0 if this is an original message, not a reply)
R1+16	message action
R1+20	message data (format depends on the message action)
...	

Note that the block length should include any string that appears on the end (eg pathnames), including the terminating character, and rounded up to a whole number of words.

On exit the block is updated as follows:

R1+4	task handle of sender
R1+8	my_ref (unique Wimp-generated non-zero positive word)

Thus the receiver of the message will know who sent the message (useful for acknowledgements) and will also have a reference that can be quoted in replies to the sender. Naturally the sender can also use these fields once the Wimp has filled them in.

Note that you can use User_Message_Acknowledge to discover the task handle of a given window/icon by calling Wimp_SendMessage with R0=19, your_ref = 0, and R2/R3 the window/icon handle(s). On exit R2 will contain the task handle of the owner, though no message would actually have been sent.

Message actions

The following is a description of the currently defined message actions. Some of these are system types, others are generated by particular modules (most notably the Wimp). Any other module or application can send its own private messages, as required. A module is allowed to use its SWI chunk number as a base for the message action values. If you require a message action chunk and do not have a SWI chunk allocated, contact Acorn Computers Customer Support.

System messages

Message_Quit 0

On receiving this broadcast message a task should tidy up (close files, de-allocate memory etc) and close down by calling Wimp_CloseDown (SWI &400DD) and OS_Exit. The task doesn't have any choice about closing down at this stage. Any objections (because of unsaved data etc) should be lodged when it gets the Message_PreQuit (8) described below.

Message_DataSave 1 – Message_RAMTransmit 7

See the section entitled **Data transfer protocol** for details of these message actions.

Message_PreQuit 8

This broadcast message gives applications the chance to object to a request to close down, if for example they have modified data which has not been saved. If the task does not mind terminating, it should ignore this message, and eventually a Message_Quit will be received.

To object to the potential closedown, the task should acknowledge the message by calling Wimp_SendMessage with:

R0 = User_Message_Acknowledge (19)
R1 = as returned by Wimp_Poll
R1+12 = R1+8 (ie my_ref copied into your_ref)

Following this, the task should display a dialogue box giving the user the chance to either save or discard files, as he sees fit.

Note that if the user subsequently selects OK (ie. discard the data and quit anyway), the task must restart the closedown sequence by issuing a key-pressed event (Ctrl-Shift-F12) to the task which sent it the PreQuit message:

```
SYS "Wimp_GetCaretPosition",,blk
blk!24=&1FC
SYS "Wimp_SendMessage",8,blk,quitsender

where quitsender is read from sender field of original PreQuit message.

The Task Manager uses the Quit and PreQuit messages when the user selects the Exit
option from its menu. The way in which this works (in pseudo-BASIC) is as follows:

REM in CASE statement for Wimp_Poll event type...
WHEN Menu_Selection : PROCdecodeMenu
IF menuChoice$="Exit" THEN
  REM send the PreQuit and remember my_ref
  SYS "Wimp_SendMessage",User_Message_Recorded,PreQuitBlock,0
  PreQuitRef = PreQuitBlock!8
ENDIF
WHEN User_Message_Acknowledge
  REM got one of our messages back. Is it the PreQuit one?
  IF pollBlock!8 = PreQuitRef THEN
    REM no-one objected to PreQuit so safe to issue quit
    SYS"Wimp_SendMessage",User_Message_Recorded,quitBlock,0
    quitRef=quitBlock!8
  ELSE REM is it the quit one then?
    REM if so, exit the Desktop
    IF pollBlk!16=Message_Quit AND pollBlk!8=quitRef THEN quit
  ENDIF

WHEN User_Message, User_Message_Recorded
  REM if someone else did a quit, then terminate desktop
  IF pollBlk!16=Message_Quit AND pollBlk!8<>quitRef THEN quit
...
```

In English, the Task Manager issues a PreQuit broadcast when the Exit item is selected from its menu. If this is returned by the Wimp (because no other task objected), the Task Manager goes ahead and issues a Quit broadcast. When this comes back unacknowledged, the Task Manager checks the reference and quits if it is correct (as all other tasks would already have done).

The Task Manager must also be able to respond to the key-pressed event (Ctrl-Shift-F12) &1FC.

Tasks should automatically restart the quit procedures as described earlier.

Finally, if the Task Manager ever gets a Quit that it didn't originate, it will close itself down.

Message_PaletteChange 9

This broadcast message is issued by the Palette utility. It should not be acknowledged. The utility generates it when the user finishes dragging one of the RGB bars for a given colour, or when a new palette file is loaded.

If a task needs to adapt to a change in the physical colours on the screen, it should respond to this message by changing any of its internal tables (colour maps etc), and then call Wimp_ForceRedraw to ensure that its windows are redrawn with the new colours. Note though that the palette utility automatically forces a redraw of the whole screen if any of the Wimp's standard colours change their logical mapping, so applications don't have to take further action.

This message is not issued when the Wimp mode changes; Message_ModeChange (&400C1) reports this, so tasks interested in colour mapping changes should recognise this message too.

Filer messages

Message_FilerOpenDir &400

A task sends this message to a Filer task. It is a request to open a new directory display. The data part of the message block is as follows:

R1+20	filing system number
R1+24	must be zero (reserved for flags)
R1+28	full name of directory to view, zero-terminated

The string given at R1+28 must be a full specification of the directory to open including fileserver (if appropriate), disc name, and pathname starting from $, using the same format as the names in Filer windows. Send the message as a broadcast User_Message. If the directory name is invalid (eg the filing system is not present), a Wimp_ReportError error will be generated by the Filer.

Note that the Filing System modules (eg. ADFSFiler) do not use a broadcast, but instead discover the Filer's task handle by means of the Service_StartFiler protocol. See the section on **Relocatable Module tasks** for further details.

Message_FilerCloseDir &401

This message takes the same form as the previous one. All open directory displays whose names start with the name given at R1+28 are closed.

NetFiler message

Message_Notify &40040

The NetFiler sends this broadcast message to enable an application to display the text of a *Notify command in some pleasing way. If no-one acknowledges the message, NetFiler simply displays the text in a window using Wimp_ReportError, with the string `Message from station xxx.xxx` in the Title Bar.

Information about the sender, and the text of the notify, are contained in the message block, as follows:

R1+20	sending station number
R1+21	sending station network number
R1+22	LSB of five-byte real time on receipt of message
R1+23	second byte of time
R1+24	third byte of time
R1+25	fourth byte of time
R1+26	MSB of five-byte real time on receipt of message
R1+27	message text, terminated by a zero byte

So if you want to do something with the notify and prevent the NetFiler from displaying it, copy the my_ref field into the your_ref field and send the message back using Wimp_SendMessage User_Message_Acknowledge (19).

Wimp messages

Message_MenuWarning &400C0

The Wimp sends this message when the mouse pointer travels over the right arrow of a menu item to activate a submenu. The menu item must have its 'generate message' bit (3) in the menu flags set for this to happen, otherwise the Wimp will just open the submenu item as normal. (The submenu pointer must also be greater than zero in order for this message to be sent.)

In the message block are the values required by Wimp_CreateSubMenu (SWI &400E8) on entry. The task may use these, or may choose to take some other action (eg create a new window and open that as the submenu).

R1+20	submenu pointer from menu item
R1+24	x coordinate of top left of new submenu
R1+28	y coordinate of top left of new submenu
R1+32	main menu selected item number (0 for first)

R1+36	first submenu selected item number
...	
R1+...-	1 to terminate list

After the three words required by Wimp_CreateSubMenu is a description of the current selection state, in the same format that would be returned by the Menu_Selection event. This information, in conjunction with the task's knowledge of the menu structure, is sufficient to work out the path taken through the menu so far.

Message_ModeChange (&400C1)

Wimp_SetMode (SWI &400E3) causes this message to be sent as a broadcast. It gives tasks a chance to update their idea of what the current screen mode looks like by reading the appropriate parameters using OS_ReadVduVariables (SWI &31). (Though applications should need to know as little about the display's attributes as possible to facilitate mode independence.)

You should not acknowledge this message.

After sending the message, the Wimp generates an Open_Window_Request event for each window that was active when the mode change occurred. This is because going from a wider to a narrower mode (eg 16 to 12) may require the horizontal coordinates of windows to be compressed to fit them all on to the new display. The whole screen area is also marked invalid to force a redraw of each window's contents.

You should take care if, on a mode change, you modify a window in a way that involves deleting it and then recreating with different attributes. This will result in the handle of the window changing just after the Wimp scans the window stack and generates the Open_Window_Request for it, but before it is delivered from Wimp_Poll, and the Wimp will use the wrong handle. In this situation, you should internally mark the window as 'to be recreated' on receipt of the ModeChange message, and then when you receive the Open_Window_Request for that window, carry out the delete/recreate/open action then.

Message_TaskInitialise &400C2

This message is broadcast whenever a task calls Wimp_Initialise. It is used by the Task Manager to maintain its list of active tasks. Information in the message block is as follows:

R1+4	new task handle (so it appears that the new task sent the message)
...	
R1+20	CAO (current active object) pointer of new task
R1+24	amount of application memory used by the task
R1+28	task name, as given to Wimp_Initialise, zero-terminated

Message_TaskCloseDown &400C3

This performs a similar task to the one above, keeping the Task Manager (and any other interested parties) informed about the state of a task. It is generated by the Wimp on the task's behalf when it calls Wimp_CloseDown. If a program 'accidentally' calls OS_Exit before calling Wimp_CloseDown, the Wimp will perform the latter action for it. The message block is standard except for

R1+4	dying task's handle

ie the Wimp makes it look as though the task sent the message itself.

Message_SlotSize &400C4

This broadcast is issued whenever Wimp_SlotSize is called. Again, its primary client is the task manager, enabling that program to keep its display up to date. The message block looks like this:

R1+4	handle of the task which owns the current slot
...	
R1+20	new current slot size
R1+24	new next slot size

As with most broadcast messages, you should not acknowledge this one.

Message_SetSlot &400C5

This message has two uses. First it allows the Task Manager to discover if an application can cope with a dynamically varying slot size. Second, it is used by the Task Manager to tell a task to change that size if it can.

The message block contains the following:

R1+20	new current slot size
R1+24	handle of task whose slot should be changed

The receiver should check the handle at R1+24, and the size at R1+20. If the handle is not the task's, it should do nothing (ie no acknowledgement).

If the slot size is big enough for the task to carry on running, it should set R0 to this, R1 to –1 and call Wimp_SlotSize (SWI &400EC). It should then acknowledge the message.

If the slot size is too small for the task to carry on running, it should not call Wimp_SlotSize, but should acknowledge the message if it wants to continue to receive these messages. If ever a Message_SetSlot is not acknowledged, the Task Manager makes that task an undraggable one on its display.

You should be prepared to receive negative values for the slot size (which of course you shouldn't pass to Wimp_SlotSize), so do a proper signed comparison when checking the value in R1+20.

Message_TaskNameRq &400C6

This forms the first of a pair of messages that can be used to find the name of a task given the handle. An application should broadcast this message. It will be picked up by the Task Manager, if running. The Task Manager will respond with a TaskNameIs message (see below). The message block should contain the following information:

R1+20	handle of task whose name is required

Message_TaskNameIs &400C7

The Task Manager responds to a TaskNameRq message by sending this message. The message block contains the following:

R1+20	handle of task whose name is required
R1+24	task's slot size
R1+28	task's Wimp_Initialise name, zero-terminated

The principle user of this message-pair is the !Help application in providing help about ROM modules.

