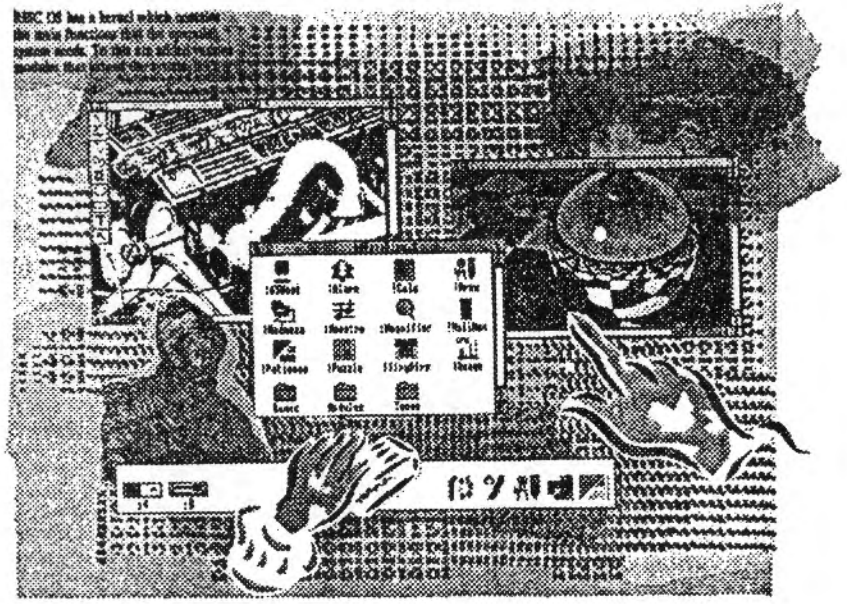
RISC OS
PROGRAMMER'S REFERENCE MANUAL
Volume IV



Acorn

Published by Acorn Computers Technical Publications Department

Published by Acorn Computers Limited
ISBN 1 85250 114 0
Edition 1
Part number 0470,294
Issue 1, October 1991

Contents

44 Writing a filing system

Writing your own filing system

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

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

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

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

What to read next

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

- if you are not using FileCore, then you should read this chapter, which tells you how to add a filing system to FileSwitch
- if you are using FileCore, then you should ignore this chapter and instead read the chapter entitled *Writing a FileCore module* on page 4-63.

In both cases you should also see the chapter entitled *Modules* on page 1-191, for more information on how to write a module.

Filing systems

Declaring a filing system

When your module initialises, it must declare itself to be a filing system, so that FileSwitch knows of its existence. You must call OS_FSControl 12 to do this – see page 3-89 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: see *Appendix* H: *Registering names* on page 6-473.

Service Call handler

Your filing system must have a Service Call handler. It must respond to ServiceFSRedeclare (see page 3-20) by redeclaring the filing system. For some filing systems, it may be appropriate to respond to Service_CloseFile (page 3-21). Disc based filing systems should also support Service_IdentifyDisc (page 3-208), Service_EnumerateFormats (page 3-470), Service_IdentifyFormat (page 3-265), and Service_DisplayFormatHelp (page 3-266).

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 page 3-91.

Other * Commands

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

If the list of * Commands you want to provide closely matches those in the chapter entitled *FileCore* on page 3-187, you ought to investigate adding your filing system to FileCore rather than to FileSwitch; this will be less work for you.

Removing your filing system

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

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

For full details of OS_FSControl 16, see page 3-93.

Image filing systems

Declaring an image filing system

When your module initialises, it must declare itself to be an image filing system, so that FileSwitch knows of its existence. You must call OS_FSControl 35 to do this – see page 3-111 for details. R1 and R2 tell FileSwitch where to find an *image filing system information block*. This in turn tells FileSwitch the locations of all the entry points to the image filing system's low level routines that interface with the hardware.

Image filing system information block

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

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

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

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

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

Image filing system information word

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

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

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

Service Call handler

Your image filing system must have a Service Call handler. It must respond to the same service calls as any other filing system; see the section entitled *Service Call handler* on page 4-4.

* Commands

There are no * Commands that your image filing system **must** provide, but most **should** provide some. See the chapter entitled DOSFS on page 3-305 for an example of what other image filing systems offer.

Removing your image filing system

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

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

For full details of OS_FSControl 36, see page 3-112.

Filing system interfaces

Calling conventions

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

Processor mode

Routines called by FileSwitch are always entered in SVC mode, with both IRQs and FIQs enabled. This means you do not have to change mode 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 (or image filing system) may use this full descending stack. When the filing system (or image filing system) is entered you should take care not to push too much onto the stack, as it is only guaranteed to be 1024 bytes deep; however most of the time it is substantially greater. The stack base is on a 1Mbyte boundary. Hence, to determine how much stack space there is left for your use, use the following code:

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

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

Using file buffers

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

1. an unmodified buffer of the same size
2. an unmodified buffer of a larger size
3. a modified buffer of the same size
4. a modified buffer of a larger size.

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

Workspace

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

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

to load the actual private word into the register.

Supporting unbuffered streams

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

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

Dealing with access

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

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

Other conventions

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

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

You may assume that:

- **all names are null terminated**

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

ImageEntry entry points

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

Interfaces

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

FSEntry_Open and ImageEntry_Open

Open a file

On entry (FSEntry_Open)

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

On exit (FSEntry_Open)

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

On entry (ImageEntry_Open)

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

On exit (ImageEntry_Open)

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

Use

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

General details

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

On exit, your filing system must return a 32-bit file handle that it uses internally to FileSwitch. FileSwitch will then use this file handle for any further calls to your filing system. You may use any value, apart from two handles that have special meanings:

- a handle of 0 means that no file is open
- a handle of –1 is used to indicate 'unset' directory contexts (see FSEntry_Func).

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

Details specific to FSEntry_Open

The reason code given in R0 has the following meaning:

Value	Meaning
0	Open for read
1	Create and open for update
2	Open for update

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

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

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

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

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

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

Details specific to ImageEntry_Open

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

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

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

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

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

Get bytes from a buffered file

On entry

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

On exit

—

Details

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

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

The memory address is not guaranteed to be of any particular alignment. You should if possible optimise your filing system's transfers to word-aligned locations in particular, as FileSwitch's and most clients do tend to be word-aligned. The speed of your transfer routine is vital to filing system performance. A highly optimised example (similar to that used in RISC OS) is given in the section entitled *Example program* on page 4-56.

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

This call is made by FileSwitch for several purposes:

- A client has called OS_BGet at a file offset where FileSwitch has no buffered data, and so FileSwitch needs to read the appropriate block of data into one of its buffers, from where data is returned to the client.
- A client has called OS_GBPB to read a whole number of the buffer size at a file offset that is a multiple of the buffer size. FileSwitch requests that the filing system transfer this data directly to the client's memory. This is often the case where language libraries are being used for file access. If FileSwitch has any buffered data in the transfer range that has been modified but not yet flushed out to the filing system, then this data is copied to the client's memory after the GetBytes call to the filing system.
- A client has called OS_GBPB to perform a more general read. FileSwitch will work out an appropriate set of data transfers. You may be called to fill FileSwitch's buffers as needed and/or to transfer data directly to the client's memory. You should make no assumptions about the exact number and sequence of such calls; as far as possible RISC OS tries to keep the calls in ascending order of file address, to increase efficiency by reducing seek times, and so on.
- A client has called OS_GBPB to perform a more general write. FileSwitch will work out an appropriate set of data transfers. You may be called to fill FileSwitch's buffers as needed, so that the data at the start and/or end of the requested transfer can be put in the right place in FileSwitch's buffers, ready for whole buffer transfer to the filing system as necessary.

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

FSEntry_GetBytes (from an unbuffered file)

Get a byte from an unbuffered file

On entry

R1 = file handle used by your filing system

On exit

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

Details

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

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

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

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

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

Put bytes to a buffered file

On entry

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

On exit

—

Details

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

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

The memory address is not guaranteed to be of any particular alignment. You should if possible optimise your filing system's transfers to word-aligned locations in particular, as FileSwitch's and most clients do tend to be word-aligned. The speed of your transfer routine is vital to filing system performance. A highly optimised example (similar to that used in FileSwitch) is given in the section entitled *Example program* on page 4-56.

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

This call is made by FileSwitch for several purposes:

- A client has called OS_GBPB to write a whole number of the buffer size at a file offset that is a multiple of the buffer size. FileSwitch requests that the filing system transfer this data directly from the client's memory. This is often the case where language libraries are being used for file access. If FileSwitch has any buffered data in the transfer range that has been modified but not yet flushed out to the filing system, then this data is discarded (as it has obviously been invalidated by this operation).
- A client has called OS_BGet/BPut/GBPB at a file offset where FileSwitch has no buffered data, and the current buffer held by FileSwitch has been modified and so must be written to the filing system. (The current FileSwitch implementation does not maintain multiple buffers on each file. It is likely that this will remain the case, as individual filing systems have better knowledge about how to do disc-caching, and intelligent readahead and writebehind for given devices.)
- A client has called OS_GBPB to perform a more general write. FileSwitch will work out an appropriate set of data transfers. You may be called to empty FileSwitch's buffers as needed and/or to transfer data directly from the client's memory. You should make no assumptions about the exact number and sequence of such calls; as far as possible RISC OS tries to keep the calls in ascending order of file address, to increase efficiency by reducing seek times, and so on.

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

FSEntry_PutBytes (to an unbuffered file)

Put a byte to an unbuffered file

On entry

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

On exit

—

Details

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

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

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

FSEntry_Args and ImageEntry_Args

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

FSEntry_Args 0

Read sequential file pointer

On entry

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

On exit

R2 = sequential file pointer

Details

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

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

FSEntry_Args 1

Write sequential file pointer

On entry

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

On exit

—

Details

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

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

- if the file was opened only for reading, or only read permission was granted, then return the error 'Outside file'
- otherwise extend the file with 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 your filing system does not support a pointer as the concept is meaningless (kbd: for example) then it must ignore the call, and **not** return an error.

FSEntry_Args 2

Read file extent

On entry

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

On exit

R2 = file extent

Details

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

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

FSEntry_Args 3 and ImageEntry_Args 3

Write file extent

On entry

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

On exit

—

Details

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

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

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

Buffered files

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

Unbuffered files

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

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

If the new extent is greater than the current one then you must extend the file with 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.

FSEntry_Args 4 and ImageEntry_Args 4

Read size allocated to file

On entry

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

On exit

R2 = size allocated to file by filing system

Details

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

FSEntry_Args 5

EOF check

On entry

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

On exit

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

Details

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

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

FSEntry_Args 6 and ImageEntry_Args 6

Notify of a flush

On entry

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

On exit

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

Details

General details

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

Details specific to FSEntry_Args 5

The modified data should be flushed to its storage media.

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

Details specific to FSEntry_Args 5

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

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

FSEntry_Args 7 and ImageEntry_Args 7

Ensure file size

On entry

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

On exit

R2 = size of file actually ensured

Details

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

FSEntry_Args 8

Write zeros to file

On entry

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

On exit

—

Details

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

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

The 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 in the section entitled *Example program* on page 4-56.

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

FSEntry_Args 9 and ImageEntry_Args 9

Read file datestamp

On entry

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

On exit

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

Details

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

FSEntry_Args 10

Inform of new image stamp

On entry

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

On exit

All registers preserved

Details

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

This call is for information only, and should not require any further action.

FSEntry_Close and ImageEntry_Close

Close an open file

On entry

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

On exit

—

Details

This entry point is called by FileSwitch to request that your filing system close an open file, and put a new date/time stamp on it.

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

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

FSEntry_File and ImageEntry_File

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

FSEntry_File 0 and ImageEntry_File 0

Save file

On entry

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

On exit

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

Details

This entry point is called by FileSwitch to request that your filing system saves data from a buffer held in memory to a file. FileSwitch has already validated the buffer, and ensured that the leafname is not wildcarded. You should return an error such as `File locked` if you could not save the specified file.

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

FSEntry_File 1 and ImageEntry_File 1

Write catalogue information

On entry

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

On exit

—

Details

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

FSEntry_File 2

Write load address

On entry

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

On exit

—

Details

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

FSEntry_File 3

Write execution address

On entry

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

On exit

—

Details

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

FSEntry_File 4

Write attributes

On entry

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

On exit

—

Details

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

FSEntry_File 5 and ImageEntry_File 5

Read catalogue information

On entry

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

On exit

R0 = object type:

0	not found
1	file
2	directory

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

Details

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

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

You should return type 0 if:

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

FSEntry_File 6

Delete object

On entry

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

On exit

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

Details

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

FSEntry_File 7 and ImageEntry_File 7

Create file

On entry

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

On exit

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

Details

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

FSEntry_File 8 and ImageEntry_File 8

Create directory

On entry

R0 = 8
R1 = pointer to directory name
R2 = load address (ignored by RISC OS 2.0)
R3 = execute address (ignored by RISC OS 2.0)
R4 = number of entries (0 for default)
R6 = pointer to special field if present, otherwise 0 (FSEntry_File 8); or image filing system's handle for image that contains file (ImageEntry_File 8)

On exit

—

Details

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

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

If directories don't support load and execute addresses (which will only be of the directory type/datestamp form) then no error should be returned. Note that RISC OS 2.0 will ignore the load and execute addresses in R2 and R3.

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

FSEntry_File 9

Read catalogue information (no length)

On entry

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

On exit

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

Details

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

It is only called by FileSwitch if bit 26 of your 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 10 and ImageEntry_File 10

Read block size

On entry

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

On exit

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

Details

This entry point is called by FileSwitch to read the natural block size for a file.

FSEntry_File 255

Load file

On entry

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

On exit

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

Details

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

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

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

The filename pointed to by R6 on exit should be the non-wildcarded 'leaf' name of the file. That is, if the filename given on entry was `$.1b*`, and the file accessed was the boot file, R6 should point to the filename !Boot.

FSEntry_Func and ImageEntry_Func

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

- amass the information into a dynamic buffer

- print from the buffer and dispose of it.

This avoids problems caused by the 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_ReadVduVariables &100, which copes with most eventualities. Don't cache the value, but read it before each output.

The actions are specified by R0 as given below.

FSEntry_Func 0

Set current directory

On entry

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

On exit

—

Details

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

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

FSEntry_Func 1

Set library directory

On entry

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

On exit

—

Details

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

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

FSEntry_Func 2

Catalogue directory

On entry

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

On exit

—

Details

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

FSEntry_Func 3

Examine directory

On entry

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

On exit

—

Details

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

FSEntry_Func 4

Catalogue library directory

On entry

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

On exit

—

Details

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

FSEntry_Func 5

Examine library directory

On entry

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

On exit

—

Details

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

FSEntry_Func 6

Examine object(s)

On entry

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

On exit

—

Details

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

FSEntry_Func 7

Set filing system options

On entry

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

On exit

—

Details

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

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

(This corresponds to the *Opt command.)