Data transfer protocol

The message-passing system is central to the transfer of data around the Wimp system. This covers saving files from applications, loading files into applications, and the direct transfer of data from one application to another. The last use often obviates the need for a 'scrap' (cut and paste) mechanism for intermediate storage; data is sent straight from one program to another, either via memory or a temporary file.

Data transfer code uses an environment variable called `Wimp$Scrap` to obtain the name of the file which should be used for temporary storage. This is set by the file `!System.!Boot`, when a directory display containing the `!System` directory is first displayed. Applications attempting data transfer should check that `Wimp$Scrap` exists. If it doesn't, they should report the error `Wimp$Scrap not defined`.

Four main message types exist to enable programs to support file/data transfer. The protocol which uses them has been designed so that a save to file operation looks very similar to a data transfer to another application. Similarly, a load operation bears much similarity to a transfer from another program. This minimises the amount of code that has to be written to deal with all possibilities.

The messages types are:

Message_DataSave	1
Message_DataSaveAck	2
Message_DataLoad	3
Message_DataLoadAck	4

There are three others which have associated uses: Message_DataOpen, Message_RamFetch and Message_RamTransmit. Before describing the message types in detail, we describe the four data transfer operations.

Note that all messages except for the initiating one should quote the other side's my_ref field in the message's your_ref field, as is usual when replying.

Saving data to a file

This is initiated through a Save entry in a task's menu. This item will have a standard dialogue box, with a 'leaf' name and a file icon which the user can drag to somewhere on the desktop, in this case a directory window. The following happens:

- The user releases the mouse button, terminating the drag of the file icon; the application receives a User_Drag_Box event.
- The application calls Wimp_GetPointerInfo (SWI &400CF) to find out where the icon was dropped, in terms of its coordinates and window/icon handles.
- The application sends a DataSave message with the file's leafname to the Filer using this information.

- The Filer replies with a DataSaveAck message, which contains the complete pathname of the file.
- The application saves the data to that file.
- The application sends the message DataLoad to the Filer.
- The Filer replies with the message DataLoadAck.

The last two steps may seem superfluous, but they are important in keeping the application-Filer and application-application protocol the same.

Saving data to another application

This is initiated in the same way as a Filer save. The following happens:

- The user releases the mouse button, terminating the drag of the file icon; the application receives a User_Drag_Box event.
- The application calls Wimp_GetPointerInfo to find out where the icon was dropped, in terms of its coordinates and window/icon handles.
- The application sends a DataSave message with the file's leafname to the destination application using this information.
- The destination application replies with a DataSaveAck message, which contains the pathname `<Wimp$Scrap>`.
- The application saves the data to that file (which the filing system expands to an actual pathname).
- The application sends the message DataLoad to the destination task.
- The external task loads and deletes the scrap file.
- The external task replies with the message DataLoadAck.

You can see now that the saving task doesn't need to know whether it is sending to the Filer or something else. In its initial DataSave message, it just uses the window/icon handles returned by Wimp_GetPointerInfo as the destination task (in R2/R3) and the Wimp does the rest. It must, if course, always use the pathname returned in the DataSaveAck message when saving its data.

Loading data from a file

This is very straightforward. A load is initiated by the Filer when the user drags a file icon into an application window or icon bar icon.

- The Filer sends the DataLoad message to the application.

- The application loads the named file and replies with a DataLoadAck message.

The receiving task is told the window and icon handles of the destination. From this it can decide whether to open a new window for the file (the file was dragged to the icon bar) or insert it into an existing window.

Loading data from another application

This is simply the case of saving data to another application, but from the point of view of the receiver:

- The external task sends a DataSave message to the application.
- The application replies with a DataSaveAck message, quoting the pathname `<Wimp$Scrap>`.
- The external task saves its data to that file.
- The external task sends the message DataLoad to the application.
- The application loads and deletes the file `<Wimp$Scrap>`.
- The application replies with the message DataLoadAck to the external task.

Again, the receiver can decide what to do with the incoming data from the destination window and icon handles.

The messages used in the above descriptions are described below. Messages 1 and 3 are generally sent as User_Message_Recorded, because they expect a reply, and types 2 and 4 are sent as User_Message, as they don't. The message blocks are designed so that a reply can always use the previously received message's block just by altering a couple of fields.

When receiving any message, allow for either type 17 or 18. ie. don't rely on any sender using one type or the other.

Message_DataSave 1

The data part of the message block is as follows:

R1+20	destination window handle
R1+24	destination icon handle
R1+28	destination x coordinate (screen coordinates, ie not relative
R1+32	destination y coordinate to the window)

R1+36	estimated size of data in bytes
R1+40	file type of data
R1+44	proposed leafname of data, zero-terminated

The first four words come from Wimp_GetPointerInfo. The rest should be filled in by the saving task. In addition to the usual &xxx file types, the following are defined for use within the data transfer protocol:

&1000	directory
&2000	application directory
&ffffffff	un-typed file (ie had load/exec address)

Message_DataSaveAck 2

The message block is as follows:

R1+12	my_ref field of the DataSave message
...	
R1+20	destination window handle
R1+24	destination icon handle
R1+28	destination x coordinate
R1+32	destination y coordinate
R1+36	estimated size of data in bytes; –1 if file is 'unsafe'
R1+40	file type of data
R1+44	full pathname of data (or <Wimp$Scrap>), zero-terminated

The words at +20 to +32 are preserved from the DataSave message. If the receiver of the file (ie the sender of this message) is not the Filer, then it should set the word at +36 to –1. This tells the file's saver that its data is not 'secure', ie is not going to end up in a permanent file. In turn the saver will not mark the file as unmodified, and will not use the returned pathname as the document's window title.

The Filer, on the other hand, will not put –1 in this word, and will insert the file's full pathname at +44. The saver can mark its data as unmodified (since the last save) and use the name as the document window title.

Message_DataLoad 3

From the foregoing descriptions you can see that this message is used in two situations, firstly by the Filer when it wants an application to load a file, and secondly by a task doing a save to indicate that it has written the data to <Wimp$Scrap>. The message block looks like this:

R1+12	my_ref from DataSaveAck message, or 0 if from Filer
...	
R1+20	destination window handle
R1+24	destination icon handle
R1+28	destination x coordinate
R1+32	destination y coordinate
R1+36	estimated size of data in bytes
R1+40	file type
R1+44	full pathname of file, zero terminated

The receiver of this message should check the file type and load it if possible. After a successful load it should reply with a Message_DataLoadAck.

If the sender of this message does not receive an acknowledgement, it should delete `<Wimp$Scrap>` and generate an error of the form `Data transfer failed: Receiver died`.

Message_DataLoadAck 4

R1+12	my_ref from DataLoad message
...	
R1+20	destination window handle
R1+24	destination icon handle
R1+28	destination x coordinate
R1+32	destination y coordinate
R1+36	estimated size of data in bytes
R1+40	file type
R1+44	full pathname of file, zero terminated

Effectively, the file-loading task just changes the message type to 4 and fills in the your_ref field, then sends back the previous DataLoad message to its originator.

Memory data transfer

The foregoing descriptions rely on the use of the Wimp scrap file. However, task to task transfers can be made much quicker by transferring the data within memory. The save and load protocols are modified as below to cope with this.

Saving data to another application (memory)

This is the same as previously described until the DataSave message. Then:

- The external task replies with a RAMFetch message.
- The application sends a RAMTransmit message with data.
- The external task replies with another RAMFetch message.
- The last two steps continue until all the data has been sent and received.

Loading data from another application (memory)

- The external task sends a DataSave message to the application.
- The application replies with a RAMFetch message.
- If this isn't acknowledged with a RAMTransmit, use the `<Wimp$Scrap>` file to perform the operation, otherwise...
- Get and process the data from the RAMTransmit buffer.
- While the RAMTransmit buffer is full:
  Send a RAMFetch for more data
  Get and process the data from the RAMTransmit buffer.

So if the first RAMFetch message is not acknowledged (ie it gets returned as a User_Message_Acknowledge), the data receiver should revert to the file transfer method. If any of the subsequent RAMFetches are unanswered (by RAMTransmits), the transfer should be aborted, but no error will be generated. This is because the sender will have already reported an error to the user.

The data itself is transferred by the sender calling Wimp_TransferBlock (SWI &400F1) just before it sends the RAMTransmit message. See the description of that call for details of entry and exit conditions.

The termination condition for the saver generating RAMTransmits and the loader sending RAMFetches is that the buffer is not full. This implies that if the amount of data sent is an exact multiple of the buffer size, there should be a final pair of messages where the number of bytes sent is 0.

Here are the message blocks for the two messages:

Message_RAMFetch 6

R1+12	my_ref field of DataSave/RAMTransmit message
...	
R1+20	buffer address for Message_RAMTransmit
R1+24	buffer length in bytes

This is sent as a User_Message_Recorded so that a lack of reply to the first one results in the file transfer protocol being used instead, and a lack of reply to subsequent ones allows the transfer to be abandoned. No error should be generated because the other end will have already reported one. A reply to a RAMFetch takes the form of a RAMTransmit from the other task. The receiver should also generate an error if it can't process the received data, eg if it runs out of memory. This should also cause it to stop sending RAMFetch messages.

When allocating its buffer, the receiver can use the estimated data size from the DataSave message, but it should be prepared for more data to actually be sent.

Message_RAMTransmit 7

R1+12	my_ref field of RAMFetch message
...	
R1+20	buffer address from RAMFetch message
R1+24	number of bytes written into the buffer

A data-saving task sends this message in response to a RAMFetch if it can cope with the memory transfer protocol. If the number of bytes transferred into the buffer (using Wimp_TransferBlock) is smaller than the buffer size, then this is the last such message, otherwise there is more to send and the receiver will send another RAMFetch message.

All but the last messages of this type should be sent as User_Message_Recorded types. If there is no acknowledgement, the sender should abort the data transfer and stop sending. It may also give an error message. The last message of this type (which may also be the first if the buffer is big enough) should be sent as a User_Message as there will be no further RAMFetch from the receiver to act as acknowledgement.

Message_DataOpen 5

This message is broadcast by the Filer when the user double-clicks on a file. It gives active applications which recognise the file type a chance to load the file in a new window, instead of having the Filer launch a new copy of the program.

The message block looks like this:

R1+20	window handle of directory display containing file
R1+24	unused
R1+28	x-offset of file icon that was double clicked
R1+32	y-offset of file icon
R1+36	0
R1+40	file type
R1+44	full pathname of file, zero-terminated

The x and y-offsets can be used to display a 'zoom-box' from the original icon to the new window, to give a dynamic impression of the file being opened.

If the user double-clicks on a directory with Shift held down, this message will be broadcast with the file type set to &1000.

The application should respond by loading the file if it can, and acknowledging the message with a Message_LoadDataAck. If no-one loads the file, the Filer will *Run it.

Note that once the resident application has decided to load the file, it should immediately acknowledge the Data Open message. This is so that if the load fails with an error (eg. `Memory full`), the Filer will not then try to *Run the file. This would only result in another error message anyway.

Related SWIs

None

Related vectors

None

Wimp_CreateSubMenu (SWI &400E8)

On entry

R1 = pointer to submenu block
R2 = x coordinate of top left of submenu
R3 = y coordinate of top left of submenu

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is made when a message type MenuWarning (&400C0) is received by an application. This message is sent by the Wimp when a submenu is about to be accessed by the pointer moving over the right-pointing arrow of the parent menu.

The contents of R1 - R3 are obtained from the three words at offsets +20 to +28 of the message block. However, the submenu pointer does not have to be the same as that given in this block (which is just a copy of the one given in the parent menu entry when it was created by Wimp_CreateMenu). For example, the application could create a new window, and use its handle instead.