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

FSEntry_Func 8 and ImageEntry_Func 8

Rename object

On entry

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

On exit

R1 = 0 if rename performed (≠0 otherwise)

Details

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

FSEntry_Func 9

Access object(s)

On entry

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

On exit

—

Details

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

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

FSEntry_Func 10

Boot filing system

On entry

R0 = 10

On exit

—

Details

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

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

This call may not return if it runs an application.

FSEntry_Func 11

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

On entry

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

On exit

—

Details

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

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

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

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

FSEntry_Func 12

Read current directory name and privilege byte

On entry

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

On exit

—

Details

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

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

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

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

The buffer pointed to by R2 will not have been validated 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 = pointer to buffer in which to put data

On exit

—

Details

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

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

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

The buffer pointed to by R2 will not have been validated 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 and ImageEntry_Func 14

Read directory entries

On entry

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

On exit

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

Details

This entry point is called by FileSwitch to read the leaf names of entries in a directory into an area of memory pointed to by R2. If the directory name is null, then 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 and ImageEntry_Func 15

Read directory entries and information

On entry

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

On exit

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

Details

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

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

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

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

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

FSEntry_Func 16

Shut down

On entry

R0 = 16

On exit

—

Details

This entry point is called by FileSwitch to request that your filing system go into as dormant a state as possible. For example, it should place 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

—

Details

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

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

FSEntry_Func 18

Set directory contexts

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

Details

This entry point is called by FileSwitch 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
R2 = pointer to buffer in which to put data
R3 = number of object names to read
R4 = offset of first item to read in directory
R5 = length of buffer
R6 = pointer to special field if present, otherwise 0

On exit

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

Details

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

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

Each record is word-aligned.

FSEntry_Func 20

Output full information on object(s)

On entry

R0 = 20
R1 = pointer to wildcarded pathname
R6 = pointer to special field if present, otherwise 0

On exit

—

Details

This entry point is called by FileSwitch to request that your filing system outputs full information on 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.

ImageEntry_Func 21

Notification of new image

On entry

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

On exit

R1 = image filing system's handle for image

Details

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

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

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

ImageEntry_Func 22

Notification that image about to be closed

On entry

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

On exit

—

Details

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

FSEntry_Func 23

Canoncalise special field and disc name

On entry

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

On exit

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

Details

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

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

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

NetFS returns these values:

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

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

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

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

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

FSEntry_Func 24

Resolve wildcard

On entry

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

On exit

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

Details

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

FSEntry_Func 25 and ImageEntry_Func 25

Read defect list

On entry

R0 = 25
R1 = pointer to name of image (FSEntry_Func 25 only)
R2 = pointer to buffer

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

On exit

R0 - R6 preserved

Details

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

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

FSEntry_Func 26 and ImageEntry_Func 26

Add a defect

On entry

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

On exit

R0 - R2, R6 preserved

Details

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

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

FSEntry_Func 27 and ImageEntry_Func 27

Read boot option

On entry

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

On exit

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

Details

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

FSEntry_Func 28 and ImageEntry_Func 28

Write boot option

On entry

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

On exit

R0 - R2, R6 preserved

Details

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

FSEntry_Func 29 and ImageEntry_Func 29

Read used space map

On entry

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

On exit

R0 - R2, R5, R6 preserved

Details

This entry point is called by FileSwitch to read the used space map for the image that holds the object specified by R1 (FSEntry_Func 29), or that is specified by the handle in R6 (ImageEntry_Func 29).

Your filing system should fill the given buffer with 0 bits for unused blocks, and 1 bits for used blocks. The buffer must be filled to its limit, or to the image's limit, whichever is less. The 'perfect' size of the buffer can be calculated from the image's size and its block size. The correspondence of the buffer to the file is 1 bit to 1 block. The least significant bit (bit 0) in a byte comes before the most significant bit.

FSEntry_Func 30 and ImageEntry_Func 30

Read free space

On entry

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

On exit

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

Details

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

FSEntry_Func 31 and ImageEntry_Func 31

Name image

On entry

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

On exit

Registers preserved

Details

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

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

FSEntry_Func 32 and ImageEntry_Func 32

Stamp image

On entry

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

On exit

Registers preserved

Details

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

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

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

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

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

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

FSEntry_Func 33

Get usage of offset

On entry

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

On exit

R2 = kind of object found at offset:

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

Details

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

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

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

rather than:

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

FSEntry_GBPB

Get/put bytes from/to an unbuffered file

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

FSEntry_GBPB 1 and 2

Put multiple bytes to an unbuffered file

On entry

R0 = 1 or 2
R1 = file handle used by your filing system/image filing system
R2 = pointer to buffer
R3 = number of bytes to put to file
If R0 = 1
    R4 = sequential file pointer to use for start of block

On exit

R0, R1 preserved
R2 = address of byte after the last one transferred from buffer
R3 = number of bytes not transferred
R4 = initial file pointer + number of bytes transferred

Details

This entry point is called by FileSwitch to request that your filing system transfer data from memory to the file at either the specified file pointer (R0 = 1), or the current one (R0 = 2). If the specified pointer is beyond the end of the file, then you must fill the file with zeros between the current file extent and the specified pointer before the bytes are transferred.

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

FSEntry_GBPB 3 and 4

Read bytes from an open file

On entry

R0 = 3 or 4
R1 = file handle used by your filing system/image filing system
R2 = pointer to buffer
R3 = number of bytes to get from file
If R0 = 3
    R4 = sequential file pointer to use for start of block

On exit

R0, R1 preserved
R2 = address of byte after the last one transferred to buffer
R3 = number of bytes not transferred
R4 = initial file pointer + number of bytes transferred

Details

This entry point is called by FileSwitch to request that your filing system transfer data from 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, mbshft}     ; 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
        ADD$    mbcnt, mbcnt, #8*4-4*4  ; Four whole words to do ?
        BLT     MovByt240

        STMDB   sp!, {mbsrc3-mbsrc5}; Push some more registers

        LDMIA   mbsrcptr!, {mbsrc3-mbsrc5, mbsrc2}              ; NB. Order!
```

```
        ORR     mbsrc1, mbsrc1, mbsrc3, LSL mbshftL

        MOV     mbsrc3, mbsrc3, LSR mbshftR
        ORR     mbsrc3, mbsrc3, mbsrc4, LSL mbshftL

        MOV     mbsrc4, mbsrc4, LSR mbshftR
        ORR     mbsrc4, mbsrc4, mbsrc5, LSL mbshftL

        MOV     mbsrc5, mbsrc5, LSR mbshftR
        ORR     mbsrc5, mbsrc5, mbsrc2, LSL mbshftL

        STMIA   mbdstptr!, {mbsrc1, mbsrc3-mbsrc5}

        LDMEQDB sp!, {mbsrc3-mbsrc5, mbshftL, mbshftR, pc}^     ; {finished}
        LDMDB   sp!, {mbsrc3-mbsrc5}

        SUB     mbcnt, mbcnt, #4*4
        MOV     mbsrc1, mbsrc2, LSR mbshftR     ; Keep mbsrc1 prepared

MovByt240
        ADDS    mbcnt, mbcnt, #2*4      ; Two whole words to do ?
        BLT     MovByt250

        STMDB   sp!, {mbsrc3}; Push another register

        LDMIA   mbsrcptr!, {mbsrc3, mbsrc2}                     ; NB. Order!
        ORR     mbsrc1, mbsrc1, mbsrc3, LSL mbshftL

        MOV     mbsrc3, mbsrc3, LSR mbshftR
        ORR     mbsrc3, mbsrc3, mbsrc2, LSL mbshftL

        STMIA   mbdstptr!, {mbsrc1, mbsrc3}

        LDMEQDB sp!, {mbsrc3, mbshftL, mbshftR, pc}^            ; {finished}
        LDMDB   sp!, {mbsrc3}

        SUB     mbcnt, mbcnt, #2*4
        MOV     mbsrc1, mbsrc2, LSR mbshftR     ; Keep mbsrc1 prepared

MovByt250
        ADDS    mbcnt, mbcnt, #2*4-1*4; One whole word to do ?
        BLT     MovByt260

        LDR     mbsrc2, [mbsrcptr], #4
        ORR     mbsrc1, mbsrc1, mbsrc2, LSL mbshftL
        STR     mbsrc1, [mbdstptr], #4

        LDMEQDB sp!, {mbshftL, mbshftR, pc}^                    ; {finished}

        SUB     mbcnt, mbcnt, #1*4
        MOV     mbsrc1, mbsrc2, LSR mbshftR     ; Keep mbsrc1 prepared
```



```
MovByt260
        ADDS    mbcnt, mbcnt, #1*4-0*4
        LDMEQDB sp!, {mbshftL, mbshftR, pc}^                    ; {finished}

        LDR     mbsrc2, [mbsrcptr]      ; Store remaining 1, 2 or 3 bytes
        ORR     mbsrc1, mbsrc1, mbsrc2, LSL mbshftL
MovByt270
        STRB    mbsrc1, [mbdstptr], #1
        MOV     mbsrc1, mbsrc1, LSR #8
        SUBS    mbcnt, mbcnt, #1
        BGT     MovByt270

        LDMDB   sp!, {mbshftL, mbshftR, pc}^

;
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

        END
```

45 Writing a FileCore module

Adding your own module to FileCore

FileCore does not know how to communicate directly with the hardware that your filing system uses. Your module must provide these facilities, and declare the entry points to FileCore.

This chapter describes how to add a filing system to FileCore. You should also see the chapter entitled *Modules* on page 1-191 for more information on how to write a module.

Declaring your module

When your module initialises, it must inform FileCore of its existence. You must call FileCore_Create to do this – see page 3-215 for details. R0 tells FileCore where to find a *descriptor block*. This in turn tells FileCore the locations of all the entry points to your module's low level routines that interface with the hardware:

Descriptor block

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

Offset	Contains
0	Bit flags
3	Filing system number (see the chapter entitled *FileSwitch*)
4	Offset of filing system title from module base
8	Offset of boot text from module base
12	Offset of low-level disc op entry from module base
16	Offset of low-level miscellaneous entry from module base

The flag bits in the descriptor block have the following meanings:

Bit	Meaning when set
0	Hard disc needs FIQ
1	Floppy needs FIQ
2	Reserved – must be zero
3	Use scratch space only when a temporary buffer is needed
4	Hard discs support mount like floppies do (ie they fill in sector size, heads, sectors per track and density)

5 Hard discs support poll change
(ie the poll change call works for hard discs and returns a sensible value; also locking them gives a sensible result)

FileCore_Create starts a new instantiation of FileCore, and, on return to your module, R0 points to the workspace that has been reserved for that new instantiation of FileCore. You must store this pointer in your module's workspace for future calls to FileCore; it is this value that tells FileCore which filing system you are (as well as enabling it to find its workspace!).

Unlike filing systems that are added under FileSwitch, the boot text offset cannot be -1 to call a routine.

Temporary buffers

The table below shows areas which may be used for temporary buffers when bit 3 of the flag word is not set:

	Scratch space	Spare screen area	Wimp free pool	RMA heap	System heap	Applic-ation area	Directory cache
FSEntry_Func 8	*	*	*	*	*		
FSEntry_Close	*	*	*	*	*		
FSEntry_Args 7	*	*	*	*	*		
AllocCompact	*	*	*	*	*		
Compact	*	*	*	*	*		
*Backup X X			*	*	*		*
*Backup X Y	*	*	*	*	*		
*Backup X X q			*	*	*	*	*
*Backup X Y q	*	*	*	*	*	*	
*Compact	*	*	*	*	*		

where AllocCompact is the auto-compact triggered when allocating space for a file, and Compact is a normal auto-compact.

Selecting your filing system

Your filing system should provide a * Command to select itself, such as *ADFS or *Net. This must call OS_FSControl 14 to inform FileSwitch that the module has been selected, thus:

```
StarFilingSystemCommand
        STMFD   R13!, {R14}
        MOV     R0, #FSControl_SelectFS
        ADR     R1, FilingSystemName
        SWI     XOS_FSControl
        LDMFD   R13!, {R15}
```


For full details of OS_FSControl 14, see page 3-91.

Other * Commands

There are no other * Commands that your filing system must provide. For many FileCore-based systems the range it provides will be enough, and your module need add no more.

Implementing SWI calls

SWI calls in a FileCore module are usually implemented by simply:

- loading R8 with the pointer to the FileCore instance private word for your module
- calling the corresponding FileCore SWI.

For example, here is how a module might implement a DiscOp SWI:

```
STMFD   R13!, {R8, R14}     ; R12 points to module workspace
LDR     R8, [R12, #offset]  ; R8 <- pointer to FileCore private word
SWI     XFileCore_DiscOp
LDMFD   R13!, {R8, R15}
```

Usually DiscOp, Drives, FreeSpace and DescribeDisc are implemented like this. Of course you can add any extra SWI calls that are necessary.

Removing your filing system

The finalise entry of your module must remove its instantiation of FileCore. For full details of how to do so, see the section entitled *Finalisation Code* on page 1-203.

Module interfaces

The next section describes the interfaces to FileCore that your module must provide.

Module interfaces

Your module must provide two interfaces to FileCore: one for DiscOps, and one for other miscellaneous functions.

DiscOp entry

The entry for DiscOps does much of the work for a DiscOp SWI. It is passed the same values as FileCore_DiscOp (see page 3-210), **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 **always** points to a disc record
- R6 points to a boot block (for hard disc operations only), with the special value &80000000 indicating that none is available.

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

Value	Meaning	Uses	Updates
0	Verify	R2, R4	R2, R4
1	Read sectors	R2, R3, R4	R2, R3, R4
2	Write sectors	R2, R3, R4	R2, R3, R4
3	Floppy disc: read track	R2, R3	
	Hard disc: read Id	R2, R3	
4	Write track	R2, R3	
5	Seek (used only to park)	R2	
6	Restore	R2	
7	Floppy disc: step in		
8	Floppy disc: step out		
9	Read sectors via cache	R2, R3, R4, R6	R2, R3, R4, R6
15	Hard disc: specify	R2	

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

Your routine must preserve R1 - R13 inclusive, except where noted otherwise above, ie:

- R2 must be incremented by the amount transferred for Ops 0, 1, 2 and 9
- R3 must be incremented appropriately for Ops 1, 2 and 9
- R4 must be decremented by the amount transferred for Ops 0, 1, 2 and 9
- R6 must be updated appropriately for Op 9

You must also preserve the N, Z and C flags.

Returning errors

If there is no error then R0 must be zero on exit and the V flag clear. If there is an error then V must be set and R0 must be one of the following:

Value	Meaning
$R0 < \&100$	internal FileCore error number
$\&100 \le R0 < 2^{31}$	pointer to error block
$R0 \ge 2^{31}$	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 – must be zero

Bits 31 and 30 are set by the caller and cleared by your module. Your module must have IRQs disabled from updating the final pair in the list to clearing the active bit.

A negative address of –N indicates that your module has reached the end of the table, and should get the next address/length pair from the start of the scatter list N bytes earlier.

Your module may be called with the scatter pointer (R3) not pointing to the first (address/length) pair. So, to find the addresses of Process error and Process status, you must search for the end of list. From this you may then calculate the start of the scatter block.

MiscOp entry

The entry for MiscOps does much of the work for a MiscOp SWI. It is passed the same values as FileCore_MiscOp (see page 3-227).

- Although FileCore_MiscOp is not available in RISC OS 2, you must still provide this entry point, as other SWIs also use it. (The MiscOp SWI merely provides a convenient way of directly calling this entry point.)

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

Value	Meaning
0	Mount
1	Poll changed
2	Lock drive
3	Unlock drive
4	Poll period

The reason codes you **must** support are 0, 2 and 3; for floppy drives, you **must** also support reason codes 1 and 4.

Your routine must preserve registers, and the N, Z and C flags – except where specifically stated otherwise.

You may only return an error from reason code 0 (Mount). This must be done in the same way as for the DiscOp entry; see the section entitled *Returning errors* on page 4-67.

Under RISC OS 2, the values returned from MiscOp 1 (Poll changed) in bits 4, 5, and 8 - 10 of R3 are ignored by FileCore.

46 Writing a device driver

Adding your own device driver to DeviceFS

FileCore does not know how to communicate directly with the hardware that your filing system uses. Your module must provide these facilities, and declare the entry points to FileCore.

This section describes how to add a filing system to FileCore. You should also see the chapter entitled *Modules* on page 1-191 for more information on how to write a module.

Registering your device driver

When your module initialises, it must register itself and its devices with DeviceFS. You must call DeviceFS_Register (see page 3-408) to register your device driver and any associated devices. Note that modules can hold more than one driver; in such cases you must call DeviceFS_Register for each one.

When you register your device driver with DeviceFS you pass it the location of an entry point to your driver's low level routines that interface with the hardware. A reason code is used to determine which of your driver's routines has been called.

- Reason codes with bit 31 clear are reserved for use by Acorn.
- Reason codes with bit 31 set are reserved for specific drivers. You do not need to register these with Acorn, although we suggest that you maintain some consistency between devices.

Registering and deregistering additional devices

You may later register additional devices by calling DeviceFS_RegisterObject (see page 3-412). This is most commonly needed for devices on a network.

You may deregister devices by calling DeviceFS_DeregisterObject (see page 3-413).

Deregistering your device driver

The finalise entry of your module must deregister all registered drivers and devices by calling DeviceFS_Deregister (see page 3-411). It must make this call for each device driver it registered.

Device driver interfaces

Calling conventions

The principal part of a device driver is the set of low-level routines that control the device's hardware. There are certain conventions that apply to them.

Private word

R8 on entry to the device driver is set to the value of R3 it passed to DeviceFS when registering by calling DeviceFS_Register. Conventionally, this is used as a private word to indicate which hardware platform is being used.

Workspace

R12 on entry to the device driver is set to the value of R4 it passed to DeviceFS when registering by calling DeviceFS_Register. Conventionally, this is used as a pointer to its workspace.

Returning errors

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

Other conventions

Device driver routines must preserve R0, R1, and all other undocumented registers.

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

Interfaces

These are the interfaces that your device driver must provide. The entry point must be declared to DeviceFS by calling DeviceFS_Register when your device driver module is initialised.

DeviceDriver_Entry

Various calls are made by DeviceFS through this entry point when files are being opened and closed, streams halted etc. The actions are specified by R0 as follows:

DeviceDriver_Entry 0

Initialise

On entry

R0 = 0
R2 = DeviceFS stream handle
R3 = flags for opening the stream:
bit 0 clear ⇒ stream opened for RX, set ⇒ stream opened for TX
all others bits reserved, and should be ignored
R6 = pointer to special field control block

On exit

R2 = device driver stream handle

Details

This entry point is called is passed as a stream is being opened onto the device driver by DeviceFS. The stream handle passed in must be stored, as you need to quote it when calling DeviceFS SWIs such as DeviceFS_Threshold, DeviceFS_ReceivedCharacter, and DeviceFS_TransmitCharacter.

The stream handle returned will be passed by DeviceFS when calling the device driver's other entry routines. It must **not** be zero: this is a reserved value, and passing this back will cause some strange effects.

The device driver is also passed a pointer to the special field string: see the section entitled *Special fields* on page 3-402.

You can be assumed that the special field block will remain intact until the stream has been closed.

DeviceDriver_Entry 1

Finalise

On entry

R0 = 1
R2 = device driver stream handle, or 0 for all streams

On exit

—

Details

This entry point is called when a stream is being closed. Your device driver must tidy up and ensure that all vectors have been released. This entry point is also called when a device driver is being removed, although in this case R2 is set to contain 0 indicating that all streams should be closed.

DeviceDriver_Entry 2

Wake up for TX

On entry

R0 = 2
R2 = device driver stream handle

On exit

R0 = 0 if the device driver wishes to remain dormant, else preserved

Details

This entry point is called when data is ready to be transmitted. Your device driver should set R0 to 0 if it wishes to remain dormant, or else start passing data to the physical device, calling DeviceFS_TransmistCharacter to obtain the data to be transmitted.

DeviceDriver_Entry 3

Wake up for RX

On entry

R0 = 3
R2 = device driver stream handle

On exit

—

Details

This entry point is called when data is being requested from the device driver. It is really issued to wake up any dormant device drivers, although you will always receive it when data is going to be read.

The device driver should return any data it receives from the physical device by calling DeviceFS_ReceivedCharacter. This will unblock any task waiting on data being returned.

DeviceDriver_Entry 4

Sleep RX

On entry

R0 = 4
R2 = device driver stream handle

On exit

—

Details

This entry point is called when data is no longer being requested from the device driver. If appropriate, the device driver can then wait to be woken up again using the 'Wake up for RX' entry point.

This call is not applicable to all device drivers; most buffered device drivers would wait for a halt and resume sequence to be triggered on their buffers.

DeviceDriver_Entry 5

EnumDir

On entry

R0 = 5
R2 = pointer to path being enumerated

On exit

—

Details

This entry point is called as a broadcast to all device drivers when the directory structure for DeviceFS is about to be read. This allows them to add and remove non-permanent devices (such as net connections) as required.

The path supplied will be full (eg S.foo.poo) and null terminated.

DeviceDriver_Entry 6 and 7

Create buffer for TX (6), and Create buffer for RX (7)

On entry

R0 = 6 or 7
R2 = device driver stream handle
R3 = suggested flags for buffer being created
R4 = suggested size for buffer
R5 = suggested buffer handle (–1 for unique generated one)
R6 = suggested threshold for buffer

On exit

R3 - R6 modified as the device driver requires

Details

This entry point is called just before the buffer for a stream is going to be created; it allows the device driver to modify the parameters as required.

- R3 contains the buffer flags as specified when the device was registered; see the chapter entitled *The Buffer Manager* on page 5-407.
- R4 contains the suggested buffer size; this should be non-zero.

- R5 contains a suggested buffer handle. This is by default set to –1, which indicates that the buffer manager must attempt to generate a free handle.
  If you specify the handle of an existing buffer, then it will be used and not removed when finished with. For compatibility, the kernel devices use this feature to link up to buffers 1,2 or 3.
- R6 contains the threshold at which a halt event is received. This usually only applies to receive streams which want to halt the receive process, although it can be supplied on either. You may change this value by calling DeviceFS_Threshold.

DeviceDriver_Entry 8

Halt

On entry

R0 = 8
R2 = device driver stream handle

On exit

—

Details

This entry point is called when the free space has dropped below the specified threshold (on creation, or by DeviceFS_Threshold). The device should stop the physical device from transmitting (eg by sending an XOff, or pulling RTS high) until the Resume entry point is called.

DeviceDriver_Entry 9

Resume

On entry

R0 = 9
R2 = device driver stream handle

On exit

—

Details

This entry point is called when the free space has risen above the specified threshold (on creation, or by DeviceFS_Threshold). The device should restart the physical device transmitting (eg by sending an XOn, or pulling RTS low) until the Halt entry point is again called.

DeviceDriver_Entry 10

End of data

On entry

R0 = 10
R2 = device driver stream handle
R3 = -1

On exit

R3 = 0 if more data coming eventually, else -1 (ie no more data coming)

Details

This entry point is called as a result of FileSwitch calling DeviceFS to check on EOF, and DeviceFS believing that there is no more data to come. In more detail:

DeviceFS informs FileSwitch that more data is coming eventually - without calling this entry point - if:

- the stream is buffered, and its buffer still holds data
- the stream is unbuffered, and its RX/TX word is not empty

Otherwise it calls this entry point. In most cases a device driver should ignore this, and return with all registers preserved (so R3 = -1, thus there is no more data coming). In some cases, such as a scanner, you may be able to give an accurate return.

DeviceDriver_Entry 11

Stream created

On entry

R0 = 11
R2 = device driver stream handle
R3 = buffer handle (-1 if none)

On exit

—

Details

This entry point is called after a stream has finally been generated. Your device driver can then perform any important interrupt handling, set itself up and start receiving.

Part 4 – The Window manager

47 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

The Window 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, i.e. 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, e.g. 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 section entitled *Wimp_SendMessage* (SWI &400E7) on page 4-261 and the section entitled *Wimp messages* on page 4-296 for details of these.

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 (e.g. when dragging an object) and responsiveness is important.

All of the event types are described in the section entitled *Wimp_Poll* (SWI &400C7) on page 4-183, along with descriptions of how the application should respond to them.

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. For a complete account of the standards to which you should write a RISC OS application see the RISC OS *Style Guide*.

General principles

For a full description of the general principles you should adopt in writing an application to run under the Wimp see the chapter entitled *General principles* in the RISC OS *Style Guide*.

The following table outlines those sections in the chapter entitled *General Principles*, in the RISC OS *Style Guide*, which describe the basic principles you should follow:

Section	describes:
Ease of use	how to make your application easy to use.
Consistency	how to make applications work together in a uniform way.
Quality	what **not** to do to ensure an application will continue to work with future operating system upgrades.
Different configurations	how to ensure your application works with any reasonable hardware configuration that runs RISC OS.
File handling	the rules for specifying files.
Naming fonts	the syntax to use in naming fonts.
Supporting !Help	what help you should provide in supporting the !Help application, and what you can assume the user knows.

Other important factors that you must consider when writing an application include the following:

Compatibility

The following points should be 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, i.e. 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 (e.g. by Wimp_GetPointerInfo).
- Wimp tasks which are modules must obey certain rules (see the section entitled *Relocatable module tasks* on page 4-131).
- 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.

Responsiveness

RISC OS system software has been written to allow you to write fast, responsive applications. For a description of how best to optimise the responsiveness of your application see the section entitled *Responsiveness* in the *Screen handling* chapter of the RISC OS *Style Guide*.

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.

Terminology

Your application will be easier to understand if your prompts and documentation use the standard RISC OS terminology defined in the chapter entitled *Terminology* in the RISC OS *Style Guide*.

The Mouse

For a description of mouse buttons and operations see the section entitled *Mouse buttons* in the *Terminology* chapter in the RISC OS *Style Guide*.

Select and Adjust

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, for example, 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.

Various parts of the Wimp enforce the interpretations given for the mouse buttons in the Style Guide. For example, icons may be programmed to respond in various ways to clicks with 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

Coordinate system

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'.

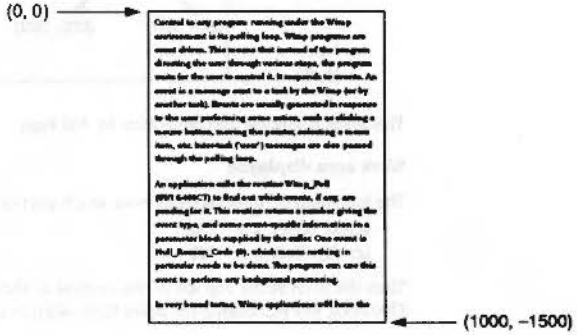
It is common to define the work area such that its origin (0,0) is at the top left of the document. This means that all x scroll offsets are positive (as you can only ever be on or to the right of the work area origin), and all y offsets are zero or negative (as you can only ever be on or below the work 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.

Work area

The following definitions give the total document size:

work_area_x_min = 0
work_area_y_min = -1500
work_area_x_max = 1000
work_area_y_max = 0

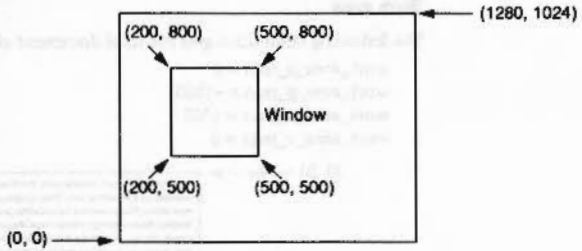


The document is therefore 1000 units wide by 1500 high, with the work area origin at the top left of the document.

Window area

The following definitions give the window's position on the screen and its size:

visible_area_x_min = 200
visible_area_y_min = 500
visible_area_x_max = 500
visible_area_y_max = 800



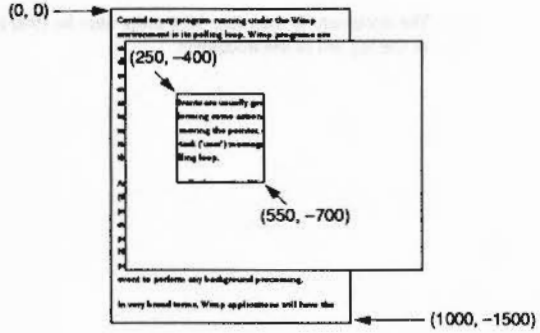
This gives a window 300 units wide by 300 high.

Work area displayed

The following definitions determine which part of the work area is displayed:

scroll_offset_x = 250
scroll_offset_y = –400

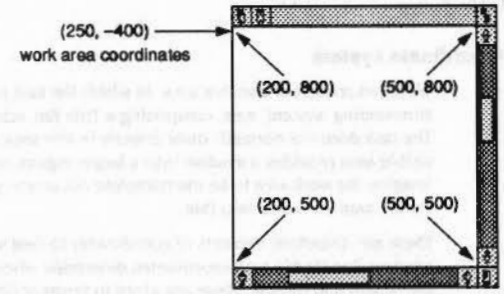
Thus the pixel at the top left of the window is shown on the screen at coordinates (200,800), but represents the point (250,–400) in the work area:



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_max_y = scroll_offset_y = –400
visible_work_area_max_x = scroll_offset_x + width = 550
visible_work_area_min_y = scroll_offset_y – height = –700

Thus on the screen at coordinates (200,500) - (500,800) would be a 300 pixel-square window showing the visible work area (250,–400) - (550,–700):



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 300/1500 = 0.2 of the scroll bar.

Finding the coordinates of a point in the work area of a window

A commonly required calculation is one which gives the coordinates of a point in the work area of a window, given a screen position (for example, where a mouse button click occurred). This mapping obviously depends on the window's screen position and its scroll offsets. The algorithm breaks down into two steps:

1 Find the work area pixel that would be displayed at the screen origin.

The work area pixel displayed at the screen origin can be calculated as follows:

work_area_pixel_at_origin_x = scroll_offset_x – visible_area_min_x
work_area_pixel_at_origin_y = scroll_offset_x – visible_area_max_y

2 Add this to the given screen coordinates.

If the screen position is given by screen_x and screen_y the formula below will return the coordinates of a point in the work area of a window:

work area x = screen_x + work_area_pixel_at_origin_x
work area y = screen_y + work_area_pixel_at_origin_y

Thus the entire formula would be:

work area x = screen_x + (scroll_offset_x – visible_area_min_x)
work area y = screen_y + (scroll_offset_x – visible_area_max_y)

Generally, when this calculation is needed, the scroll offsets and visible work area coordinates are available (e.g. 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).

Window stacks

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'.

Window flags

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.

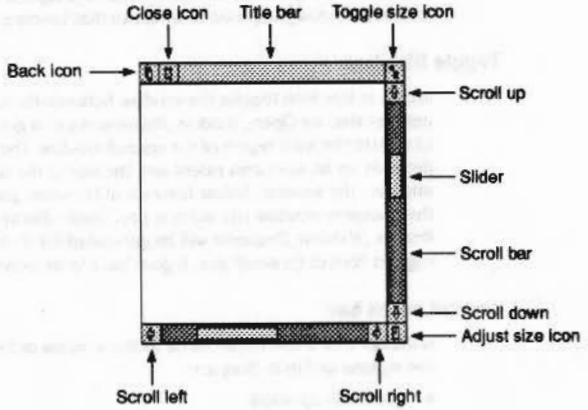
Finally there are miscellaneous properties such as the sprite area address to use for icon sprites, the minimum size of the window, and the icon data for the Title Bar.

Appended to the window definition are any initial icons that it owns. Further icons can be added using the call Wimp_CreateIcon (SWI &400C2) on page 4-166.

Window system areas

For additional details on how windows must behave on the RISC OS desktop see the chapter entitled *Windows* in the RISC OS *Style Guide*.

The window illustrated below has a fully defined system area showing 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 (below the Slider)
- 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', i.e. whereas scroll up increases the y-scroll offset, scroll right increases the x-scroll offset. The five regions within it are:

- the scroll left arrow
- the page left area (left of the Slider)
- the Slider
- the page right area (right of the Slider)
- the scroll right arrow.

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

The Wimp and the application must 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, e.g. 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.

For a description of improving redrawing speed see the section entitled *Redrawing speed* in the *Screen handling* chapter in the RISC OS *Style Guide*.

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, e.g. 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, for example, 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.

Taking over the screen

If you feel that your application must be able to take over the whole screen you can do so by opening a window the size of the screen on top of all other windows. For a description of how best to do this see the section entitled *Taking over the screen* in the *Screen handling* chapter in the RISC OS *Style Guide*.

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, i.e. !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.

Positioning icons on the icon bar

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 on page 4-166 for details).

See the section entitled *Positioning icons on the icon bar* in the *Sprites and icons* chapter in the RISC OS *Style Guide* for a summary of the rules governing the positioning of such icons.

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 writable, 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) on page 4-166.

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 writable, 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.

Update of writeable icons

If an application wishes to update the contents of a writeable icon directly, while the caret is inside the icon, then it cannot in general simply write to the icon's indirected buffer and make sure it gets redrawn.

The general routine goes as follows:

```
REM In: window% = window handle of icon to be updated
REM     icon%   = icon handle of icon to be updated
REM     buffer% = address of indirected icon text buffer
REM     string$ = new string to put into icon

DEF PROCwrite_icon(window%,icon%,buffer%,string$)
LOCAL cw%,ci%,cx%,cy%,ch%,ci%
$buffer% = string$
SYS "Wimp_GetCaretPosition" TO cw%,ci%,cx%,cy%,ch%,ci%
IF cw%=window% AND ci%=icon% THEN
  IF ci% > LEN($buffer%) THEN ci% = LEN($buffer%)
  SYS "Wimp_SetCaretPosition",cw%,ci%,cx%,cy%,-1,ci%
ENDIF
PROCseticonstate(window%,icon%,0,0) :REM redraw the icon
ENDPROC
```

Basically if the length of the string changes, it is possible for the caret to be positioned off the end of the string, in which case nasty effects can occur (especially if you delete the string terminator!).

Deleting and creating icons

Using Wimp_CreateIcon and Wimp_DeleteIcon to create and delete icons has certain disadvantages: the window is not redrawn, and the icon handles can change.

An alternative is to use Wimp_SetIconState to set and clear the icon's 'deleted' bit (bit 23).

However, it should be noted that when calling Wimp_SetIconState to set bit 23 of the icon flags (i.e. to delete it), the icon will not be 'undrawn' unless bit 7 of the icon flags ('needs help to be redrawn') is also set. This is because icons without this bit set are simply redrawn on top of their old selves without filling in the background, to avoid flicker.

Thus to delete an icon, use:

```
block%!0 = window_handle%
block%!4 = icon_handle%
block%!8 = &00800080                    :REM set
block%!12= &00800080                    :REM bits 7 and 23
SYS "Wimp_SetIconState",,block%
```

and to re-create it, use:

```
block%!0 = window_handle%
block%!4 = icon_handle%
block%!8 = &00000000                    :REM clear
block%!12= &00800080                    :REM bits 7 and 23
SYS "Wimp_SetIconState",,block%
```

Note that when re-creating the icon, bit 7 should normally be cleared, to avoid flicker when updating the icon.

Icon sprites

For the rules governing how you must define the appearance and size of sprites, see the chapter entitled *Sprites and icons* in the RISC OS *Style Guide*.

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 (see the section entitled *Use of sprite pools* in the *Sprites and icons* chapter in the RISC OS *Style Guide* for more details). 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 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.

Menus

The Wimp enforces some of the behaviour of menus, the following table outlines those sections in the chapter entitled *Menus and dialogue boxes*, in the RISC OS *Style Guide*, which describe the behaviour of menus under the Wimp:

Section	describes:
Basic menu operation	the different methods of providing menus.
Shading menu items	the rules for shading menu items.
Menu colours	the standard colours you must use for a menu.
Menu size and position	the size and position of menus.
Other points	a list of other rules for formatting a menu. For example; menu titles, splitting items, item ticks.
Making menu choices	the action to perform when a user presses Select, Menu, or Adjust.

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 writable, 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 (i.e. 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, writable 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. See Wimp_CreateMenu for details on how to implement persistent menus.

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

The Wimp enforces some of the behaviour of dialogue boxes, the following table outlines those sections in the chapter entitled *Menus and dialogue boxes*, in the RISC OS *Style Guide*, which describe the behaviour of dialogue boxes under the Wimp:

Section	describes:
Types of dialogue box	the three basic types of dialogue box: ordinary dialogue boxes detached dialogue boxes static dialogue boxes
Dialogue box colours	the standard colours you must use for a dialogue box.
Dialogue boxes and keyboard shortcuts	the rules for consistency.
Wording of dialogue boxes	the how best to construct the wording in a dialogue box.
Default actions	how to ensure the default actions are correct
Standard icons used in dialogue boxes	the various forms of icon: writable icons action icons option icons radio icons arrow icons and sliders
Scrollable lists and pop-up menus	how to use scrollable lists and pop-up menus to present a list of alternative choices within a dialogue box.

Basic operation

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. If the necessary windows are permanently created and linked to the menu data structure, then the Wimp will handle all opening and closing automatically. The Wimp can be made to deal with button clicks within the window, for example automatically highlighting icons.

Also, because writable 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 on page 2-39) 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.

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.

Informational dialogue boxes

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.

Keyboard shortcuts

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.

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.

Icons used in dialogue boxes

There are various 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.

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, i.e. 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 (i.e. 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.

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.

Keyboard input and text handling

The following table outlines those sections in the chapter entitled *Handling input*, in the RISC OS *Style Guide*, which describe how you should implement input under the Wimp:

Section	describes:
Gaining the caret	the conditions under which you may gain the caret.
Unknown keystrokes	what you should do if you receive a keystroke that you do not understand or use - hand it back using Wimp_ProcessKey.
Abbreviations	examples of abbreviations for menu operations useful to expert users.
Selections	the rules to follow when a user selects text (or objects) within your application.
Keyboard shortcuts	consistent shortcuts for common commands, including a table of shortcuts you should provide for particular functions (e.g. Help, Close window, Scroll window, Move etc).
International support	how to make an application more portable in the international market.

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* on page 2-1.

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, e.g. 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 (i.e. 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', for example, 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 may well not be in the window with the grab-hot-keys bit set, and of course the caret position returned by Wimp_Poll will correspond to the window with the carett. 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 reason for a window is to allow its creator to grab hot keys, i.e. 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.

An application should not change or use F12, or any of its shift variants, as it is used by RISC OS.

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. See the section entitled *Keyboard shortcuts* in the *Handling input* chapter in the RISC OS *Style Guide* for details of how you should implement modifiers.

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.

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, i.e. 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 (i.e. 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 2.0 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* on page 6-73.

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, i.e. it is over one of the task's windows, it should re-program the pointer shape, if required.

Mode Independence

For a general description of providing mode independence see the sections entitled *Modes* and *Screen size* in the *Screen handling* chapter in the RISC OS *Style Guide*.

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.

The Wimp broadcasts a message when the mode changes, so any mode-specific data can be changed at that point.

Programs uninterested in colours must also check operation in 256-colour modes, e.g. 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.

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

See the chapter entitled *Colour and sound* in the RISC OS *Style Guide* for:

- a description of different colour models used to define colour;

- the meanings that various colours instinctively convey to users;
- guidelines for which colours to use in your application.

For a general description of colours and the palette see the section entitled *Colours and the palette* in the *Screen handling* chapter in the RISC OS *Style Guide*.

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 table in the following section on colour handling, 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.

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:

standard colour	usage
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, i.e. white logical colour 1 is set to Wimp colour 7, i.e. 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, i.e. white logical colour 1 is set to Wimp colour 2, i.e. light grey logical colour 2 is set to Wimp colour 4, i.e. dark grey logical colour 3 is set to Wimp colour 7, i.e. 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`, i.e. 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.

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 provided in the RISC OS 3 ROM and is available as a RAM loaded module for RISC OS 2.0.

If you want to override the Wimp's translation of colours, you can use 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

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 turn 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 (e.g. a Title Bar to drag its position). However, you might want to allow a non-titled window to be moved, or a window without an 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 top left 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 bottom right 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.

Drag operations within a window

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.

Editors

An editor presents files of a particular format (known as documents) as abstract objects which a user can load, edit, save, and print. Text editors, word processors, spreadsheets and draw programs are all editors in this context.

The following table outlines those sections in the chapter entitled *Editors*, in the RISC OS *Style Guide*, which describe how you should implement editors under the Wimp:

Section	describes:
Editor windows	the title of an editor window and how to position it.
	the colours you should use for the editor window.
Starting an editor	when and how you should start your editor.
Creating a new document	when and how you should create a new document.
Loading a document	when you must load a document.
Inserting one document into another	when you must try to insert a document into the one you are editing.
Saving a document	how to save a document.
Internal RAM based filing system	how to provide an internal RAM based filing system for your editor.
Printing a document	when to print a document.
Closing documents	how and when to close a document.
Quitting editors	how to quit your editor.
Providing information about your editor	why you should include an 'About this program' dialogue box.

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.

File types

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 as described in the section entitled *Filetypes* on page 6-473.

The user interface

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 the section entitled *Application resource files* on page 4-128 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 the section entitled *Message_DataOpen* (5) on page 4-319.

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

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.

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, i.e. it should not give an error.

More details of these operations can be found in the section entitled *Wimp_SendMessage* (SWI &400E7) on page 4-261.

Saving documents

For a description of saving documents see the section entitled *Saving a document* in the *Editors* chapter in the RISC OS *Style Guide*.

To remove the Save dialogue box after saving a file use Wimp_CreateMenu (–1).

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 (e.g. 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

You must supply a `Quit` option at the bottom of an editor's icon bar menu. For a description of how you should implement quitting editors see the section entitled *Quitting editors* in the *Editors* chapter in the RISC OS *Style Guide*.

See *Message_PreQuit* (8) on page -289 for details of what to do if your editor receives a preQuit broadcast message.

Memory management

For a general description of how to use memory efficiently see the section entitled *Use of memory* in the *General Principles* chapter in the RISC OS *Style Guide*.

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, e.g. 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 (i.e. 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 section entitled *Message_SetSlot* (&400C5) on page 4-298 for details.

Applications with 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 or later 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.

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 Resources:$.Resources.*, where * is the name of the module. You can base your own templates on these by loading a ROM file into the template editor (FormEd – available with Release 3 or later 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 AppSPath, where App is the application name. These variables contain a comma-separated list of prefixes, usually directory names, in which the Wimp will search for the directory Templates when opening template files. Their default value points to the ROM, but you could change it to, say, ADFS::MyDisc.<old values> 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.

Application resource files

For a general description of resource files see the section entitled *Application resource files* in the *Application directories* chapter in the RISC OS *Style Guide*.

The following table outlines those sections in the *Application directories* chapter in the RISC OS *Style Guide* which describe the standard resource files available under the Wimp:

Section	describes:
The !Appl.!Boot file	the file which is *Run when the application is first 'seen' by the Filer.
The !Appl.!Sprites file	the sprite file that provides sprites for the Filer to use to represent your application's directory.
The !Appl.!Run file	the file which is *Run when the application directory is double-clicked.
The !Appl.!Messages file	the file used to store all of an application's textual messages.
The !Appl.!Help file	the file used to store plain text that provides brief help about your application.
The !Appl.!Choices file	the file used to store user-settable options so they are preserved from one invocation of the application to the next.
Shared resources	those resources of general interest to more than one program; for example, fonts.
Large applications	how to cope with very large applications.

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 ObeySDir and obey files are important here. Applications must always be invoked with their full pathnames, so that ObeySDir 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.

The !Appl.!Sprites file

For rules about the size and appearance of sprites you use to represent an application see the section entitled *Appearance of sprites* in the chapter entitled *Sprites and icons* in the RISC OS *Style Guide*.

This file must be of type 'Sprite'.

The !Appl.!Run file

For a general description of the `!Appl.!Run` file see the section entitled *The !Appl.!Run file* in the *Application directories* chapter in the RISC OS *Style Guide*.

Example

Here is an example !Run file:

```
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 the section entitled *Message_SetSlot* (&400C5) on page 4-298 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 DrawSDir from ObeySDir. (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 start-up.

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.

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 be able to terminate when asked to, e.g. 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* on page 1-191 for details. You may also like to read the sections on *Wimp_Initialise* (SWI &400C0) on page -157 and *Wimp_CloseDown* (SWI &400DD) on page -241 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.

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.

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, i.e. outside of application space

Most of the above 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.