Related SWIs

None

Related vectors

None

Wimp_SpriteOp (SWI &400E9)

On entry

R0 = reason code (in the range 0 - &FF, see OS_SpriteOp (SWI &2E))
R1 not used
R2 = pointer to sprite name
R3... OS_SpriteOp parameters

On exit

R2... OS_SpriteOp results

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call allows operations on Wimp sprites, without having to specify the Wimp's sprite area pointer. Sprites are always accessed by name (ie &100 is added to the reason code given); pointers to actual sprites are not used. Only read-type operations are allowed, except that you may use the reason code MergeSpriteFile (11) to add further sprites to the Wimp area.

The Wimp first tries to access the sprite in the the RMA part of its sprite pool. If it is not found there, it tries the ROM sprite area. If this fails, it returns the usual `Sprite not found` message.

Related SWIs

None

Related vectors

None

Wimp_BaseOfSprites (SWI &400EA)

On entry	—
On exit	R0 = base of ROM sprite area R1 = base of RMA sprite area
Interrupts	Interrupts are not defined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This can be used to find out the actual addresses of the two areas that make up the Wimp sprite pool, for use with OS_SpriteOp. Note that the the RMA area may move around, eg after a sprite file has been merged with it. In view of this, you should use Wimp_SpriteOp if possible.
Related SWIs	None
Related vectors	None

Wimp_BlockCopy (SWI &400EB)

On entry

R0 = window handle
R1 = source rectangle minimum x coordinate (inclusive)
R2 = source rectangle minimum y coordinate (inclusive)
R3 = source rectangle maximum x coordinate (exclusive)
R4 = source rectangle maximum y coordinate (exclusive)
R5 = destination rectangle minimum x coordinate
R6 = destination rectangle minimum y coordinate

On exit

R0 - R6 = preserved

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

All coordinates are relative to the window's work area origin. The call copies a block of work area space to another position. The Wimp does as much on-screen work as it can, using the VDU block copy primitive, and then invalidates any areas which must be updated by the application itself. The call is useful for performing insert/delete operations in editors.

Note that if any of the source area contains icons, their on-screen images will be copied, but their bounding boxes will not automatically be moved to the destination rectangle. It is up to the application to move the icons explicitly (by deleting and re-creating then) so that they are redrawn correctly.

If the source area contains an ECF pattern, eg representing Wimp colours in a two-colour mode, and the distance between the source and destination is not a multiple of the ECF size (eight pixels vertically and one byte horizontally), then the copied area will be 'out of sync' with the existing pattern.

Note that this call must not be made from inside a Wimp_RedrawWindow or Wimp_UpdateWindow loop.

Related SWIs	None
Related vectors	None

Wimp_SlotSize (SWI &400EC)

On entry

R0 = new size of current slot (–1 to read size)
R1 = new size of next slot (–1 to read size)

On exit

R0 = size of current slot
R1 = size of next slot
R2 = size of free pool

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

Tasks can use this call to read or set the size of the current slot, ie that in which the task is executing, and the next slot (for the next task to start up). It also returns the (possibly altered) size of the Wimp free pool.

If a task wants to alter its memory, it should set R0 to the required amount and R1 to –1. Note that the next slot size does not actually have any effect until the next new task is run. It is simply the amount of the free pool that is allocated to a new task by default. No tasks should set their current slot size – normally, a new task will call *WimpSlot, which then calls Wimp_SlotSize.

On exit from Wimp_SlotSize, the OS_ChangeEnvironment variables MemoryLimit and ApplicationSpaceSize are updated. Note that it is not possible to change the application space size if this is greater than MemoryLimit. This is the situation when, for example, Twin loads at &80000 and runs another task at &8000, setting that task's memory limit to &80000.

Wimp_SlotSize does not check that the currently active object is within the application workspace, or issue Memory service calls, so it should be used with caution. The same applies to *WimpSlot which uses this SWI.

Possible ways in which this call could be used are:

- the run-time library of a language could provide a system call to set the current slot size using Wimp_SlotSize. An example is BASIC's END=&xxxx construct, which allows a program to adjust its HIMEM limit dynamically.
- a program could use Wimp_SlotSize to give itself a private heap above the area used by the host language's memory allocation routines. This only works if the run-time library routines read the MemoryLimit value once, when the program is started. Edit uses this method to allocate memory for its text files.

Related SWIs

None

Related vectors

None

Wimp_ReadPixTrans (SWI &400ED)

On entry

R0 = &0xx if sprite is in the system area
&1xx if sprite is in a user area and R2 points to the name
&2xx if sprite is in a user area and R2 points to the sprite
R1 = 0 if the sprite is in the system area
1 if the sprite is in the Wimp's sprite area
otherwise a pointer to the user sprite area
R2 = a pointer to the sprite name (R0 = &0xx or &1xx) or
a pointer to the sprite (R0 = &2xx)
R6 = a pointer to a four-word block to receive scale factors
R7 = a pointer to a 2, 4 or 16 byte block to receive translation table

On exit

R6 block contains the sprite scale factors
R7 block contains a 2, 4, or 16 byte sprite translation table

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The size of the table pointed to by R7 depends on the sprite's mode. Note that sprites cannot have 256 colours.

The format of the R6 block is:

R6+0	x multiplication factor
R6+4	y multiplication factor
R6+8	x division factor
R6+12	y division factor

All quantities are 32-bits and unsigned.

The format of the R7 block is:

R7+0	colour to store sprite colour 0 as
R7+1	colour to store sprite colour 1 as
...	
R7+14	colour to store sprite colour 14 as
R7+15	colour to store sprite colour 15 as

The purpose of this call is to discover, for a given sprite, how the Wimp would plot it if it was in an icon to give it the most consistent appearance independently of the current Wimp mode. The blocks set up at R6 and R7 on exit can be passed directly to the above mentioned sprite plotting calling.

Scale factors depend on the mode the sprite was defined in and the current Wimp mode. The colour translation table is only valid for sprites defined in 1, 2 or 4-bits per pixel modes. The relationships between the sprite colours and the Wimp colours used to display them are:

Sprite bpp	Colours used
1	Colours 0 - 1 –> Wimp colours 0, 7
2	Colours 0 - 3 –> Wimp colours 0, 2, 4, 7
4	Colours 0 - 15 –> Wimp colours 0 - 15
8	Translation table is undefined

So sprites defined with fewer than four bits per pixel have their pixels mapped into the Wimp's greyscale colours.

Use ColourTrans if you want to plot the sprite using the best approximation to its actual colours. This works for sprites in a 256-colour mode as well.

Related SWIs

None

Related vectors

None

Wimp_ClaimFreeMemory (SWI &400EE)

On entry

R0 = 1 to claim, 0 to release
R1 = amount of memory required

On exit

R1 = amount of memory available (0 if none/already claimed)
R2 = start address of memory (0 if claim failed because not enough)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is analogous to OS_ClaimScreenMemory (SWI &41). It allows a task to claim the whole of the Wimp's free memory pool (the 'Free' entry on the Task Manager display) for its own use. There are restrictions however: the memory can only be accessed in processor supervisor (SVC) mode, and while it is claimed, the Wimp can't use the free pool to dynamically increase the size of the RMA etc. For the second reason, tasks should not hang on to the memory for any longer than absolutely necessary. They should also avoid calling code which is likely to have much to do with memory allocation, eg which claims RMA space. In other words, do not call Wimp_Poll while the free pool is claimed.

Related SWIs

None

Related vectors

None

Wimp_CommandWindow (SWI &400EF)

On entry

R0 = operation type, see below

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call opens a text window in which normal VDU 4-type output can be displayed. It is useful for running old-fashioned, text-based programs from within the Wimp environment. The exact action depends on R0 as follows.

R0 > 1 R0 is treated as a pointer to a text string. This is used as the title for the command window. However, the command window is not opened immediately; it is just marked as 'pending'. It does not become 'active' until the next call to OS_WriteC. When this occurs, the window is opened and the VDU 4 text viewport is set to the same area on the screen.

R0 = 1 The command window status is set to 'active'. However, no drawing on the screen occurs. This is used by the ShellCLI module so that if Wimp_ReportError is called, the error will be printed textually and not in a window.

R0 = 0 The window is closed and removed from the screen. If any output was generated between the window being opened with R0 > 1 and this call being made, the Wimp prompts with `Press SPACE or click mouse to continue` before re-building the screen.

R0 = –1 The command window is closed without any prompting, regardless of whether it was used or not.

The Wimp uses a command window when starting new tasks. It calls Wimp_CommandWindow with R0 pointing to the command string, and then executes the command. If the task was a Wimp one, it will call Wimp_Initialise, at which point the Wimp will close the command window with R0 = –1. Thus the window will never be activated. However, a text-based program will never call Wimp_Initialise, so the command window will be displayed when the program calls OS_WriteC for the first time.

Certain Filer operations which result in commands such as *Copy being executed also use the command window facility in this way.

Wimp_ReportError (SWI &400DF) also interacts with command windows. If the window is active, the error text will simply be displayed textually. However, if the command window is pending, it is marked as 'suspended' and the error is reported in a window as usual.

Related SWIs

None

Related vectors

None

Wimp_TextColour (SWI &400F0)

On entry

R0 = colour

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

R0 on entry has the following form:

Bits	Meaning
0 - 3	Wimp colour (0 - 15)
7	0 for foreground, 1 for background

This call is the text colour equivalent of Wimp_SetColour (SWI &400E6). It is used to set the text foreground or background colour to one of the 16 standard Wimp colours. As text can't be displayed using ECF patterns, only solid colours are used in the monochrome modes.

Wimp_TextColour is used by Wimp_CommandWindow (SWI &400EF) and on exit from the Wimp. It can be called by applications that wish to display VDU 4-type text on the screen in a special window.

Related SWIs

None

Related vectors

None

Wimp_TransferBlock (SWI &400F1)

On entry

R0 = handle of source task
R1 = pointer to source buffer
R2 = handle of destination task
R3 = pointer to destination buffer
R4 = buffer length

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

A block of memory is copied from the source task's address space to the destination task. The buffer addresses and the length are byte aligned, ie the buffers don't have to start on a word boundary or be a whole number of words long.

This call is used in the memory data transfer protocol, described in the section about Wimp_SendMessage (SWI &400E7). The Wimp ensures that the addresses given are valid for the task handles, and generates the error `Wimp transfer out of range` if they are not.

Related SWIs

None

Related vectors

None

Wimp_ReadSysInfo (SWI &400F2)

On entry

R0 = information item index

On exit

R0 = information value

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is used to obtain information from the Wimp which is not readily available otherwise. The value in R0 on entry indicates which item of information is required; its value on exit is the appropriate value. Currently defined values for R0 are:

R0	Meaning
0	number of active tasks

As the call can be used regardless of whether Wimp_Initialise has been called yet, it can be used to see if the program is running from within the desktop environment (R0 > 0 on exit) or simply from a command line (R0=0). Note that even if a program is activated from the Task Manager's command line (F12) facility, R0 will be greater than zero.

Related SWIs

None

Related vectors

None

Wimp_SetFontColours (SWI &400F3)

On entry

R1 = font background colour
R2 = font foreground colour

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the anti-aliased font colours from the two (standard Wimp) colours specified. It calculates how many intermediate colours can be used, and makes the appropriate Font Manager calls. It takes the display mode into account, so that using this call instead of setting the font colours directly saves the application quite a lot of work.

You should not assume the font colours are as you left them across calls to Wimp_Poll, as another task may have called Wimp_SetFontColours before you regain control. Conversely, you don't have to preserve the colours before you change them, as no-one else will be expecting you to.

This call is less powerful than ColourTrans_SetFontColours (SWI &4074F), in that it assumes that Wimp colours 0-7 form a grey-scale sequence.

Related SWIs

None

Related vectors

None

*Configure WimpMode

Sets the default mode for the Desktop

Syntax

```
*Configure WimpMode <number>
```

Parameter

`<number>` the display mode that the Desktop should use after a power-up or hard reset

Use

This configuration parameter is used to set the default mode that the Desktop will use when the machine is first switched on, or after a hard reset. If you leave the Desktop and then re-enter it before powering on again or pressing Ctrl Break, the mode used is the one that was last used by the Desktop.

Example

```
*Configure WimpMode 15
```

Related commands

*Status WimpMode

Related SWIs

Wimp_SetMode

Related vectors

None

*Configure WimpFlags

Sets the drag style and noisiness for the Wimp.

Syntax

```
*Configure WimpFlags <number>
```

Parameter

<number> a value between 0 and 31, as follows:

Bit	Meaning when set
0	window position drags are continuously redrawn
1	window resizing drags are continuously redrawn
2	horizontal scroll drags are continuously redrawn
3	vertical scroll drags are continuously redrawn
4	no beep is generated when an error box appears

The effect of clearing bits 0 - 3 is that the drag operation is performed using an outline, and the window is redrawn at the end of the drag.

Use

This configuration parameter allows the user to control aspects of the Wimp's operation. Generally, all of bits 0 - 3 will be either set or cleared, depending on whether the user requires continuous updates or outline dragging. Bit 4 controls the action of the standard Wimp error reporting window.

Examples

```
*Configure WimpFlags 0
*Configure WimpFlags 15
```

Related commands

*Status WimpFlags

Related SWIs

Wimp_Poll, Wimp_OpenWindow, Wimp_ReportError

*DeskFS

Selects the Desktop filing system

Syntax

```
*DeskFS
```

Use

The desk filing system is a read-only one which contains 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: `Wimp$Path` defaults to `DeskFS:`

Examples

```
*DeskFS
*Copy DeskFS:Templates.Wimp ADFS:Templates.Wimp
```

Related commands

*DeskTop
*Set Wimp$Path

Related SWIs

Wimp_OpenTemplate, Wimp_LoadTemplate, Wimp_CloseTemplate

Related vectors

None

*Desktop

Starts up the Wimp Desktop.

Syntax

```
*Desktop [<command> | -File <pathname>]
```

Parameters

`<command>`	a *Command that will be passed to Wimp_StartTask after the Desktop starts up
`<pathname>`	a text file, each of whose lines will be passed to Wimp_StartTask when the desktop starts up

Use

The Desktop provides an environment in which Wimp programs can operate. When called, it automatically starts resident Wimp task modules such as the filers, the palette utility and the Task Manager. Its optional parameters also allow further tasks to be automatically started.

The Desktop may also be configured as the default language, using the command:

```
*Configure Language 4
```

Examples

```
*Desktop
*Desktop !FormEd
*Desktop -F DTFiles
```

Related commands

*DeskFS, *Desktop_Filer, *Desktop_ADFSFiler et al.

Related SWIs

Wimp_StartTask

Related vectors

None

*Desktop_ADFSFiler
*Desktop_Palette
*Desktop_Filer
*Desktop_NetFiler
*Desktop_RAMFSFiler
*Desktop_TaskManager

Commands to initialise ROM-resident Desktop utilities

Syntax

As above

Parameters

None

Use

These commands are provided for the use of the Desktop, in order to start up the various tasks that appear automatically on the icon bar. Because it is only possible to start a new task using a command lime, these have to be present. However, they are not for users to type, and an attempt to start, say, the palette utility outside of the Desktop will yield the error message `Use *Desktop to start the Palette utility`.

Related commands

*Desktop

Related SWIs

Wimp_StartTask

Related vectors

None

*Filer_CloseDir

Closes a directory display on the Desktop

Syntax

```
*Filer_CloseDir <dirname>
```

Parameter

<dirname> the *full* pathname of a directory whose directory display is to be closed.

Use

This command can be used to close a directory display window and any sub-directories. Ordinarily the display would have be opened by an earlier *Filer_OpenDir command, but it can actually be used to close any open display.

The <dirname> must exactly match a leading sub-string of the title of the directory displays that are to be closed. That is, it must include filing system, drive name and a full path from $. The case of letters is not significant, but the Filer uses lower case for filing system names.

This call must be able to close all directory displays that match the specified sub-string.

You can only use this command from within the desktop environment, or within a Desktop 'startup' or 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 <dirname>
```

Parameter

`<dirname>` the *full* pathname of a directory whose directory display is to be opened.

Use

This command can be used to open a directory display window.

If the display is already open, it simply stays open; no new display appears. For this to work though, `<dirname>` must exactly match the title of the directory display that is already open. That is, it must include filing system, drive name and a full path from $. The case of letters is not significant, but the Filer uses lower case for filing system names.

If the name is even slightly different from an already open one (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. However, if you don't use a proper full pathname, problems can occur when you run applications from within the directory, since they use their pathnames to reference files within themselves.

You can only use this command from within the desktop environment.

Example

```
*Filer_OpenDir adfs::applDisc.$.progs.basic
```

Related commands

*Filer_CloseDir

Related SWIs

None

Related vectors

None

*IconSprites

Merge a sprite file into the Wimp sprite area

Syntax

```
*IconSprites <pathname>
```

Parameters

<pathname> the name of the sprite file to load

Use

This command merges the specified sprite file with those already loaded in the Wimp's shared sprite area. Sprites in this area are used automatically by certain Wimp operations, and because all applications can access them, the need for multiple copies of sprite shapes can be avoided.

Example

```
*IconSprites <Obey$Dir>.!Sprites
```

Related commands

*SSave, *Sload, *Pointer

Related SWIs

Wimp_SpriteOp

Related vectors

None

*Pointer

Displays or hides the mouse pointer

Syntax

```
*Pointer [0|1]
```

Parameters

None	the pointer shape is defined to be the standard arrow shape (held in the Wimp sprite ptr_default) and the sprite colours are programmed to be their default values. The pointer is displayed.
0	the pointer is hidden.
1	this is the same as omitting the parameter.

Use

The *Pointer command is used for ensuring that the pointer is set to its default colours and shape. It can also be used to hide the pointer. Pointer shape number 1 is used by the *Pointer command. Wimp programs that re-program it should use shape 2. Shapes 3 and 4 are used by the Hourglass module.

Examples

```
*Pointer
*Pointer 0
```

Related commands

None

Related SWIs

Wimp_SetPointerShape, Wimp_SpriteOp

Related vectors

None

*Status WimpFlags

Display the WimpFlags configuration parameter.

Syntax

```
*Status WimpFlags
```

Parameters

None

Use

Displays the current WimpFlags configuration; see *Configure WimpFlags.

Example

```
*Status WimpFlags
```

Related commands

*Configure WimpFlags

Related SWIs

None

Related vectors

None

*Status WimpMode

Display the WimpMode configuration parameter.

Syntax

```
*Status WimpMode
```

Parameters

None

Use

Displays the current WimpMode configuration; see *Configure WimpMode.

Example

```
*Status WimpMode
```

Related commands

*Configure WimpMode

Related SWIs

None

Related vectors

None

*WimpPalette

Syntax

```
*WimpPalette <pathname>
```

Parameters

`<pathname>` a file of type &FED (Palette).

Use

The file is used to set the Wimp's colour palette. Typically the file would have been saved using the Desktop's palette utility. An attempt to use a file other than type &FED with this command yields the message `Error in palette file`. The RunType for Palette files is *WimpPalette %0, so a new palette may be set-up from the Desktop simply by double-clicking on the file's icon.

If no task is currently active, the palette is simply stored for later use. Otherwise it is enforced immediately. Palette files can be read in either of two formats:

1. As a list of RGB bytes corresponding to Wimp colours 0 - 15, then the border colour and then the three pointer colours.
2. As a complete VDU sequence, again corresponding to Wimp colours 0 - 15, the border colour and the pointer colours. Typically an entry would be 19,colour,R,G,B.

Type (1) is read for compatibility with Arthur, but since the palette utility always saves files in format (2), this should be used in preference.

Example

```
*WimpPalette greyScale
```

Related commands

None

Related SWIs

Wimp_SetPalette

Related vectors

None

*WimpSlot

Sets the size of application space

Syntax

```
*WimpSlot [-min]<size>[K] [[-max]<size>[K]]
```

Parameters

`-min <size>`	the minimum application workspace size, in bytes or Kilobytes, that the current Wimp application requires. This switch must be given.
`-max <size>`	the maximum amount of application space that the current Wimp application requires. If omitted, there is no maximum. ie. this switch is optional.

Use

This command is typically used by *Obey files called !Run, which the Filer uses to launch a new Wimp application. It calls Wimp_SlotSize to try to set the application memory slot for the current task to be somewhere between the limits specified in the command.

If there are fewer than –min bytes free, the error `Application needs at least <min>K to start up` is generated.

Otherwise, if the current slot is less than the –min value, then the slot size will be set to –min bytes. If it is already between the –min and –max values, then the slot size is unaltered. If –max is supplied, and the current slot size is greater than the –max value, then it will be reduced to –max bytes.

The slot size that is set by this command will also apply to the application that the *Obey file finally invokes.

Examples

```
*WimpSlot 32K
*WimpSlot -min 150K -max 300K
```

Related commands

*WimpTask

Related SWIs

Wimp_SlotSize

Related vectors

None

*WimpTask

Starts up a new task (from within another task)

Syntax

```
*WimpTask <command>
```

Parameter

<command> * Command which is used to start up the new task

Use

The *WimpTask command simply passes the supplied command to the SWI Wimp_StartTask. It can only be called from active Wimp tasks (ie ones that have called Wimp_Initialise).

Example