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 (i.e. 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.
```

Wimp behaviour under RISC OS 3

As the Wimp is developed, it is often necessary to make alterations or additions to the application interface. Sometimes this can be done in such a way that the new behaviour is 'back-compatible' with the old (i.e. it will not confuse applications which do not know about the extension), for example, where a reserved field can be set non-zero to enable the new feature.

However, it is occasionally necessary to make changes that could potentially confuse an application which was not aware of them. In order to cope with this, the Wimp allows an application to inform it of how much it knows, by supplying the version number of the latest release of the Wimp which the programmers have taken into account:

```
SWI Wimp_Initialise

R0 = latest known version of Wimp * 100
R1 = "TASK"
R2 = pointer to the task name string
```

Applications written for RISC OS 2.0 should all have R0 set to 200 when calling Wimp_Initialise.

This allows the Wimp to provide 'incompatible' new facilities only to those applications which it knows are aware of them, thereby avoiding compatibility problems with the others.

In many cases a 'compatible' extension can be made, where it is clear to the Wimp whether or not the application is trying to use the new facility, so not all extensions require the application to 'know' about the later version of the Wimp.

Under RISC OS 3 an application can only pass 200 or 300 to Wimp_Initialise. The wimp will give an error if any other value is passed in.

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* on page 1-191.

Service Calls

Service_Memory (Service Call &11)

Memory controller about to be remapped

On entry

R0 = amount application space will change by
R1 = Service_Memory
R2 = current active object pointer (CAO)

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_Reset (Service Call &27)

Post-Reset

On entry

R1 = &27 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This is issued at the end of a machine reset. It must never be claimed.

Since MessageTrans does not close message files on a soft reset, applications that do not wish their message files to be open once they leave the desktop should call MessageTrans_CloseFile for all their open files at this point. However, it is perfectly legal for message files to be left open over soft reset.

Service_StartWimp (Service Call &49)

Start up any resident module tasks using Wimp_StartTask

On entry

R1 = Service_StartWimp

On exit

R0 = pointer to * Command to start module
R1 = 0 to claim call

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_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 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_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 (e.g. if the module is *RMKilled, or the Filer task is quitted from the TaskManager window) the Filer module tries to ensure that all the other filing system tasks are also closed down, by issuing this service call.

On receipt of this service call, a filing system task should check to see if it is active and if it is, it should close itself down by calling Wimp_CloseDown as follows:

```
serviceCode
...
        TEQ     R1, #Service_FilerDying
        BNE     try next
        STMFD   SP!, {R0-R1, R14}
        LDR     R0, taskHandle          ;in workspace
        CMP     R0, #0
        MOVNE   R14, #0
        STRNE   R14, taskHandle
        LDRGT   R1, taskid
        SWIGT   XWimp_CloseDown
        LDMFD   SP!, {R0-R1, PC}^       ;can't return errors from service call

trynext
...
taskid  DCB     "TASK"                  ;word-aligned
```

Service_MouseTrap (Service Call &52)

The Wimp has detected a significant mouse movement

On entry

R0 = mouse x coordinate
R1 = &52 (reason code)
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

All registers preserved

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, e.g. ten times a second. Some Wimp events are dependent 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, e.g. 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 (e.g. 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_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)
R1 = &53 (reason code)
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_WimpReportError (Service Call &57)

Request to suspend trapping of VDU output so an error can be displayed

On entry

R0 = 0 (window closing) or
R0 = 1 (window opening)
R1 = &57 (reason code)

On exit

All registers preserved

Use

This service is provided so that certain tasks which usually trap VDU output (e.g. 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_WimpSaveDesktop (Service Call &5C)

Save some state to the file

On entry

R0 = flag word (as in Message_SaveDesktop)
R1 = &5C (reason code)
R2 = file handle of file to write *commands to

On exit

R0 = pointer to Error, if necessary, else preserved
R1 = 0 for error (i.e. claim), else preserved
All other registers preserved

Use

This call is provided for modules which need to save some state to the file, e.g. ColourTrans saves its calibration.

When a module receives this service code it should write out any *commands, to the specified file handle, which should be performed by a Desktop Boot file on entry to the Desktop.

If an error occurs (`Disc full`, `Can't extend`, or even a module specific error like '`Can't save desktop now because...`' then the service should be claimed, and R0 should point to the error block.

This service call is performed before the task manager issues the Wimp broadcast message Message_SaveDesktop.

Service_WimpPalette (Service Call &5D)

Palette change

On entry

R1 = Service_WimpPalette

On exit

All register preserved

Use

This call is issued by the Window Manager when SWI Wimp_SetPalette is called to set the WIMP's palette. It can be used to tell when the palette has changed.

This service call should not be claimed.

Service_DesktopWelcome (Service Call &7C)

Desktop starting

On entry

R1 = &7C (reason code)

On exit

R1 = 0 to claim and stop startup screen from appearing.

Use

This service call is issued just before the RISC OS 3 startup screen is drawn. It should be claimed if you want to replace the startup screen, or to prevent it from appearing.

Service_ShutDown (Service Call &7E)

Switcher shutting down

On entry

R1 = &7E (reason code)

On exit

R1 = 0 to claim and stop shutdown.

Use

This service call is issued by the Task manager when it is asked to perform a shutdown, it should be claimed to stop the shutdown from happening.

For example this is used by RAMFS to warn the user that there are unsaved files in the RAM disc.

Service_ShutdownComplete (Service Call &80)

Shutdown completed

On entry

R1 = &80 (reason code)

On exit

This service call should not be claimed.

Use

This service call is issued when the machine has been brought to the state where it can be safely turned off and the shutdown dbox is on the screen.

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, e.g. 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	except in the icon bar
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, i.e. 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 (at least 300 for RISC OS 3 applications)
R1 = 'TASK' (low byte = 'T', high byte = 'K', i.e. &4B534154)
R2 = pointer to short description of task, for use in Task Manager display
R3 = pointer to a list of message numbers terminated by a 0 word (not if R0 is less than 300)

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 (i.e. a task that starts from and returns to the Desktop 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.

The task will only receive messages which are included in the list pointed to by R3. The list should not (and cannot) include Message_Quit (0) as this message will always be delivered to all tasks.

The messages list is not required if the value passed in R0 is 200.

Note that an application may still get a message that is not in the list if it is run under an older Wimp, you should not give an error in this case.

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.

The Window extent, in versions of RISC OS later than 2.0, is automatically rounded to be a whole number of pixels (and is re-rounded on a mode change).

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, i.e. 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, i.e. 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, i.e. it is on top of a tool window
6	window can be opened (or dragged) outside the screen area (see also *Configure WimpFlags)
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	forces window to stay on screen (not in RISC OS 2.0)
14	ignore right-hand extent if the size box of the window is dragged (not in RISC OS 2.0)
15	ignore lower extent if the size box of the window is dragged (not in RISC OS 2.0)

Flags marked * are old-style control icon flags. You should use bits 24 to 31 in preference.

The five bits below are set by the Wimp and may be read using Wimp_GetWindowState (SWI &400CB).

Bit	Meaning when set
16	window is open
17	window is fully visible, i.e. 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	force window to screen once on the next Open_Window

If any of the following circumstances occur, the Wimp sets bit 21 of the window flags, which causes the window to be restricted to the screen area for one call to Wimp_OpenWindow only (this causes the bit to be cleared):

- a toggle-to-full-size occurs
- while you are dragging the size box
- immediately after a mode change
- on the next call to Wimp_OpenWindow after Wimp_SetExtent is called for a window which is fully on-screen at the time.

When a window is first opened it will be forced onto the screen, but can subsequently be dragged off by the user.

If you are dragging the size box of a window, and you move the pointer off the bottom-right of the screen, the Wimp will try to make the window bigger. If it succeeds, the window will be forced onto the screen, so it will appear to grow upwards and left. The speed of growing can be controlled by how far the pointer is off-screen.

Window flags bit 21 is also set automatically by the Wimp when a menu or a dialogue window is opened as a result of the pointer moving over the relevant submenu icon, or as a result of a call to Wimp_CreateMenu or Wimp_CreateSubMenu. This forces the menus onto the screen normally, but allows them to be dragged off-screen if desired.

This bit is not supported in RISC OS 2.0.

Bit	Meaning when set
22 - 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.

Bit	Meaning when set
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 Size icon 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, i.e. 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, i.e. bit 2 is ignored
- the title background is filled, i.e. bit 5 is ignored
- the Wimp redraws the title, i.e. 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) on page -166 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: –1 for right of icon bar –2 left of icon bar The following are not available in RISC_OS 2.0 –3 ⇒ create icon on iconbar to left of icon handle R0 –4 ⇒ create icon on iconbar to right of icon handle R0 –5 ⇒ create icon on left side, scanning from the left –6 ⇒ create icon on left side, scanning from the right –7 ⇒ create icon on right side, scanning from the left –8 ⇒ create icon on right side, scanning from the 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

This call does not affect the screen, except when creating an icon on the icon bar. Use Wimp_ForceRedraw to do this.

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 following sections:

  Wimp_SetCaretPosition (SWI &400D2) on page -219

  Wimp_GetCaretPosition (SWI &400D3) on page -221

  Wimp_Poll *Key_Pressed* 8 event on page 4-190.

The window handle at R1+0 may be an application window, or:

–1 for the right half of the icon bar (applications)
–2 for the left half of the icon bar (devices)

Note that creating an icon on the icon bar may cause other icons to 'shuffle', changing their x coordinates.

The following features are not available in RISC_OS 2.0:

The window handle at R1+0 can also be:

–3 to create an icon on the iconbar to the left of icon handle R0, or
–4 to create an icon on the iconbar to the right of icon handle R0

where R0 = handle of icon to open next to, if |R1+0| = –3 or –4
= –1 ⇒ create icon at the extreme left (–3) or right (–4)

This allows icons to be recreated and deleted (in order to change their width, for example) such that they stay in the same relative position on the iconbar. It also allows applications to keep groups of iconbar icons together.

Iconbar icons can also be prioritised, so that, for example, the RAM disc icon can be positioned immediately to the right of the Apps icon. Instead of using window handle values –1, –2, –3 or –4 you are advised to prioritise iconbar icons using the following values:

–5 ⇒ create icon on left side, scanning from the left
–6 ⇒ create icon on left side, scanning from the right
–7 ⇒ create icon on right side, scanning from the left
–8 ⇒ create icon on right side, scanning from the right
where R0 = signed 32-bit priority (higher priority ⇒ towards outside)

The Wimp positions the icons so that they are sorted, with those of higher priority nearer the extreme ends of the iconbar. Where icons are of equal priority, the position of the new icon is determined by the scan direction.

The priorities assumed for the other possible window handle values are:

window handle values	priority
handle = –1	0
handle = –2	0
handle = –3, R0 = icon handle	same as matched icon
handle = –3, R0 = –1	&78000000
handle = –4, R0 = icon handle	same as matched icon
handle = –4, R0 = –1	&78000000

The various Desktop modules create icons with the following priorities:

module	priority
Task Manager	&60000000
!Help	&40000000
Palette Utility	&20000000
Applications 0	
ADFS hard discs	&70000000
ADFS floppy discs	&60000000
'Apps' icon	&50000000
RAM disc	&40000000
Ethernet	&30000000
Econet	&20000000
Other filing systems	&10000000
Printer drivers	&0F000000
TinyDir	–&40000000

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 area 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.

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 - ; ~ \

B command

The (B)order command sets the border type for the Icon. The border will only be drawn if the border bit for the Icon is also set. This command will override the Wimp's default border for the Icon.

`B Type Slab_in_colour`

Type	0	3d raised border
	1	Group border
	2	Default action border
	3	Writable icon border
	4	Pressed 3d border
	5	Normal 2d border
	6	Pressed type 2 border

Border type 0 changes to 4 and 2 changes to 6 when a mouse event is about to be reported to the application, and changes back when you release the mouse button, or when you move the pointer away from the icon. It will not change if the icon's button type means that the click is not reported to the application.

Slab_in_colour	is used to set the icon's background colour just before a mouse click is reported to the task.

The (B)order command is not available in RISC OS 2.0

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.

K command

The (K)eys command is used to assign specific functionalities to various keys. You should follow the K with any or all of R, A, T, D, or N:

Option	Action
R	If the icon is not the last icon in the window, pressing Return in the icon will move the caret to the beginning of the next writable icon in the window. If the icon is the last writable icon in the window then Return (code 13) will be passed to the application.
A	Pressing the up or down arrow keys will move the caret to the previous or next writable icon in the window, retaining the same position in the string. Pressing the up arrow key in the first writable icon in a window will move the caret to the last writable icon. Pressing the down arrow key in the last icon will move the caret to the first icon.
T	Pressing Tab in the icon will move the caret to the beginning of the next writable icon in the window. Pressing Shift-Tab will move the caret to the beginning of the previous writable icon in the window. The caret wraps around from last to first in the same way as in the A option.
D	Pressing any of Copy, Delete, Shift-Copy, Ctrl-U, or Ctrl-Copy will notify the application with the appropriate key codes as well as doing its defined action as specified in the section entitled *Key_Pressed* 8 on page 4-190.
N	The application will be notified about all key presses in the icon, even if they are handled by the Wimp.

Options can be combined by including more than one option letter after the K command. For example:

KA	will give the arrow keys functionality
KAR	will give the arrow keys and the Return functionalities

The (K)eys command is not available in RISC OS 2.0

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.

P command

The (P)ointer Shape command changes the pointer shape while over the icon.