```
*WimpTask myProg
```

Related commands

*WimpSlot

Related SWIs

Wimp_StartTask

Related vectors

None

Application Notes

In this section we gather together various points which will enable you to write more effective programs running under the RISC OS Wimp. The section is aimed at readers who now have a good understanding of the Wimp calls, but want to ensure that they use them in the most effective ways.

Much of what is said below is to do with consistency and standards. Providing the user with a consistent, reliable interface is the first step towards producing a powerful environment, and one that the user will want to work with instead of just being forced to.

Other topics covered include mode-independence, use of colour, ease of use, the icon bar and dialogue boxes.

General principles

Consistency

The multi-tasking Wimp emphasises that applications work together for the user of the machine:

- They co-operate in sharing the machine.
- They look harmonious.
- Their user interfaces are similar.
- The whole is more important than a single application.

When porting applications into the desktop environment, check that they work well with the existing applications and utilities. Strive to ensure that the habitual user of the desktop environment and the Appplications Suite programs will find your program easy to use, and natural to learn.

Quality

It is much better to write a small program that does something simple, and does it well, than a sprawling mass that crashes occasionally. With a view to this:

- Do not bypass operating system interfaces or access hardware devices directly.
- Do not peek and poke page zero locations (the hardware vectors etc), or kernel workspace.

Such tricks may well not work on future machine and operating system upgrades. Acorn will pursue a policy of continuous improvement and expansion for its product lines: build your software to last.

Responsiveness

RISC OS runs on extremely fast machines, and this speed can be used to make the system easier to use and more productive. The system software has been written very carefully so that all of this performance is delivered to be used by the application, rather than being swallowed up within the operating system. Fast, smooth scrolling and redraw are worth striving for as they make effective and productive use of an application much easier.

Colour

Covering a wide range of screen modes can seem troublesome when constructing an application, but it allows a wide price-range for the end user, who can choose between resolution and cost. Animated bright colour graphics can help make a program easier to understand and to use. Not relying on screen size allows your program to move easily to new better screens and modes when they become available.

Ease of use

This is what WIMP systems are about most of all. All of the various elements described here are ultimately designed to make the computer easier and more pleasant to use, over a wide range of user experience and practice. An application should be:

- easy to learn
- easy to re-learn
- easy to use productively.

These things can conflict with each other, and with other things (eg system cost, program size, program development time, backwards compatibility). Design is not easy, and not all users agree.

Compatibility

The following points should noted, to ensure that your application is compatible with future versions of the Wimp and behaves as well as it can with pre-version 2.00 Wimps.

- Reserved fields must be set to 0, ie reserved words must be 0, and all reserved bits unset.

- Unknown Wimp_Poll reason codes, message actions etc must be ignored – do not generate errors.
- Applications should check Wimp version number, and either adapt themselves if the Wimp is too old, or report an error to the current error handler (using OS_GenerateError).
- Beware of giving errors if window handles are unrecognised as they may belong to another task and it is sometimes legal for their window handles to be returned to you (eg by Wimp_GetPointerInfo).
- Wimp tasks which are modules must obey certain rules (see the section **Relocatable module tasks**).
- Tasks that can receive Key_Pressed events must pass on all unrecognised keys to Wimp_ProcessKey. Failure to do so will result in the 'hot key' facilities not working.

Terminology

Remember that the primary users of RISC OS are users and not programmers: consistent terminology, and the avoidance of jargon, are important in order to make RISC OS friendly. Because RISC OS is not solely a desktop operating system (eg the user has access to the command line interpreter, and non-windowing applications such as BASIC) some jargon inevitably slips through, but this should always be minimised.

Mouse terms

Press	press a button down
Release	release a button
Click	press and release
Drag	press and move the mouse, or press for more than 0.2s
Double click	press,release,press,release within 1s without moving the mouse
Choose	what you do to a menu entry
Type	what you do to keys on the keyboard
Select	Change an object's state by clicking on it

It is a common fault to confuse 'press' and 'click', and to talk about 'selecting' menu entries.

Files

The model of files and filing systems presented within the RISC OS Desktop is that files are always manipulated by their full pathname, including the filing system name, disc title, etc. This gives each file in the system a unique name. There is no concept of 'current directory', so you should not refer to, or

rely on, its being set. Every effort is made to ensure that people never have to type a full pathname, but they do have to see and (more or less) understand them.

Heavy use is made of file types. All files should be typed and date-stamped, rather than being of the older load/exec address form (but be prepared to encounter these, and respond correctly).

Never build absolute drive numbers or absolute file names or absolute filing system names into your program. Check your program working from floppy, Winchester, and Econet, and ensure that installation is easy.

Application Resource files

Applications in the multi-tasking world are typically represented as a directory whose name begins with `!`, eg `!Draw`. When referring to these applications in this document, however, we leave the ! off the name. The Filer provides various mechanisms to help such applications, so that the program and its resources can be treated as a single unit and installation etc is straightforward.

Resources of any form can be held within an application directory. There are several standard types, some of which are discussed in more detail below. Common resource files are:

`!Boot`	`*Run` by the Filer when it first displays the application directory
`!Run`	`*Run` by the Filer when the user double-clicks on the application directory
`!Sprites`	Passed to `*IconSprites` by the `!Boot` file, or the Filer
`!RunImage`	The executable code of a program
`Templates`	The application's window template file
Sprites	The application's private sprite file

If an application is intended for international use then all textual messages within the program should be placed in a separate text file, so that they can be replaced with those of a different language. It may be unhelpful for the application to read such messages one by one, however, as this forces the user of a floppy disc-based system to have the disc containing the application permanently in the drive. Error messages should all be read in when the application starts up, so that producing an error message does not cause a `Please insert disc title` message to appear first.

Note that Obey$Dir and obey files are important here. Applications must always be invoked with their full pathnames, so that Obey$Dir is set correctly. For example, if a resource file is accessed later when the current directory has changed, using a full pathname means it will work OK.

Resources may also be updated by the program during the course of execution. For instance, if an application has user-settable options which should be preserved from one invocation of the program to the next, then saving them within the application directory means that the user does not have to worry about separate files containing such data. As a source of user-settable options this technique is preferable to reading an environment string, since with the latter system the user has to understand how to set up a boot file.

Shared resources

Some resources are of general interest to more than one program. Typical examples include fonts, and modules that provide general facilities. Such resources should be placed in the System application (whose `!Boot` sequence sets a variable `System$Path`) or in a separate application such as Fonts.

It should be noted that the use of shared resources makes applications slightly harder to install, so check carefully that error messages are helpful if the shared resources cannot be located.

The rules above may break down for large applications. Some applications occupy more than one floppy disc, with swapping required during operation. It is difficult to give precise guidelines for such programs, because their requirements vary so widely. The rules above, however, will be used for many smaller programs and so will be reasonably familiar to users. Larger programs should be designed and organised to fit within the same general philosophy, so that users find them easy to install, understand and operate.

The key resource names that are built in to the system are as follows. The application directory is presumed to be called '!Appl' in the examples below.

The !Appl.!Boot file

This is the name of a file which is *Run when the application directory is first 'seen' by the Filer. It is usually an Obey file, ie a list of commands to be passed to the command line interpreter (see the *Obey command for details). Here is a list of commands executed by a typical `!Boot` file, the one for Draw:

```
IconSprites <Obey$Dir>.!Sprites
Set Alias$@RunType_AFF Run <Obey$Dir>.!Run %%*0
Set File$Type_AFF DrawFile
```

The actions of these are to respectively

- load the sprite file for the application into the Wimp sprite area. The Filer does this automatically if there is no `!Boot` file, provided that the sprite file is called !Sprite.
- set the run-type for files created by the application, so that double-clicking on them in the Filer runs Draw on the file. Note the use of `%%` so that the substitution takes place when the file is run, not when the `!Boot` file is obeyed.

  Note that *Set... means that the current value of Obey$Dir is copied into the private variable, ie it is expanded immediately, so if it changes later to Alias$.., the variable will not change.
- set up a name for files created by Draw, for use in Full Info filer displays.

The Filer only runs `!Appl.!Boot` if the sprite called `!appl` does not already exist in the Wimp sprite pool (sprite names are lower case). This prevents repeated delays from re-executing `!Boot` files (or even re-examining application directories). However, it relies on the various applications seen by the Filer having unique names, eg if you have more than one `System` directory then only the first one 'seen' will be used.

The !Appl.!Sprites file

This is the name of a sprite file. For an application `!Appl` this can provide a sprite called `!appl` which is used by the Filer when displaying the directory, and also a sprite named `sm!appl` for use in small-icon displays.

`!Sprites` can also provide sprites that relate to data files controlled by this application, in small and large form (using sprite names `file_ttt` and `small_ttt`, with `ttt` being the hex identity of the file type). See Editor icons below for rules about the appearance of sprites.

Note that these are merged into the Wimp's shared sprite pool using *IconSprites. Private sprites should be contained in !Appl.Sprites, and be loaded into a private sprite area by the application.

The Draw `!Sprites` file contains the following:

`!draw`	The !Draw application directory, Large icons and icon bar
`sm!draw`	The !Draw application directory, Small icons and Full info
`file_aff`	Draw files, Large icons
`small_aff`	Draw files, Small icons and Full info

If your application creates or uses one of the standard file types, you may not have to provide a `file_ttt` icon for it. The following are provided in the Wimp sprite ROM area:

Sprite	Type
`file_bbc`	BBC ROM
`file_feb`	*Obey
`file_fec`	Template
`file_fed`	*Palette
`file_ff6`	Font
`file_ff7`	BBC font
`file_ff8`	*Absolute
`file_ff9`	*Sprite
`file_ffa`	*Module
`file_ffb`	*BASIC
`file_ffc`	Utility
`file_ffd`	*Data
`file_ffe`	*Command
`file_fff`	*Text
`file_dir`	*Non-application directory (folder)
`file_xxx`	*Un-typed (load/exec address) file

Sprites with a * also have small format versions in the Wimp sprite area. Those which haven't can be scaled to half size to achieve the Full info representation. There are also `application` and `small_app` icons for applications which don't have a sprite called !appl.

The standard size for the larger size sprite is 68 OS units square. This is to give them a consistent appearance in Filer windows and on the icon bar. For mode 12 (the usual mode used to define sprites), this size converts to 34

pixels wide by 17 pixels high. The small sprites are half this size: 34 OS units square. Exact sizes are less critical for these ones; typically a mode 12 small sprite will be 16 - 19 pixels wide by 9 high.

Check the appearance of sprites in two, four, sixteen and 256-colour screen modes; the Wimp will do its best to translate from mode 12 colours to those available. Note that icon sprites must not be defined in 256-colour modes (the Wimp can't currently cope with these because of colour translation limitations).

Sprites for document files (`file_ttt`) conventionally have a square black (colour 7) border four OS units wide. (That is, vertical lines are two pixels wide and horizontal ones are one pixel high, if the sprite is defined in mode 12.)

Sprites for devices (eg printers, disc drives) conventionally have a grey (5) outline and cream (12) body. `!appl` sprites are conventionally not square (so require a transparency mask) and are related visually to the `file_ttt` sprites used to represent the files that they edit.

The !Appl.!Run file

This is the name of a file which is *Run when the application directory is double clicked. It is usually an Obey file. Here is what !Draw.!Run does:

```
WimpSlot -min 260K -max 260K
RMEnsure FPEmulator 2.60 RMLoad System:Modules.FPEmulator
RMEnsure FPEmulator 2.60 Error You need FPEmulator 2.60
or later
|
|       also RMEnsure SharedCLibrary and ColourTrans modules
|
Set Draw$Dir <Obey$Dir>
Set Draw$PrintFile printer:
Run "<Draw$Dir>.!RunImage" %*0
```

The action of these commands is to respectively

- call *WimpSlot to ensure that there is enough free memory to start the application.

  Draw, like many applications, knows exactly how much memory it should be loaded with. It acquires more memory once executing (without the knowledge of the language system underneath) by calling SWI

Wimp_SlotSize. Paint, Draw and Edit all maintain shifting heaps above the initial start-up limit, ensuring that extra memory is always given back to the central system when it is not needed.

Applications can also arrange to have the user control dynamically how much memory they should have, by dragging the relevant bar in the Task Manager display. See Message_SetSlot for details.

- ensure that any soft-loaded modules that the application requires are present, using *RMEnsure. If your call to *RMEnsure can load a module from outside your application directory then you should call it twice, to ensure that the newly loaded module is indeed recent enough. If the *RMLoaded module comes from your application directory, one *RMEnsure is sufficient.

- set an environment variable called Draw$Dir from Obey$Dir. (Note that you should not use the variable `Obey$Dir` as another macro could quite likely change the setting of `Obey$Dir`, so it is safer to make a copy.) This allows Draw to access its application directory once the program itself is running, enabling it to access, for example, template files by passing the pathname `<Draw$Dir>.Templates` to Wimp_OpenTemplate. In general you should use the variable `Appl$Path` if the application is called `!Appl`.

- set another environment variable. Different applications will have their own requirements.

- run the executable image file. !RunImage is the conventional name of the actual program. It is also used by the Filer to provide the date-stamp of an application in the Full info display. Note that this time there is only a single % to mark the parameter, as the parameters passed to the *Obey command must be substituted immediately.

Other possible actions that may occur within !Run files are

- execute !Boot. This will usually have been done already, but in the presence of multiple applications with the same name the !Boot file of a different one may have been seen first. This can be done explicitly using a command such as *Run <Obey$Dir>.!Boot, or you could just edit the !Boot file into the !Run file.

- if shared system resources are used then ensure that System$Path is defined, and produce a clean error message if it is not. For example:

  *If "<System$Path>" = "" Then Error 0 System resources cannot be found

- loading a module can take memory from the current slot size, so the *WimpSlot call must be called after loading modules. If you do it both before and after, you avoid loading modules in the case where the application definitely won't fit anyway.

  However, some applications wish to ensure that there is also some free memory after they have loaded, for example if they use the shifting heap strategy outlined above. Such applications may call *WimpSlot again just before executing !RunImage, with a slightly smaller slot setting, to leave just the right amount in the current slot while at the same time ensuring that there is some memory free.

It should be emphasised that the presence of multiple applications with the same name should be thought of as an unusual case, but should not cause anything to crash. Also, complain 'cleanly' if your resources can no longer be found after program startup.

One point to note here is that when an application is starting up from its *Run file, if a screen mode change is to take place, you must call *WimpSlot 0 0 before the change and reset the slot size afterwards.

Memory

In a multi-tasking environment, memory should be used sparingly. Most simple applications require a fixed amount of memory, which can be arranged using a !Run file that specifies exactly the right *WimpSlot size.

Applications with more complex requirements can arrange to call Wimp_SlotSize at run-time to take (and give back) memory. BASIC programs may use the `END=&xxxx` construct to call Wimp_SlotSize.

C programs should call Wimp_SlotSize directly or use 'flex' (available with Release 3 of the Acorn C Compiler), which provides memory allocation for interactive programs requiring large chunks of store.

If Wimp_SlotSize is used directly, the language run-time library (and malloc()) will be entirely unaware that this is happening and so you must organise the extra memory yourself. A common way of doing this is to provide

a shifting heap in which only large blocks of variable size data live. By performing shifting on this memory, pages can be given back to the Wimp when documents are unloaded.

Important:

- Do not reconfigure the machine.
- Do not kill off modules to get more workspace.

Such sequences are quite likely to be hardware-dependent and OS version-dependent.

Graphics

Mode independence

Programs should work in all screen modes in which the Wimp works. Read the current screen mode rather than setting it when your program is loaded, and call OS_ReadVduVariables (SWI &31) to obtain resolution, aspect ratio etc instead of building these into the program.

At the very least, the program should not crash in inappropriate modes, but should display a message in its window, eg 256-colour modes only, as appropriate. Mode 0 is not usually useful, but it is worth making it work if you possibly can. You should make Mode 23 work for users with big monochrome screens. Also, try a square pixel mode (e.g. mode 9). Also check in modes 13, 15, 16 18, 19 and 20.

The Wimp broadcasts a message when the mode changes, so any mode-specific data can be changed at that point.

Think of an OS graphics unit as being a constant unit of measurement, rather than a fraction of the width of the screen. The standard assumption is that there are 180 OS-units to the inch, even though this may in fact vary between physical screens. 'Device-independent' should be interpreted as meaning 'the same size in OS units in any mode' rather than 'the same fraction of the screen'.

Mode 16 is highly non-square, ie the aspect ratio is wrong. Do not try to correct for this automatically, it is an inevitable consequence of trying to fit a great deal of text onto a standard monitor. Some monitors can in any case be adjusted to make the pixels square.

Programs uninterested in colours must also check operation in 256-colour modes, eg some EOR (exclusive OR) tricks do not work quite the same. For instance, see Wimp_SetCaretPosition for a description of how the Wimp draws the caret using EOR plotting. Clock uses a similar trick for the second hand of the clock. As another example, Edit uses EORing with Wimp colour 7 (black) to indicate its selection, but redraws the text in 256-colour modes.

In two-colour modes the Wimp uses ECF patterns for Wimp colours 1 to 6 (grey levels). Note that certain EOR-ing tricks do not work on these, and that use of Wimp_CopyBlock can cause alignment problems for the patterns.

Colours

Use Wimp_SetColour rather than GCOL. You should also consider using the ColourTrans module (see below) if your program deals with absolute colours. Do not change the palette, but read and use the existing one.

Many programs choose to render graphics in 'true' (RGB triplet) colours, and use the current palette to give as close an approximation as possible to the colour they intend. This approach to colour has the enormous benefit that it is not tuned to the limitations of today's hardware.

The palette utility produces a broadcast message when the user changes the palette settings, allowing such programs to repaint for the new palette. A module called ColourTrans (used by Paint and Draw) gives the closest setting possible to a given RGB value. This module is currently provided in RAM in Release 2.0 of RISC OS in !System.Modules.Colours, but may be moved into ROM in later releases of the OS.

Redrawing speed

A technique used on some systems is to remember the bit-map behind a menu or dialogue box when popping it up, to make removing it faster. This is not possible in a multi-tasking environment because a window from a separate task may be changing in the background. Rather than doing this, concentrate on making redraw fast. One available technique for this is to use sprites to remember the contents of a window which are difficult to redraw quickly.

Extensive use of icons within dialogue boxes puts most of the onus of redrawing onto the Wimp, relieving the application writer of that particular burden. It should only be necessary to process redraw events for dialogue boxes when they contain complex user graphics.

Another important technique for speeding up redraw is the use of source-level clipping. During redraw and update, the Wimp always informs the application program of the current clipping rectangle. The redraw of icons, and of objects in the Draw application is fast because this rectangle is used as a rapid test for the rejection of many redraw operations. For an example of how to use this technique, see the Patience program.

Other points

Do not use the system sprite pool (sprite area pointer=0 in window blocks) in production programs, build a user one or use the Wimp area (pointer=1) if appropriate. The system sprite pool is present under RISC OS for backwards compatibility with the Acorn Master, and to help the construction of very simple programs.

Some program developers feel very strongly that a program should be able to take over the entire screen, without any scroll bars etc. It is perfectly possible for a program to do this and still benefit from the multi-tasking environment, as long as this is treated as a specific mode of operation (chosen by a menu entry saying 'Fill screen', for instance), and the program can also operate in a window. This facility can easily be implemented by opening a window the size of the screen, on top of all others. If you set its 'backdrop' bit, then this will also stop any windows from going behind yours. Some programs may even have special properties that only operate when in this mode, eg animation implemented using direct writing to the screen. The desire for this mode of operation, however, should not alone lead you to abandon the multi-tasking world entirely.

The Mouse

The mouse has three buttons, called Select (left), Menu (middle) and Adjust (right).

Select and Adjust

The button actions should follow those of the Filer, Edit and Draw, namely:

- Select is used to make an initial selection
- Adjust is used to toggle elements in and out of this selection and to add extra selections without cancelling the current ones

This is the origin of the names for the buttons.

Always use Select as the 'primary' button of the mouse, used for pointing at things, dragging etc. Adjust is used for less common or less obvious functions, or for slight variations and speedups. If you have no useful separate operation in any particular context, then make Adjust do nothing rather than duplicating the functionality of Select : this is all part of training the user to use Select first.

Another technique for speedups and variations on mouse operations is to look at the setting of the Shift key when the mouse event occurs. Such combinations should never be necessary to the operation of a program, eg a user experimenting with your program should not be expected to try all such combinations.

Double clicks

The Wimp automatically detects double clicks, typically used to mean 'open object'. It should be noted that a double click causes a single click event to be sent to the program first. Some other systems avoid this, which may appear to simplify the task of programming but leads to reduced responsiveness to mouse operations (because the application doesn't get to hear about the first click until the WIMP system is sure it's not a double click). A double click should in any case be thought of as a consolidation of a single click.

The mouse pointer

If you set your own pointer shapes, use pointer shape 2 for them. Set it back to pointer 1 (the standard generic arrow) using *Pointer on a Pointer_Leaving_Window event. Pointers 3 and 4 are used by the hourglass.

Do not use pointer colour 2, as it behaves improperly on high resolution monochrome screens (it alternates between black and transparent).

Menus

Basic operation

Menus are accessed via the middle mouse button, referred to as Menu. As a general rule an application should provide a single menu tree within a window, rather than a collection of little menus that require the user to point at a specific place in the window before pressing Menu. The problem with the latter approach is that it is hard for the user to determine the complete set of commands available.

(This is known as the 'closure' principle. The user should be able to guess what your program can do, and discover fairly rapidly what it can't do, without having to search everywhere for hidden menus etc.)

It is reasonable for entries in the menu to be context-sensitive. In general such portions of the menu depend either on the object indicated by the pointer when Menu is pressed (eg in the Paint file window, Filer directory displays), or on a selected object or objects (eg in Edit, and again Filer directory displays). Where options are not available due to the context of the menu, they should be shaded, rather than omitted.

Menu format

The standard attributes for menus are

- the title says 'Appl' (the application name) rather than 'Appl Menu'
- the title is black (7) on a grey (2) background
- the body is black on a white background (0)
- items are 44 OS-units high
- items have initial letters capitalised and are in lower case otherwise
- there is no separation between items
- text is left-justified (except for keyboard equivalents, see Keystrokes below)
- items use the system font

Try not to make non-leaf menus too wide: reducing the horizontal travel necessary helps rapid choice in nested menus. Open the menu initially 64 OS units to the left of the actual button press, in order to aid this.

Do not make non-leaf items shaded (unselectable). This helps the user to understand the complete set of operations available at any time.

Making menu choices

A button press on a non-leaf should have no effect (other than removing the menu tree), or should duplicate functionality that is available elsewhere.

Pressing Select or Menu on a menu entry should should choose the current menu item, closing the menu tree.

Pressing Adjust on a menu entry should cause the relevant action to be performed, but the menu to remain – see Wimp_CreateMenu for details on how to implement persistent menus.

Icon bar menus

Menus produced from the icon bar (see below) should be moved up a little before opening, so that the relevant menu bar entry is still visible. The base of the menu should be 96 OS units from the bottom of the screen, so it doesn't obscure icon bar sprites. The menu should be horizontally aligned so that the lefthand edge of the menu is 64 OS units to the left of the point where the mouse click occurred.

Dialogue Boxes

Basic operation

The simplest way to provide dialogue boxes is as leaves of the menu tree. If the necessary windows are permanently created and linked to the menu data structure, then the Wimp will handle all opening and closing automatically.

Alternatively, the menu tree can be arranged so that the application is informed (by a message from the Wimp) when the dialogue box is being opened; this allows any computed data to be delayed until the last minute. For a large program with many dialogue boxes this is preferable, as the Wimp has a limit on the number of windows in existence between all tasks.