```
P spritename active_x active_y
```

The sprites should be in the Wimp sprite area.

The (P)ointer command is not available in RISC OS 2.0

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 +12 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

R0 corrupted

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.

Errors

If a window is deleted while being dragged, an error is reported by the Wimp, except in the case of a menu, where pressing Escape causes the drag to terminate and the menu tree to be deleted.

This error is not returned under RISC OS 2.0.

Related SWIs

None

Related vectors

None

Wimp_DeleteIcon (SWI &400C4)

On entry

R1 = pointer to block

On exit

R0 corrupted

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 (–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 unless the window handle is –2 (i.e. the iconbar). 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

R0 corrupted

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 -3 means the window behind the Wimp's backwindow, hiding it from sight (-3 not available in RISC OS 2.0)

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

R0 corrupted

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)
R3 = pointer to poll word if R0 bit 22 set (not in RISC OS 2.0)

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.

Errors

If any error occurs inside Wimp_Poll (apart from an error in the parameters to the call), it is reported by the Wimp itself, and is not passed back to any of the applications.

If an Escape Condition is pending when Wimp_Poll is called, or if Escape Conditions are enabled, the Wimp will report an error, and will cancel the escape condition and disable escape condition generation.

These errors are not returned under RISC OS 2.0.

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	Poll word non-zero
14 - 16	reserved
17	User_Message
18	User_Message_Recorded
19	User_Message_Acknowledge

The highest priority are types 17 - 19, however, any event sent using Wimp_SendMessage has the same priority as a type 17, 18 or 19. In particular, this means that types 11 and 12 are higher in priority than type 1 (because the Wimp sends them using Wimp_SendMessage).

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	do not return PollWord_NonZero (not in RISC OS 2.0)
14 - 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 - 21	must be 0
22	R3 on entry is pointer to poll word (not in RISC OS 2.0)
23	scan poll word at high priority (not in RISC OS 2.0)
24	save or restore floating point registers (not in RISC OS 2.0)
25 - 31	must be 0

Note that the bits above which are marked 'queue for later handling' stop the Wimp from proceeding, i.e. it stops all other tasks too.

Saving floating point registers

If R0 bit 24 is set (not available in RISC OS 2.0) the FP registers will be preserved over calls to Wimp_Poll.

The floating point registers should only be saved if one or more of the following is true:

- The task is controlling arbitrary applications 'underneath' it, which may use floating point instructions. An example of such a controlling task is the TaskWindow module.
- The task requires to set up a floating point status register value that is different from that used by the C run-time system (which happens to be &70000).

This is because in general other C programs running under the Wimp that use floating point will not save their FP registers, but will assume that the status register is still correct for the C run-time environment.

To enable this to work, the Wimp resets the FP status register to the correct value for the C run-time environment immediately after saving the FP registers for a task that requests it.

There is one complication with this: when the Wimp comes to save the floating point registers for a task, it is possible (when using the actual FP hardware, as opposed to the emulator) for an asynchronous exception to be generated (for example, after a divide by 0, the next FP instruction is the one that actually generates the error).

In this case OS_GenerateError is called by the FP support code, once it has determined the cause of the exception. The important point here is that the error is passed to the task whose FP registers were being saved. When OS_GenerateError is called, the supervisor stack is cleared out, so it is as though the Wimp_Poll call never happened. Note that the error number here has the top bit set, which indicates to the error handler that execution cannot be resumed after the PC address where the error occurred.

Reason codes

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, e.g. 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 button s = 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 *Icon button types* on page -170 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, <-I	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 (i.e. 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.

PollWord_NonZero 13

This facility is not available under RISC OS 2.0.

If R0 bit 23 was set, the poll word will be scanned before the messages or the Redraw_Window_Requests are delivered. Note that this means that the screen may not yet be up-to-date, and certain messages may not have been delivered (in particular Message_ModeChange).

If the Wimp discovers that the word has become non-zero, it will return the following event from Wimp_Poll:

R0 = 13 (PollWord_NonZero)
|R1+0| = address of poll word
|R1+4| = contents of poll word

This facility is used to transfer control to a task's foreground process, where control is currently in an interrupt routine, service call handler or the like.

For example, the NetFiler module intercepts a special service call which is issued by NetFS whenever a *Logon, *Bye or *SDisc is executed. This tells NetFiler that it should re-scan its list of fileservers and update the iconbar as appropriate, but it cannot do this directly because it needs to get control in the foreground in order to call the Wimp.

It therefore sets a flag in its workspace, which tells it that it should rescan the list the next time the Wimp returns to it from Wimp_Poll. Using the new facility, it can use a 'fast poll' to get the Wimp to tell it **before** the screen is up-to-date, which means that if the user issues a *Logon from within ShellCLI, the NetFiler can update the iconbar before the screen is redrawn when ShellCLI returns, and so the iconbar does not have to be redrawn twice.

A more normal application for this would be for a background process to buffer incoming data in the RMA, and to signal to its foreground process when there was enough data to use. It would normally use the 'slow' form of polling, so that it could update its window with the new data.

Note that there is no guarantee about how long it will take before the application regains control, since other applications can take control away from the Wimp for arbitrarily long periods of time (e.g. ShellCLI).

Events 14 - 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 (i.e. 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, i.e. 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, i.e. 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 entitled *Redrawing windows* on page 4-97. 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

R0 corrupted

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 (or –2 to indicate the icon bar)
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)

A window handle value of –2 is not available in RISC OS 2.0.

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 (in RISC OS 2.0), else in later versions of RISC OS:
bit 0 set ⇒ just return window header (without icons)
bit 1 reserved (must be 0)
bits 2 - 31 pointer to buffer to receive data

On exit

R0 corrupted

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 (or –2 to indicate the icon bar)

A window handle value of –2 is not available in RISC OS 2.0.

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

R0 corrupted
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:

Value 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

R0 corrupted

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

R0 corrupted

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	Right-hand 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

In versions of RISC OS later than 2.0 shaded icons in menus are treated differently from normal shaded icons, in that the latter are treated as being 'invisible' to the Wimp, i.e. Wimp_GetPointerInfo will never return them. In menus, however, the icons are not invisible, but are not allowed to be selected. This allows the interactive help program to see the icons and to ask for help on them.

If the mouse is over a greyed out icon an icon handle of -1 will be returned, unless it is in a menu, where the icon handle is returned.

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

R0 corrupted

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 (or –2 to indicate the icon bar) - 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)

A window handle value of –2 is not available in RISC OS 2.0.

The coordinates are passed as screen coordinates, i.e. 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 (i.e. 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 and 12. 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
12	drag horizontal and vertical scroll bars (not in RISC OS 2.0)

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 unmovable). 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 (i.e. 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 (i.e. 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 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.)

When this call is made the pointer leaves the current window, when the drag ends a pointer entering window event will be generated.

Type 12

This is similar to types 1 - 4. It is equivalent to an Adjust drag on one of the scroll bars.

This type is not available in RISC OS 2.0.

Related SWIs

None

Related vectors

None

Wimp_ForceRedraw (SWI &400D1)

On entry

R0 = window handle (–1 means whole screen, –2 indicates the icon bar)
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

R0 corrupted

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 window handle value of –2 on entry is not available in RISC OS 2.0.

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

R0 corrupted

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

R0 corrupted

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
	4	allow submenu to be opened even if this item is greyed out (not in RSIC OS 2.0)
	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.

If a menu title starts with '\', then it and all submenus opened off it are reversed, so that:

- ticks appear on the right, arrows on the left;
- submenus are opened to the left (including Message_MenuWarning);
- left-justified menu items are right-justified, and vice-versa.

The above only applies in versions of RISC OS later than 2.0.

Pressing Return while the caret is inside a writeable item is equivalent to pressing a mouse button, i.e. 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); i.e. 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, e.g. 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 section entitled *Menus* on page 4-105 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

R0 corrupted
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, e.g. `Display.Small icons` for a typical Filer menu selection.

Related SWIs

None

Related vectors

None

Wimp_WhichIcon (SWI &400D6)

On entry

R0 = window handle (or –2 to indicate the icon bar)
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

R0 corrupted
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

A window handle value of –2 on entry is not available in RISC OS 2.0.

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

R0 corrupted

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

The Window extent, in versions of RISC OS later than 2.0, is automatically rounded to be a whole number of pixels (and is re-rounded on a mode change).

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, e.g. 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

R0 corrupted

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 (e.g. 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 (i.e. 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

R0 corrupted

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

R0 corrupted

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
(or 0 to find the size of the template - not available in RISC OS 2.0)
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

R0 corrupted
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

You must call Wimp_OpenTemplate before calling this SWI.

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.

Errors

No errors are generated if the template could not be found. To check for this condition check for R6 = 0 on exit.

If an error occurs you are still expected to close the template file.

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 (e.g. 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 (e.g. 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.

Finding out the size of the template

This is not available in RISC OS 2.0.

If R1 on entry ≤ 0 ⇒ return R1 = size of buffer required
R2 = size of indirected data

On exit
|R5...| = actual name only if |R5...| on entry was wildcarded

No error is generated for objects of type ≠ 1: the object is simply loaded into the buffer, and no indirected data processing occurs. This is different from RISC OS 2.0, which reported an error in these circumstances.

Related SWIs

None

Related vectors

None

Wimp_ProcessKey (SWI &400DC)

On entry

R0 = character code

On exit

R0 corrupted

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 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

R0 corrupted

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 (i.e. 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, i.e. 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 entitled *Relocatable module tasks* on page 4-131 for details.

Related SWIs

None

Related vectors

None

Wimp_StartTask (SWI &400DE)

On entry

R0 = pointer to * Command to be executed

On exit

R0 = handle of task started, if it is still alive; 0 otherwise
(not available in RISC OS 2.0)

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, e.g. `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

R0 corrupted
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 (e.g. 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	will not produce a 'beep' even if WimpFlags bit 4 is clear (this bit is reserved in RISC OS 2.0)
8 - 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 timeouts (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 entitled *Relocatable module tasks* on page 4-131 for details.

If you press Escape when a Wimp_ReportError box is up, the code returned is for the non-highlighted box, i.e. R1=2 if OK is highlighted, and R1=1 if `Cancel` is highlighted.

Note that RISC OS 2.0 will always return R1=1 (i.e. OK clicked), even if the `Cancel` box is highlighted.

Pressing Return selects the highlighted box, and returns 1 or 2 as appropriate.

In either case, if the box that would have been selected is not present, the other box is selected.

Related SWIs

None

Related vectors

None

Wimp_GetWindowOutline (SWI &400E0)

On entry

R1 = pointer to a five-word block

On exit

R0 corrupted
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 (or –2 to indicate the icon bar)
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

A window handle value of –2 is not available in RISC OS 2.0.

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
R3 = pointer to poll word if R0 bit 22 is set (not in RISC OS 2.0)

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'; (e.g. 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

R0 corrupted

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 on page 4-166 for details about these fields.

Under RISC OS 3 this SWI can be called from outside the redraw code of an application. In this case, the block pointed to by R1 should contain screen coordinates instead of window relative ones.

Related SWIs

None

Related vectors

None

Wimp_SetMode (SWI &400E3)

On entry

R0 = mode number

On exit

R0 corrupted

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.

On the next call to Wimp_Poll after a mode change, the Wimp issues Message_ModeChanged and Open_Window_Requests for all open windows. If the new mode is smaller than the previous one, the windows are also forced back onto the screen. This does not happen in RISC OS 2.0.

Related SWIs

None

Related vectors

None

Wimp_SetPalette (SWI &400E4)

On entry

R1 = pointer to 20-word palette block

On exit

R0 corrupted
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

R0 corrupted
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 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.

The values returned from Wimp_ReadPalette are analogous to those returned by OS_ReadPalette, in that they always have the bottom nibbles clear. These colours are not correct for passing to ColourTrans: you have to make the bottom nibbles into copies of the top ones.

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

R0 corrupted

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
window handle; message sent to window's creator, or
–2 (icon bar) and
R3 = icon handle; message sent to icon's creator, or
0; broadcast message, sent to all tasks, including the originator

On exit

R0 corrupted
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.

For a full description of all the message action codes see the section entitled *Messages* on page 4-289.

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, e.g. 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 (e.g. 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.

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

R0 corrupted

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

R0 corrupted
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 (i.e. &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 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 RMA area may move around, e.g. after a sprite file has been merged with it. In view of this, you should use Wimp_SpriteOp if possible.

Note: This call should not be used if you are writing applications that you wish to be compatible with future versions of RISC OS.