This form of dialogue box can be visited by the user without clicking on mouse buttons, just like traversing other parts of the menu tree. This is possible because redraw is typically much faster than on previous systems, so popping up the dialogue box and then removing it does not cause a significant delay.

Dialogue box format

The standard attributes for dialogue box windows are:

- title is black (7) on a grey (2) background
- title highlight (input focus) background is also grey (2)
- body is black (7) on a grey (1) or white (0) background
- writeable icon fields are black (7) on white (0), with a border
- action buttons are black (7) on cream (12), with a border

- dialogue boxes do not have Close icons

Dialogue boxes match the colouring of menus, to show that they are part of the menu tree. If the dialogue box is large and has fill-in fields then use colour 1 as the window background rather than 0. Large expanses of white background can make fill-in fields harder to see.

The 'About this program' dialogue box is a useful convention. Provide an 'Info' item at the top of the application's menu, and make the dialogue box its submenu. You should also have the 'Info' item at the top of the menu that you produce when the user clicks with Menu on your icon bar icon. Use Edit's template file to obtain an exact copy of the standard layout used in the Applications Suite programs.

If a menu operation leading to a dialogue box has a keyboard short-cut, Wimp_CreateMenu should be used to initially open the dialogue box, rather than Wimp_OpenWindow (although Wimp_OpenWindow should still be used in response to an Open_Window_Request event). This will ensure that it has the same behaviour concerning cancellation of the operation etc as when accessed through the menu tree.

Dialogue box controls

There are various common forms of icon that occur within dialogue boxes, the most common forms are described here to improve consistency between applications.

Writeable icons

Writeable icons are used for various forms of textual fill-in field. They provide validation strings so that specific characters can be forbidden. Alternatively arbitrary filtering code can be added to the application to ensure that only legal strings (within this particular context) are entered.

Code should be added to the handling of each dialogue box for the following keystrokes. It is not possible for the Wimp to do this, because it doesn't know the right ordering to use.

Down-arrow	move to the next writeable icon within the dialogue box, or to the first if currently at the last.
Up-arrow	move to the previous writeable icon within the dialogue box, or to the last if currently at the first.

Return	move to the next writeable icon within the dialogue box, or perform the default 'go' operation for this dialogue box if currently within the last writeable icon.

When moving to a new writeable icon, place the caret at the end of the existing text of the icon. See Wimp_SetCaretPosition for details of how to do this.

Action icons

This term refers to 'buttons' on which the user clicks on in order to cause some event to occur, typically the event for which the parameters have just been entered in the dialogue box. An example is the OK button in a 'Save as' dialogue box.

The best button type to use is 7 (Menu), with non-zero ESG. This will cause the button to invert while the pointer is over it (like a menu item), and for a button press to be reported.

It is sometimes appropriate to provide keyboard equivalents for action buttons. For instance, if the dialogue box is available via a function key as well as on the menu (see Keystrokes below) then adding key equivalents for action icons may mean that the entire dialogue box can be driven from the keyboard. A conventional use of keys is:

- Return – in the last writeable icon. 'Go' – perform the obvious action initiated by filling in this dialogue box.
- Escape – cancel the operation; remove the dialogue box. Note that Escape is dealt with by the Wimp automatically in this case, as the dialogue box was opened using Wimp_CreateMenu.
- F2, F3 etc to F11 – if the action icons are arranged positionally at the top or bottom of the dialogue box in a simple row, then define F2, F3 etc as positional equivalents of the action buttons, ie F2 activates the left-most one, F3 the next etc. Note that F1 is normally reserved by convention to 'get help', so it should be used to provide help, or do nothing. Similarly, F12 should remain a route to the CLI.

Option icons

This term refers to 'switches', which can either be on or off.

The best icon to use is a text plus sprite one. The text has the validation string `Soptoff,opton`, where the sprites `optoff` and `opton` are defined in the Wimp ROM sprite area. The HVR bits of the icon flags (3, 4 and 9) are set

to 0,1 and 0 respectively (see Wimp_CreateIcon). This generates a box to the left of the text, with a star within it if the option is on (ie the icon is selected). The button type is 11.

The ESG can be zero to make Select and Adjust both toggle the icon state, or non-zero (and unique) to make Select select and Adjust toggle the icon state.

The Filer's menu item `Access` dialogue box for a particular file, uses this type of control (with ESG=0).

Radio icons

This term refers to a set of options where one, and only one, of a set of icons can be selected.

The text plus sprite form is again best, using the validation string `Sradiooff,radioon` from the Wimp sprite area, and a non-zero ESG shared by all the icons in the group, to force exclusive selection. If required, the icons can have their 'adjust' bit set to enable Adjust to toggle the state without deselecting the other icons.

Static dialogue boxes

A static dialogue box is opened using Wimp_OpenWindow rather than Wimp_CreateMenu. A static dialogue box matches normal ones in colours, but has a Close icon.

One common form of use is a 'tool' dialogue box. This typically consists entirely of action icons, and is used in preference to a submenu because typically it is used to switch rapidly between the tool box and the pointer. Examples are the Colour and Tool windows in Paint. These are activated (for a given sprite) using menu entries 'Show colours' and 'Show tools'.

Another example is the Draw 'Shapes' window which appears on the left of the drawing window. Selecting from this dialogue box acts as a short cut to choosing from the Draw menu.

A further reason to use a static dialogue box is that it may be appropriate to attach menus to a dialogue box. Yet another is that it is not possible to drag an icon (eg from a Filer directory display) into a menu tree dialogue box from another application, because as soon as the drag starts the menu tree is cancelled.

The icon bar

The window manager provides an icon bar facility to allow tasks to register icons in a central place. It appears as a thick bar at the bottom of the screen, containing filing system and device icons on the left, and application icons on the right.

When an application is loaded, it registers an icon on the icon bar using Wimp_CreateIcon with window handle =–1 (or –2 for devices). The icon is typically the same as the one used to represent the application directory within the Filer, ie !Appl.

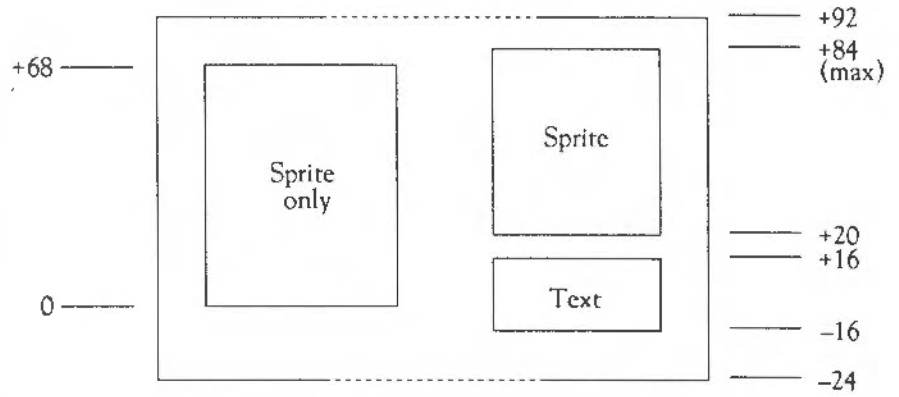
If there are so many icons on the icon bar that it fills up, the Wimp will automatically scroll the bar whenever the mouse pointer is moved close to either end of the bar.

When the mouse is clicked on one of the icons, the Wimp returns the Mouse_Click event (with window handle = –2) to the task which created the icon originally. Similarly, Wimp_GetPointerInfo returns –2 for the window handle when the pointer is over (either part of) the icon bar.

Icon bar dimensions

When Wimp_CreateIcon is called to put an icon on the bar, the Wimp uses the x coordinates of the icon only to determine its width, and then positions the icon as it sees fit. However, for reasons of flexibility, it does not vertically centre the icon, but actually uses both the y coordinates given to determine the icon's position. This means that applications must be aware of the 'standard' dimensions of the bar, in order to position their icons correctly.

There are two main types of icon which are put onto the icon bar: those consisting simply of a sprite, and those consisting of a sprite with text written underneath (see Wimp_CreateIcon for details). The diagram below summarises the rules governing the positioning of such icons, with y coordinates in terms of the icon bar work area origin:



Lower coordinates are inclusive, and upper coordinates are exclusive.

Note that there are two 'baseline' positions: one for sprites with text underneath at –16, and one for those without at 0. The overall effect is better if most of the sprites are of a similar size, since otherwise there is a conflict between wanting the sprites to line up on the baseline, and wanting to centre them vertically.

The general rule is that sprites with text underneath them should always be positioned on the baseline (y=–16), whereas sprites without text can safely be vertically centred (although if they are close to 68 OS units high, it is better to put them on their baseline, y=0).

Keystrokes

Gaining the caret

When you gain the caret, do not automatically re-open your window on top of all others. Also, a window should generally only gain the caret if the user clicks inside it – the exceptions being menus and dialogue boxes, which should give the caret back to the previous owner when they close. This is done automatically by the Wimp for menus and menu dialogue boxes.

A converse to the first point is that when a window is popped to the front of the window stack, do not automatically gain the caret; again, obtaining the caret should only be a result of the user clicking in the work area or on a writeable icon.

Unknown keystrokes

If you receive a keystroke that you do not understand or use, hand it back using Wimp_ProcessKey. This allows other windows to provide hot key operations that work anywhere, it also allows the Wimp to do function key expansion in the last resort.

Shortcuts

Keyboard speedups for menu operations are useful to expert users. Reminders of their existence should be placed right-justified in parentheses in the relevant menu entry. The following are examples of the abbreviations that should be used:

Ctrl-X	control character
F3	function key
Shift-F3	shifted function key
Ctrl-F3	control function key
Ctrl-Shift-F3	control shifted function key
Ctrl-Alt-Shift-F3	control, Alt and Shift plus key

Use function keys for most speedups. Use ctrl-Z, ctrl-X, ctrl-C and and ctrl-V for operations that refer to a 'selected' object or objects. Standard menu text for these would be:

Copy (Ctrl-C)	to caret or pointer
Move (Ctrl-V)	to caret or pointer
Delete (Ctrl-X)	
Clear (Ctrl-Z)	ie de-select

This allows an experienced RISC OS user to access these operations extremely rapidly.

There are also some other keyboard shortcuts, that should be available for expert users. For example:

F1	help
Ctrl-Alt-F12, Alt <nnn>	select keyboard driver for a particular country

Special characters

Use Alt as a shifting key rather than as a function key. Different forms of international keyboards have standardised the use of Alt for entering accented characters.

Do not forbid the use of top-bit-set characters in your program, as many standard accented characters are available in the ASCII range &A0 - &FF. The Wimp clearly distinguishes between these characters and the function keys, which are returned as codes with bit 8 set.

Due to their frequent polling, Wimp programs do not normally need to use escape conditions. The Wimp sets the Escape key to generate an ASCII ESC (&1B) character. If you perform a long calculation without calling Wimp_Poll, you may set the escape action of the machine to generate escape conditions (using *FX 229,0), as long as you set it back again (using *FX 229,1 and then *FX 124) before calling Wimp_Poll.

Editors

An editor is a program that presents files of a particular format as abstract objects which the user can load, edit, save, and print. Text editors, word processors, spreadsheets, draw programs are all editors in this context. Their data files are referred to as documents.

Terminology

Each document being edited is typically displayed in a window. Such windows are referred to as editor windows.

Most editors record, for each document currently being edited, whether the user has made any adjustments yet to the document. This is known as an updated flag.

Some editors are capable of editing several documents of the same type concurrently, while others can edit only one object at a time. Being able to edit several documents is frequently useful, and removes the need for multiple copies of the program to be loaded. Such programs are referred to here as multi-document editors. Edit, Draw and Paint are all multi-document editors, while Maestro and FormEd are not.

The title of an editor window is conventionally the pathname of the current document, centred, with a following * (preceded by a space) if updated. The window's minimum size field can be set to ensure that the title length does not restrict the window's minimum size. Use `<untitled>` if the document has not yet been saved or loaded. If there are multiple views of the same document then append n to this, where n is the number of views of this document that exist.

Window colours

Standard editor window colours are:

- title is black (7) on a grey (2) background
- title highlight (input focus) background is cream (12)
- scroll bar outer colour is dark grey (3)
- scroll bar inner colour is light grey (1)

Editors use RISC OS file types to distinguish which application belongs to which file. Application `!Boot` files should define `Alias$@RunType_ttt` and `File$Type_ttt` variables, and `!appl`, `sm!appl`, `file_ttt` and `small_ttt` sprites (in the Wimp sprite area), as described earlier. File types are allocated by Acorn Customer Support.

The user interface of RISC OS concerning loading and saving documents is rather different from that of other systems, because of the permanent availability of the Filer windows. This means that there is no need for a separate 'mini-Filer' which presents access to the filing system in a cut-down way. Although this may feel unusual at first to experienced users of other systems, it soon becomes natural and helps the feeling that applications are working together within the machine, rather than as separate entities.

Editor icons

Icons that appear on the icon bar should have bounding boxes 68 OS units square. Icons with a different height are strongly discouraged, as they will have their top edges aligned within the Filer Large icon display. A wider icon is permissible, but the size above should be thought of as standard. If the width is greater than 160 OS units then the edges will not be displayed in the Filer Large icon display.

Icons are often displayed half size to save screen space. The Filer will use `sm!appl` and `small_ttt` if these are defined, or scaled versions of `!appl` and `file_ttt` if not.

Starting an editor

The standard ways to start an editor are to:

- double-click on the application icon within the directory display, or
- double-click on a document icon within the directory display

The action taken in the first case is to load a new copy of the application (by running its !Run file). The only visible effect to the user is that the application icon appears on the icon bar. So when you start up with no command line arguments, use Wimp_CreateIcon to put an icon containing your !app sprite onto the icon bar, then enter your polling loop quietly.

In the second case, create the icon bar icon, load the specified document and open a window onto it. This typically occurs by the activation of the run-type of the document file, which in turn will invoke the application by name with the pathname of the document file as its single argument.

For example, the run-type for a Draw file (type &AFF) is:

```
*Run <disc>.!Draw.!Run %*0
```

where `<disc>` is the name of the disc on which the Draw application resides. So when the user double-clicks on a type &AFF file, the Filer executes `*Run pathname`, which in turn executes `<disc>.!Draw.!Run pathname`.

Typically, the !Boot file of the application sets up the run-type for its data files when the application is first seen by the filer. In the case of Draw, the boot file says:

```
*Set Alias$@RunType_AFF
*Run <Obey$Dir>.!Run %%*0
```

See Application Resource Files for details

When a document icon is double-clicked, and a multi-object editor of the appropriate type is already loaded, it is not necessary to reload the application. In this case, the active application will notice the broadcast message from the Filer announcing that a double click has occurred, and will open a window on the document itself. For details, see Message_DataOpen in the section on Wimp_SendMessage (SWI &400E7).

A further way of opening an existing document is to drag its icon from the Filer onto the icon bar icon representing the editor. In this case, a DataLoad message is sent by the Filer to the editor, which can edit the file. This form is important because it specifies the intended editor precisely. For instance if both Paint and FormEd are being used (both can edit sprite files) then double-clicking on a sprite file could load into either.

Newly opened windows on documents should be horizontally centred in a mode 12 screen, and should not occupy the entire screen. This emphasises that the application does not replace the existing desktop world, but is merely added to it. Subsequent windows should not totally obscure ones that this application has already opened. Use a –48 OS unit y offset with each new window.

Creating new documents

In a multi-document editor, clicking on the editor icon on the icon bar creates a new, blank document in a newly opened window. If arguments are required in creating a new document, it may also be appropriate to use a dialogue box during the course of this process. If a style sheet is required (eg for a DTP program) then it may be appropriate to use a static dialogue box, and drag the style sheet from the Filer.

The window created from the loading or creation of a document should be no larger than about 700 OS units wide by 500 high. The first window should be centred horizontally and vertically on the screen. Open subsequent windows 48 OS units lower than the previous one, but if this would overlap the icon bar then return to the original starting position. The initial size and position of windows should be user-configurable, by editing a template file.

In a single-document editor, clicking on the editor icon on the icon bar creates a new, blank document if there is not already a document loaded. If there is already a document loaded, this moves the document window to the front of the window stack, in case it has been obscured by other windows.

The 'About this file' dialogue box provides useful information about a file being edited. Provide it at the top of the menu in a file window, or within a 'Misc' submenu (if there are other miscellaneous menu entries to collect). See the Edit template file for the layout of this dialogue box.

Editing existing documents

To open an existing document, double-click on the document in the Filer. This will cause a broadcast DataOpen message from the Filer, so if your editor can edit multiple documents it can intercept this and load the document into the existing editor.

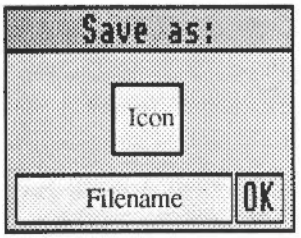
To insert one document into another, drag the icon for the file to be inserted into the open window of the target document. The Filer will then send a message to that window, giving the type and name of the file dragged. The

target (if the file is of a type that can be inserted) can now read the file. If the file is not of a type that can be inserted in this document then the editor should do nothing, ie it should not give an error.

More details of these operations can be found in the section on Wimp_SendMessage (SWI &400E7).

Saving documents

To save a document, provide a dialogue box as follows.



The dialogue box consists of a sprite icon, a writeable icon, and an action icon. The best way to obtain the correct colours and dimensions is to load the template file `DeskFS:Templates.Palette` into FormEd, change the writeable icon text and sprite name appropriately and resave the template into your application's template file.

This is the standard equivalent of the 'mini-Finder' in other systems. If there is no pathname, then invent a simple leaf name, eg `TextFile` in Edit, so that just dragging will not cause a filing system error.

The user can now:

1. press Return or click on OK to save in the already named file (clicking on the menu entry that leads to this dialogue box should have the same effect)
2. edit the pathname as desired using the keyboard
3. drag the icon into a directory display
4. press Escape to cancel the operation.

(1) is used when saving an already saved document. (3) is used when specifying a destination directory for the save. (2) is used when editing the leaf-name of the file. Typically, it is necessary to do (2) and (3) when first saving a file, and (1) thereafter.

When (1) happens, the application should check that the proposed name does at least contain one '.' character. This prevents a common error in beginners, who just see the proposed leaf name, and attempt to select OK immediately. The error message `To save, drag the file icon to a directory display` should be used for this case.

When a drag occurs, send a message to the window (and icon, to cope with the icon bar) where the drag ends, saying that you want to save the file. The window (if it is a directory display) will reply with the full pathname. Save the file using this name. The filename filed for the writeable icon should be up to 255 characters long with a '~<Space' validation string to prevent a space being inserted in the pathname. Longer pathnames should not crash your application.

If the save is successful, update your stored name for the document and remove the save dialogue box using Wimp_CreateMenu (-1): the operation is complete. Check return codes and errors from saves.

Save should be interpreted as being like 'save and resume' from some other systems, ie after the operation the user is still editing the same document. Save should cause the document to be marked as unmodified, unless the save was to a scrap file (see the section on the data transfer protocol in the section on Wimp_SendMessage).

Remember the date-stamp on a loaded document. If the document is saved without being modified, save with an unchanged date-stamp. Otherwise, save with the current date-stamp and update the retained date-stamp for the document.

Menu entries for saving a portion of a document (eg 'Save selection') may be grouped either with other selection operations, or with the 'Save file' operation. If there are several possible selection save formats, putting it on Save may be more appropriate. Balancing submenus may also be an issue. Edit and Paint, for instance, group Save selection with other selection operations while Draw (which has several different forms of Save selection, and many other operations on the Selection submenu) groups it with Save file.

Closing document windows

If the user clicks on the Close icon of a document window, and there is unsaved data, then you should pop up a dialogue box asking:

- Do you want to save your edited file? (if the document has no title)
- Do you want to save edited file 'name'?

You can copy this dialogue box from Edit's template file. If the answer is `Yes` then pop up a Save dialogue box, and if the result is saved then close the document window. If the answer is `No`, or any cancel-menu (eg Escape) occurs, then the operation is abandoned.

If the user clicks Adjust on the Close icon, call Wimp_GetPointerInfo on receipt of the Close_Window_Request. Also, you must open the file's home directory after closing it. This can be obtained by removing the leafname from the end of the file's name and sending a Message_FilerOpenDir broadcast to open the directory.

Quitting editors

A Quit operation should be supplied at the bottom of the menu attached to an editor's icon bar icon. If it is selected when there is unsaved data, then ask the question:

n files edited but not saved in prog: are you sure you want to Quit?

(files should say file if n=1.) The same question should be asked if a preQuit broadcast message is received. See Message_PreQuit for further details.

Miscellaneous points

Null events

Mask out Null_Reason_Code events when you call Wimp_Poll, unless you really need them. Causing the Wimp to return to an application only to have it call Wimp_Poll again immediately slows the system down. If you do need to take null events, Wimp_PollIdle is preferable to Wimp_Poll, unless the user is directly involved (eg when dragging an object) and responsiveness is important.

Dragging

The Wimp's drag operations are specifically for drags that must occur outside all windows. As well as the cycling dashed box form, they allow the use of user-defined graphics, allowing arbitrary objects to be dragged between windows.

If you build drag operations within your window, check that redraw works correctly when things move in the background (the Madness application is useful for testing this). Also, it is important to note that such 'within-window' dragging must use Wimp_UpdateWindow to update the window, rather than drawing directly on the screen.

If the drag works with the mouse button up then menu selection and scrolling can happen during the drag, which is often useful. Stop following the drag on a Pointer_Leaving_Window event, and start again on a Pointer_Entering_Window event.

If the drag works with the button down, then it may continue to work if the pointer is moved out of the window with the button still down. Alternatively for button-down drags, you can restrict the pointer to the visible work area, and automatically scroll the window if the pointer gets close to the edge.

Errors

Always check error returns from Wimp calls. Beware errors in redraw code; they are a common form of infinite loops (because the redraw fails, the Wimp asks you again to redraw, and so on). A suddenly missing font, for instance, should not lead to infinite looping. Check that the failure of Wimp_CreateWindow or Wimp_CreateIcon does not lead you to crash or lose data.

Check cases concerning running out of space.

If the user is asked to insert a floppy disc and selects `Cancel`, you get an error `Disc not present` (&108D5) or `Disc not found` (&108D4) from the ADFS. If you get either of these errors from an operation you need not call Wimp_ReportError, just cancel the operation. This avoids the user getting two error boxes in a row.

Do not have phrases like '`at line 1230`' in error messages from BASIC programs; '`(internal error code 1230)`' is preferable.

Time

There are two clocks that keep track of real time in the system, the hardware clock and a software centi-second timer. The two can diverge by a few seconds a day, but are resynchronised at machine reset. For consistency, always use the centi-second timer.

When using Wimp_PollIdle, remember that monotonic times can go negative (ie wrap round in a 32-bit representation) after around six weeks. So when comparing two times the expression

(newtime – oldtime) > 100

is a better comparison than

newtime > oldtime + 100.

Error messages

&280	Wimp unable to claim work area the RMA area is full
&281	Invalid Wimp operation in this context Some operations are only allowed after a call to Wimp_Initialise
&282	Rectangle area full Screen display is too complex
&283	Too many windows Maximum 64 windows allowed
&284	Window definition won't fit No room in RMA for window
&286	Wimp_GetRectangle called incorrectly
&287	Input focus window not found
&288	Illegal window handle
&289	Bad work area extent Visible window is set to display a non-existent part of the work area
&29F	Bad parameter passed to Wimp in R1 The address in R1 was less than &8000,ie outside of application space

Most of these errors are provided as debugging aids to development programmers, and should not occur when the system is working properly, except for `Too many windows`, which can happen if a task program allows the user to bring up more and more windows. The error is not serious, as long as the task program's error trapping is written properly – when creating a window, you should only update any data structures relating to it once the window has been successfully created.