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, e.g. 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 (i.e. memory for current task)
R1 = size of next slot (i.e. desirable allocation for next task)
R2 = size of free pool (i.e. free memory)

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, i.e. 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.

Next is a number and can be larger than free, in which case next task just gets free. 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, 0 ⇒ do not fill in
R7 = a pointer to a 2, 4 or 16 byte block to receive translation table,
0 ⇒ do not fill in (must be 16 bytes long)

On exit

R0 corrupted
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

R0 corrupted
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, e.g. 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

R0 corrupted

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

R0 corrupted

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

R0 corrupted

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, i.e. 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 entitled *Data transfer protocol* on page 4-305. 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
1	R0 = current Wimp mode on exit
2	R0 = pointer to iconsprites filename suffix for the configured mode When loading sprite files containing icons, the suffix should be tried - if the file does not exist, try the original filename
3	R0 = 0 ⇒ we are in text output mode (ie outside the desktop, or in the ShellCLI, or in a command window) = 1 ⇒ we are in the desktop other values reserved (test for non-zero when looking to see whether we're in command mode or not) The Wimp also supports a code variable `Wimp$State`, which can take the following values: `commands` Wimp_ReadSysInfo (3) returns 0 `desktop` Wimp_ReadSysInfo (3) returns 1 other values should be treated as 'not `commands`'.
4	R0 = 0 ⇒ left to right text entry = 1 ⇒ right to left text entry this returns the state last set by *WimpWriteDir

Note: Values for R0 of 1, 2, 3 and 4 are only available in versions of RISC OS later than 2.0.

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

R0 corrupted

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

Wimp_GetMenuState (SWI &400F4)

On entry

R0 = 0 ⇒ report current state of tree, ignoring R2,R3
= 1 ⇒ report tree which leads up to R2,R3:
R2 = window handle
R3 = icon handle
R1 = pointer to buffer to contain result

On exit

R0 corrupted
The tree is put into the buffer in R1 in the same format as that returned by Wimp_Poll reason code 9 (Menu_Select), i.e. a list of selection indices 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

The tree returned will be null:

- if R0 = 1 and the window/icon in R2/R3 is not in the tree, or
- if R0 = 0 or 1 and the menu tree is owned by a different application, or is closed altogether.

If the window is a dialogue box, the tree returned will go up to (but not include) the dialogue box.

Note: This SWI is not available under RISC OS 2.0.

Related SWIs

None

Related vectors

None

Wimp_RegisterFilter (SWI &400F5)

Registers a filter to be called on call to or before return from Wimp_Poll

On entry:

R0 = reason code:
0 - Register / Deregister Pre-Filter
1 - Register / Deregister Post-Filter
R1 = address of filter, or 0 to de-register
R2 = value to be passed in R12 on entry to filter

On exit:

Registers 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 SWI is provided for the use of the FilterManager, and should **not** be used unless you want to replace the whole filter system. Use the FilterManager to register filters for specific tasks.

A pre filter is called whenever a task calls Wimp_Poll:

On Entry:

R0 = event mask as passed to Wimp_Poll
R1 = pointer to User block as passed to Wimp_Poll
R2 = task handle
R12 = value of R2 when registered

The task that called Wimp_Poll is paged in.

On Exit:

R0 may be modified by the filter
All other register and processor mode must be preserved

A post filter is called when the Wimp is about to return an event to a task.

On Entry:

R0 = event code for event that is about to be returned
R1 = pointer to Event block for event to be returned (Owner task paged in)
R2 = task handle of task that is about to receive the event

The task to which the event is to be returned is paged in.

On Exit:

The filter may modify R0 and the contents of the buffer pointed to by R1, to return a different event.

R1,R2 must be preserved.

If R0 = –1 on exit, the event will not be passed to the task.

Related SWIs

None

Related vectors

None

Messages

Changes applying to applications passing 300 to Wimp_Initialise

If a message is sent to a menu window, then it will be delivered to the task which opened the menu tree. This applies to any reason code greater than Close_Window_Request, as well as the messages (open, close and redraw are all dealt with automatically by the Wimp).

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, refer to the section entitled SWI *chunk numbers and names* on page 6-474.

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* on page 4-305 for details of these message actions.

Message_PreQuit (8)

This broadcast message gives applications the chance to object to a request to close down; for example, if 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 (i.e. my_ref copied into your_ref)

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.

If the Task Manager ever gets a Quit that it didn't originate, it will close itself down.

Restarting the desktop closedown sequence

Applications can tell whether they should restart the desktop closedown sequence after prompting the user to save any unsaved data. If bit 0 of the flag word is set, then the task should not send a Ctrl-Shift-F12 Key_Pressed event to the task which sent it the PreQuit message, to restart the closedown sequence, but should instead just terminate itself.

This facility is not available in RISC OS 2.0.

R1+0	24 (size)
R1+16	Message_PreQuit (8)
R1+20	flag word: bit 0 set ⇒ just quit this task, else desktop being quit bits 1 - 31 reserved (i.e. ignore them)

Note that if the flag word is not present (ie the block is too small), the task should treat the flag word as having been zero. Following this, the task should display a dialogue box giving the user the chance to either save or discard files, as he sees fit.

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.

Message_SaveDesktop (10)

R1+16	Message_SaveDesktop (10)
R1+20	(word) file handle of desktop file being written
R1+24	flag word: bits 0 - 31 reserved (ignore them)

Note that this is a RISC OS rather than a C file handle, so fprintf() cannot be used. The RISC OS SWIs OS_BPut or OS_GBPB should be used instead.

This facility is not available in RISC OS 2.0.

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 (e.g. 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 entitled *Relocatable module tasks* on page 4-131 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.

Filer Action Window

The Filer Action Window is a module which performs file manipulation operations for the Filer without the desktop hanging whilst they are under way.

The Filer Action Window is not available in RISC OS 2.0.

To drive Filer_Action you must:

1. Wimp_StartTask with a command of *Filer_Action
2. Send a sequence of messages to the new task describing the activity:
   - specify the directory in which the objects that are going to be acted upon exist (using Message_FilerSelectionDirectory);
   - specify the objects in the directory (using several Message_FilerAddSelection messages);
   - start the action using Message_FilerAction.

Filer_Action will sort out its own slot size as appropriate. If no messages are sent, then Filer_Action will kill itself.

Controlling the Filer_Action task

To set the Filer_Action going, the following messages are sent:

Message_FilerSelectionDirectory (&403)
Message_FilerAddSelection (&404)
Message_FilerAction (&405)

The selection directory is the name of the directory in which the selection of files being operated upon lies. AddSelection sends a set of files which are to be added to the list of files in the selected directory. You should just send a space separated list of leaf names of the selected objects.

FilerAction starts the operation going.

Once the Filer_Action is going it can be controlled by using the Message_FilerControlAction message.

Message_FilerSelectionDirectory (&403)

The data for this message should be a nul-terminated name of a directory. Sending this message clears out the current selection of files.

This message is not available in RISC OS 2.0.

Message_FilerAddSelection (&404)

The data for this message should be a nul-terminated string which is a space separated list of leaf names of objects in the selection directory which are to be operated upon. This adds the given names to the list.

This message is not available in RISC OS 2.0.

Message_FilerAction (&405)

The format of the data for this message takes the following form:

Word	Meaning		
0	Operation to be performed:		
	0	Copy	Copy a number of objects from one directory to another
	1	Move (rename)	Move a number of objects from one directory to another by trying a rename first then doing a copy/delete if that fails
	2	Delete	Delete a number of objects in a particular

directory

3	Set access	Set the access of a number of objects to a given value
4	Set type	Set the file type of a number of objects to a given value
5	Count	Count the file sizes of the selected objects
6	Move (by copying and deleting afterwards)	Move a number of objects from one directory by copying them then deleting the source
7	Copy local (within directory)	Copy a single object to a different name in the same directory
8	Stamp files	Stamp the selected objects with the time when they get stamped
9	Find file	Find an object with a given name.

1 Option bits:

Bit	Meaning when set
0	Verbose
1	Confirm
2	Force
3	Newer (as opposed to copying always)
4	Recurse (only applies to access)

2 onwards

Information specific to the particular operation:

Operation		Meaning
0	Copy	nul terminated destination directory
1	Move (rename)	nul terminated destination directory
2	Delete	unused
3	Set access	How to set the access The 1st two bytes are the access values to be set The 2nd two bytes are a mask which, when set, disable the corresponding access bit from being set
4	Set type	Numeric file type to set
5	Count	unused
6	Move (copy/delete)	nul terminated destination directory
7	Copy local	nul terminated destination object name
8	Stamp	unused
9	Find	nul terminated name of object to find

This message is not available in RISC OS 2.0.

Message_FilerControlAction (&406)

The 1st word determines what control is to be performed:

0 Acknowledge the control message (to check FilerAction is still going)
1 Show the action window (turn verbose on)
2 Hide the action window (turn verbose off)

This message is not available in RISC OS 2.0.

Message_FilerOpenDirAt (&402)

This message is not available in RISC OS 2.0.

Message_FilerSelection (&407)

This message is not available in RISC OS 2.0.

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 (e.g. 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 (e.g. 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

i.e. 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 (i.e. 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.

Message_TaskStarted (&400C8)

This is sent by the Filer after it has started up all the desktop filers so that the TaskManager can 'renumber' it. This is so that during the deskboot saving sequence, the Filer_Boot and Filer_OpenDir commands are inserted after the logons returned by the NetFiler.

This message is not available under RISC OS 2.0.

Message_MenusDeleted (&400C9)

This message is returned by the Wimp, with block+20 = menu pointer for the menu tree that was deleted, in the following circumstances:

- if a task has a menu tree open, and another task calls Wimp_CreateMenu, thereby deleting the first tree;
- if a task has a menu tree open, and it calls Wimp_CreateMenu with a different menu pointer than the one last used;
- if a task has a menu tree open, and the user clicks somewhere outside the menu tree, thereby closing it. The Wimp now sends mouse clicks as messages if the message queue is not empty, which ensures that the click event arrives after the Message_MenusDeleted.

Note in particular that no message is returned if a menu selection event is returned, or if a menu tree is replaced by another with the same menu pointer.

This message is not available under RISC OS 2.0.

Application messages

Alarm

In addition to the 'normal' user facilities of !Alarm as documented in the RISC OS *User Guide*, it is also possible for applications to set and receive alarms by using some WIMP messages. These are as follows:

- To set or cancel an alarm send Message_AlarmSet.
- When an alarm goes off !Alarm broadcasts Message_AlarmGoneOff.

Message_AlarmSet (&500)

Setting an alarm

To set an application alarm, send the following message:

R1+16	&500	indicates message to !Alarm
R1+20	0/1	indicates set an alarm (1 if 5 byte format)
R1+24	date/time	
R1+30	name of application sender, terminated by 0	
R1+n	application-specific unique alarm identifier, terminated by 0	

Date & time must be given in standard 5 UTC byte format if +20 is 1, otherwise the layout is as follows (local time values):

R1+24	year as low-byte/high-byte
R1+26	month
R1+27	date
R1+28	hour
R1+29	minutes

Neither the name nor the alarm identifier may be longer than 40 chars each.

Cancelling an alarm

To cancel the alarm, use the following message block:

R1+16	&500	indicates message to !Alarm
R1+20	2	indicates cancel an alarm
R1+24	name of application, terminated by 0	
R1+n	application-specific unique alarm identifier, terminated by 0	

The name & identifier must match exactly for the alarm to be successfully cancelled. It is not necessary to specify the time of the alarm, as this may have changed due to being deferred by Alarm.

If these messages are sent recorded, !Alarm will acknowledge with 0 if successful, or a 0 terminated error string (message type = &500).

This message is not available in RISC OS 2.0.

Message_AlarmGoneOff (&501)

The format of the block sent by !Alarm as a broadcast is:

R1+16	&501	indicates an alarm has gone off
R1+20	name of application sender, terminated by 0	
R1+n	application-specific unique alarm identifier, terminated by 0	

If the named application recognises the identifier, it must acknowledge this message, otherwise !Alarm will ask the user to install the named application. If the latter occurs, the alarm is deferred for one minute to allow the application to be installed.

This message is not available in RISC OS 2.0.

Help

For an application to use interactive help, two application messages are employed. One is used by Help to request the help text, and the other is used by the application to return the text message.

Message_HelpRequest (&502)

To request help, the Help application must send a message as follows:

R1+16	&502 - indicates request for help	
R1+20	mouse x co-ordinate	
R1+24	mouse y co-ordinate	
R1+28	mouse button state	
R1+32	window handle	(-1 if not over a window)
R1+36	icon handle	(-1 if not over an icon)

Locations 20 onwards are the results of using Wimp_GetPointerInfo.

The Wimp will pass this message automatically to the task in charge of the appropriate window/icon combination.

The Help application issues message type &502 every $^1/_{10}$th of a second to allow applications such as Edit and Draw to change the help text according to the current edit mode. To avoid flicker, the display is only updated when the returned help string changes.

With certain applications, such as the Filer, no interactive help is supplied and the Help application supplies some default message sin instances like this.

Message_HelpReply (&503)

If an application receives a Message_HelpRequest, and wishes to produce some interactive help, it should respond with the following message:

R1+16	&503
R1+20	help message, terminated by 0

The help text may contain any printable character codes (including top-bit-set ones). If the sequence |M is encountered, this will be treated as a line break and subsequent text will be printed on the next line in the window. If !Help needs to split a line because it is too long, it does so at a word boundary (space character).

The help text is terminated by a null character.

The desktop save protocol

Once the file to be saved is known, the save protocol can start:

1 The Task Manager first opens the output file and makes a note of the handle.

2 The Task Manager then inserts a comment saying when the file was created, so that when the user refers to the file they will know how recent it is.

3 The Task Manager then inserts four *commands:

- WimpSlot -next <wimp slot 'next' size>K
- ChangeDynamicArea -FontSize <font area size>K
- ChangeDynamicArea -SpriteSize <system sprite area size>K
- ChangeDynamicArea -RamFsSize <RAM disc size>K

These set the sizes of the 'Next' slot, the font and sprite area sizes, and the RAM disc size, as would be expected. It is not sensible to set the RMA size or the system stack in this way, as they are much more system-dependent than those described above. The screen size cannot be set as it is always reset to the size of the current screen mode by the Task Manager.

If there is not enough memory free to be allocated for a particular slot then, instead of giving errors, the largest amount of memory which is free will be allocated to the slot.

When the user selects **Exit** or **Shutdown** from the task manager's menu, it looks to see if the variable SaveDeskSFile is set up - if it is, it automatically saves the desktop state in this file before exiting.

4 Rather than using broadcast messages, the Task Manager talks to all the other tasks by using its list of task handles and names. This ensures that the tasks are asked to restart in the same order as they were originally started (which is not true for broadcasts).

5 For each task in its list, the task manager sends a Message_SaveDesktop (see page 4-291).

6 If the task understands the message, it then writes data directly into the desktop file, using the file handle supplied.

The data is a sequence of *commands suitable for inclusion in a Desktop file, each terminated by a linefeed character (&0A). When the file is run to start the desktop, each command will be executed as a separate Wimp task.

A typical example for a C application follows:

```
#include <os.h>
#include <swis.h>
```



```
os_error *save_desktop(int handle)
{
  char *ptr;

  for (ptr=getenv ("Edit$Dir"); *ptr; ptr++) {
    os_error *error = os_swi2(OS_BPut, *ptr, handle);
    if (error) return error;
  }

  return os_swi2(OS_BPut, 10, handle); /* line terminator */
}
```

The data the application should add to the boot file is a restart command which is usually a GSTrans'd form of something like /<Edit$Dir>.

Note that since several copies of !Edit can be loaded at once, this GSTrans-ing operation should be done as soon as the application is loaded (and the result stored in a buffer), in case the value of Edit$Dir changes subsequently.

Resident modules

Resident module tasks do not require a restart command of the above form, since they are automatically started when the desktop is entered (by means of the Service_StartWimp protocol). However, if the modules are not stored in the ROM, they will probably be loaded by means of some form of *RMEnsure command in a !foo application, so the !foo application should be re-run instead.

There is a service call provided for modules which need to save some state to the file, e.g. ColourTrans saves its calibration. For details of this call see the section entitled *Service_WimpSaveDesktop (Service Call &5C)* on page 4-150.

7 If the message is **not** acknowledged, the task manager goes on to the next one in the list. This means that:

- Tasks which don't understand desktop saving will not be saved in the desktop file.
- If an application gets an error while writing to the file, it should acknowledge the message and report the error. The Task Manager will detect that the message has been acknowledged, and will abort the save operation and remove the file.

8 When all the tasks have been asked for their restart commands, the file is closed, and if the output was a boot file, *Opt 4,2 is executed for the appropriate disc drive / user id.

The device claim protocol

Under RISC OS there are a number of devices which can only be used by one task at a time, such as the serial and parallel ports. This protocol provides a method by which a task can claim one of those devices for its exclusive use.

1 A task wishing to claim exclusive use of a device broadcasts a Message_DeviceClaim message.

2 If a task which currently owns a device wishes to prevent another task from claiming the device it should reply to the above message with a Message_DeviceInUse message. If a Message_DeviceInUse is received in reply, the claim has failed, and the task should issue an error message.

3 If a DeviceClaim message sent by a task is not acknowledged, the task can assume it has claimed the device.

Note: It is legal for a task to claim a device it already owns, as long as it does not object to its own requests.

This protocol can be used under RISC OS 2.0, but will not be used by applications written for it, such as printer drivers prior to version 2.42.

Device Numbers

Currently allocated device numbers are:

	Major device	Minor Device	
Parallel port	1	0	Internal port
Serial port	2	0	Internal port
Screen palette	3	0	
Midi Interface	4	-1	All ports
		0-3	Port number
Floppy discs	5	-1	All floppy discs
		0-3	Drive number (:0 - :3)
Sound system	6	0	Entire sound system

Example

The printer drivers use the above protocol in the following way:

- If the printer driver starts up with the serial port selected it tries to claim the serial port (Major Device 2, Minor Device 0). If it fails, it issues an error message and selects Null: as its output.
- Whenever the user selects **Serial** from the printer driver's menu, the printer driver tries to claim the serial port, and if it fails it issues an error message and leaves the setting as it was.

- If the printer driver receives a DeviceClaim message while the serial port is selected as its destination, it replies with a DeviceInUse message.

The same procedure is followed for the parallel port.

Note: There is no need to release a device after you have finished using it, you should simply stop objecting to other tasks claiming it.

When a task exits, it no longer objects to other tasks claiming devices, and so all the devices it owned are effectively released.

Message_DeviceClaim (11)

R1+16	Message_DeviceClaim (11)
R1+20	major device number
R1+24	minor device number
R1+28	zero terminated information string

This message is broadcast by a task wishing to claim exclusive use of a device.

The information string should contain the name of the application claiming the device.

Message_DeviceInUse (12)

R1+16	Message_DeviceInUse (12)
R1+20	major device number
R1+24	minor device number
R1+28	Zero terminated information string

If a task which currently owns a device wishes to prevent another task from claiming the device it should reply with Message_DeviceInUse.

The information string should be used to give information about the task currently using the device (for example, 'Serial terminal connection open' if a terminal currently owns the serial port). This information can then be used by the task trying to claim the device in its error message.

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:

1 Message_DataSave
2 Message_DataSaveAck
3 Message_DataLoad
4 Message_DataLoadAck

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:

1. The user releases the mouse button, terminating the drag of the file icon; the application receives a User_Drag_Box event.
2. The application calls Wimp_GetPointerInfo (SWI &400CF) to find out where the icon was dropped, in terms of its coordinates and window/icon handles.
3. The application sends a DataSave message with the file's leafname to the Filer using this information.
4. The Filer replies with a DataSaveAck message, which contains the complete pathname of the file.
5. The application saves the data to that file.
6. The application sends the message DataLoad to the Filer.
7. 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:

1. The user releases the mouse button, terminating the drag of the file icon; the application receives a User_Drag_Box event.
2. The application calls Wimp_GetPointerInfo to find out where the icon was dropped, in terms of its coordinates and window/icon handles.
3. The application sends a DataSave message with the file's leafname to the destination application using this information.
4. The destination application replies with a DataSaveAck message, which contains the pathname `<Wimp$Scrap>`.
5. The application saves the data to that file (which the filing system expands to an actual pathname).
6. The application sends the message DataLoad to the destination task.
7. The external task loads and deletes the scrap file.
8. 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.

1. The Filer sends the DataLoad message to the application.
2. 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:

1. The external task sends a DataSave message to the application.

2 The application replies with a DataSaveAck message, quoting the pathname <Wimp$Scrap>.

3 The external task saves its data to that file.

4 The external task sends the message DataLoad to the application.

5 The application loads and deletes the file <Wimp$Scrap>.

6 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, i.e. 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	untyped file (i.e. 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 (i.e. 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', i.e. 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.

Message_DataSaved (13)

R1+12	reference from DataSave message
R1+16	13

In some cases a file can become 'safe' after the DataSaveAck has been sent. This message can be used to tell the originator of the save that the file has become 'safe'. The reference at R1+12 should be the one from the my_ref field of the original DataSave message.

In order to make use of this message, the saving task should store the my_ref value of the DataSave message with each document it tries to save. On receiving the DataSaved message it should compare its reference number with the number stored for each active document, and mark the document as saved if the numbers match. Note that a document can be modified by the user between the time that the DataSave message was sent and the time that the DataSaved message is received; in this case, the task should forget any reference number it holds for the document, and ignore any subsequent DataSaved messages.

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 in the section entitled *Saving data to another application* on page 4-307 up until the DataSave message. Then:

1. The external task replies with a RAMFetch message.
2. The application sends a RAMTransmit message with data.
3. The external task replies with another RAMFetch message.
4. The last two steps continue until all the data has been sent and received.

Loading data from another application (memory)

1. The external task sends a DataSave message to the application.
2. The application replies with a RAMFetch message.
3. If this isn't acknowledged with a RAMTransmit, use the <Wimp$Scrap> file to perform the operation, otherwise...
4. Get and process the data from the RAMTransmit buffer.
5. 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 (i.e. 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, e.g. 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.

The iconize protocol

This protocol is not available in RISC OS 2.0.

Shift held down when the close tool of a window is clicked

If shift is held down when the close tool of a window is clicked, the wimp does not close the window, but instead broadcasts a Message_Iconize.

If no iconizer is loaded nothing happens.

If iconizer is loaded:

1. It acknowledges the message (stops the broadcast).
2. It sends a Message_WindowInfo to the window.

Old application

If the application is an old one it will ignore the above message.

Iconizer gets acknowledgement back and uses the information in the first Message_Iconize to iconize the window.

New application

If the application is a new one it will react as follows:

- If it doesn't want to help it will ignore the message.

- If it wants to help it will reply with a Message_WindowInfo. The iconizer will then use this info to iconize the window.
- This should enable applications such as edit to give a different icon depending on the file type of the file being edited in the window.
- If the application wants to iconize its own window it acknowledges the original Window_Info message, and does all the work itself.

Closing a window

Whenever a window is closed the Wimp broadcasts the message Message_WindowClosed.

The iconizer should then remove the icon.

When a task exits

The iconizer should spot the Message_TaskQuit and remove all the icons for that task.

When a new iconizer starts up

It broadcasts a Message_WindowInfo.

An iconizer receiving this message should reopen all iconized windows.

All applications should ignore such a message.

Current iconizer (Pinboard) behaviour

If it does not get a reply to the Message_WindowInfo

1. It gets the task name for the task that owns the window and then tries to find a sprite called `ic_`*`task name`* in the wimp sprite area. If it fails it uses a sprite called `ic_?`.
2. It uses the title given in the Message_Iconize.

If it gets a Message_WindowInfo

1. It tries to find the sprite `ic_`*`name given in message`*. If it fails it uses `ic_?`.
2. It uses the title given in the Message_WindowInfo.

Message_Iconize (&400C10)

R1+20	window handle
R1+24	task handle for task which owns the window
R1+28	20 Bytes of title string (last part of first word)
R1+48	

This message is not available in RISC OS 2.0.

Message_WindowInf (&400C11)

R1+20	window handle
R1+24	reserved, must be 0
R1+28	sprite name to use, null terminated (MAX = 7 chars + NULL) sprite name used is `icon_string`
R1+36	title string to use null terminated (as short as possible truncated to 20 characters)

This message is not available in RISC OS 2.0.

Message_WindowClosed (&400C12)

R1+20	window handle
R1+24	

This message is not available in RISC OS 2.0.

The Printer protocol

The printer application has two main functions; managing printer selection and printing files in the background (scheduled on nul events).

The protocol for entering a file into the print queue is:

1 The application issues either:

DataSave (user has dropped file onto printer icon), or
PrintSave (user has initiated an application print option).

2 !Printers replies with PrintError if there is an error or PrintFile.

3 The application does one of the following:

- ignores PrintFile, in which case !Printers will respond with DataSaveAck, to which the application should respond with a DataLoad;
- replies with WillPrint and then proceeds to print the file;
- converts the file, stores the output in `Printer$Temp`, and replies with DataLoad.

4 Upon receipt of a DataLoad, the file gets queued and the application is sent a DataLoadAck.

When !Printers is ready to print the file it broadcasts PrintTypeOdd. The application can then decide whether to print or not.

Message_PrintFile (&80140)

This message is broadcast as a recorded delivery upon receipt of a DataSave or PrintSave message. The reason for having this message is two-fold:

- the application doing the DataSave might need to know that, in effect, the user is wanting to print;
- it allows applications to try and improve on !Printer-provided services such as text printing.

The format of the message is:

R1+12	your_ref
R1+16	&80140
R1+20	
...	from DataSave/PrintSave block
R1+44	

This allows any application to try and do better than !Printers can do with the default actions available to it. Such an application has 3 options:

- it can ignore the message, in which case, if no-one else claims it, !Printers will resort to the normal processes (i.e. issue a DataSaveAck);
- it can respond with WillPrint, in which case !Printers takes no further action;
- it can convert the file into another format and store it in the file specified by `Printer$Temp`. It should then reply with a DataLoad with the filetype reflecting the new type.

Note: It is recommended that you use the PrintTypeOdd protocol in preference to Message_PrintFile.

Message_WillPrint (&80141)

This message is sent by an application in response to a PrintFile broadcast. The application should then proceed to print the file.

Note: It is recommended that you use the PrintTypeOdd protocol in preference to this message.

Message_PrintSave (&80142)

The format of this message is:

R1+12	0
R1+16	&80142
R1+20	
...	as for Message_DataSave
R1+44	

This message allows applications to send files to the printer manager for printing without having to know the task handle, etc, since the message is broadcast. The message simply needs to be broadcast as a recorded delivery, at which point the printer manager will enter the PrintFile dialogue. If the message bounces, the application should complain as the printer manager is not loaded.

Message_PrintInit (&80143)

This is broadcast when a printer manager is starting up. Any active printer managers should quit quietly upon receipt of this message to avoid a clash occurring.

Message_PrintError (&80144)

Under RISC OS 2.0

This message is sent by RISC OS 2.0 managers in response to a PrintSave if they are already printing (as they can only queue one file at a time). It is known as Message_PrintBusy under RISC OS 2.0.

Under !Printers

With !Printers, this message is sent if an error occurs as a result of one of the other messages being used. The format of the block is:

R1+12	your_ref
R1+16	&80144
R1+20	error number
R1+24	error message (null terminated)

To maintain compatibility with RISC OS 2.0 printer managers, if the message is the original Message_PrintBusy, the size (in R1+0) will be 20.

Error numbers and messages

```
1       Can only print from applications when a printer has been selected
```

This is sent in reply to a PrintSave when there isn't a selected printer.

Message_PrintTypeOdd (&80145)

This message is broadcast if the filetype is not considered known by !Printers. 'Known' is qualified as being the current printer type (e.g. FF5 for PostScript), text, obey or command files. The format of the message is:

R1+12	0
R1+16	&80145
R1+40	file type of data
R1+44	zero terminated filename

If an application can print this filetype directly, it should respond with PrintTypeKnown. The application can either:

- print the file directly to printer;
- output it to `Printer$Temp`, in which case this must be done before replying with PrintTypeKnown.

Message_PrintTypeKnown (&80146)

This message is sent by an application in response to a PrintTypeOdd.

Message_SetPrinter (&80147)

This message is broadcast by !Printers when the printer settings or selection has changed.

Message_PSPrinterQuery (&8014C)

This message is sent as a recorded delivery by !FontPrint to !Printers when !FontPrint either starts up or receives SetPrinter. The layout of the block is:

R1+12	0
R1+16	&8014C
R1+20	buffer address (or zero)
R1+24	buffer size

If the buffer address is non-zero, !Printers places the following information into the buffer (all Null terminated):

- current printer name,
- current printer type,
- pathname to printer font file.

Regardless of the buffer address, !Printers places the real buffer size into the block and replies with PSPrinterAck.

This message is not available in RISC OS 2.0.

Message_PSPrinterAck (&8014D)

This is sent by !Printers to !FontPrint in response to PSPrinterQuery. If !FontPrint does not receive this message, it should raise an error to advise the user (e.g. !Printers is required to allow use of !FontPrint).

This message is not available in RISC OS 2.0.

Message_PSPrinterModified (&8014E)

This is sent by !FontPrint to !Printers when the user clicks on the **Save** button. !Printers then re-reads the font file and resets the printer's font list.

This message is not available in RISC OS 2.0.

Message_PSPrinterDefaults (&8014F)

This is sent by FontPrint to !Printers when the user clicks on the **Default** button. !Printers then resets the font file, resets the printer's font list and replies with PSPrinterDefaulted.

This message is not available in RISC OS 2.0.

Message_PSPrinterDefaulted (&80150)

This is sent by !Printers to !FontPrint when the font file has been reset.

This message is not available in RISC OS 2.0.

Message_PSPrinterNotPS (&80151)

This is sent by !Printers upon receipt of PSPrinterQuery if the currently selected printer is not a PostScript printer.

This message is not available in RISC OS 2.0.

Message_ResetPrinter (&80152)

This can be sent to !Printers to ensure that the printer settings are correct for the currently selected printer.

This message is not available in RISC OS 2.0.

Message_PSIsFontPrintRunning (&80153)

If !FontPrint receives this message, it will acknowledge it.

This message is not available in RISC OS 2.0.

The DataOpen Message

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 file type is set to &3000 for untyped files.

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.

*Commands

*Configure WimpAutoMenuDelay

Sets the configured time before a submenu is automatically opened

Syntax

`*Configure WimpAutoMenuDelay` *delay*

Parameters

delay	time before a submenu is automatically opened, in 1/10 second units

Use

*Configure WimpAutoMenuDelay sets the configured time the pointer must rest over a menu item before its submenu (if any) is automatically opened.

Note that automatic opening of submenus is disabled if bit 7 of the WimpFlags is clear.

Example

```
*Configure WimpAutoMenuDelay 5
```

Related commands

*Configure WimpFlags, *Configure WimpMenuDragDelay

*Configure WimpDoubleClickDelay

Sets the configured time during which a double click is accepted

Syntax

`*Configure WimpDoubleClickDelay` *delay*

Parameters

delay	time during which a double click is accepted, in 1/10 second units

Use

*Configure WimpDoubleClickDelay sets the configured time after a single click during which a double click is accepted.

A pending double-click will be immediately cancelled if any of the following occur:

- Wimp_DragBox is called (for example, in response to a drag button event);
- the pointer moves by more than the configured number of OS units;
- the mouse is not clicked again inside the configured amount of time.

Example

```
*Configure WimpDoubleClickDelay 12
```

Related commands

*Configure WimpDoubleClickMove

*Configure WimpDoubleClickMove

Sets the configured distance within which a double click is accepted

Syntax

```
*Configure WimpDoubleClickMove distance
```

Parameters

distance	distance within which a double click is accepted, in OS units

Use

*Configure WimpDoubleClickMove sets the configured distance from the position of a single click within which a double click is accepted.

If the pointer moves this distance or further from the first click, the double click is cancelled.

Example

```
*Configure WimpDoubleClickMove 20
```

Related commands

*Configure WimpDoubleClickDelay

*Configure WimpDragDelay

Sets the configured time after which a drag is started

Syntax

```
*Configure WimpDragDelay delay
```

Parameters

delay	time after which a drag is started, in 1/10 second units

Use

*Configure WimpDragDelay sets the configured time after a single click after which a drag is started.

Example

```
*Configure WimpDragDelay 8
```

Related commands

*Configure WimpDragMove

*Configure WimpDragMove

Sets the configured distance the pointer has to move for a drag to be started

Syntax

```
*Configure WimpDragMove distance
```

Parameters

distance	distance the pointer has to move for a drag to be started, in OS units

Use

*Configure WimpDragMove sets the configured distance from the position of a single click that the pointer has to move for a drag to be started.

Example

```
*Configure WimpDragMove 40
```

Related commands

*Configure WimpDragDelay

*Configure WimpFlags

Sets the configured behaviour of windows when dragged, and of error boxes

Syntax

```
*Configure WimpFlags n
```

Parameter

n 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
5	windows can be dragged partly off screen to right and bottom
6	windows can be dragged partly off screen in all directions
7	open submenus automatically If set and the pointer is kept on a non-leaf menu item for more than the time specified by *Configure WimpAutoMenuDelay then the submenu will be opened automatically by the Wimp.

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

*Configure WimpFlags sets the configured behaviour of windows when dragged, and of error boxes. 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. Bits 5 and 6 control whether the window can move partly off screen (even if bit 6 is clear). Bit 7 controls whether submenus are automatically opened when the pointer rests over their parent entry for longer than the configured WimpAutoMenuDelay.

Examples

```
*Configure WimpFlags 0
*Configure WimpFlags 15
```

Related commands

*Configure WimpAutoMenuDelay, *Status WimpFlags

Related SWIs

Wimp_Poll, Wimp_OpenWindow, Wimp_ReportError

*Configure WimpMenuDragDelay

Sets the configured time before an automatically opened submenu is closed

Syntax

```
*Configure WimpMenuDragDelay delay
```

Parameters

delay	time before an automatically opened submenu is closed, in 1/10 second units

Use

*Configure WimpMenuDragDelay sets the configured time before an automatically opened submenu is closed. During this time you can move the pointer over other menu entries without closing the submenu, making it easy to reach the submenu.

Note that automatic opening of submenus is disabled if bit 7 of the WimpFlags is clear.

Example

```
*Configure WimpMenuDragDelay 7
```

Related commands

*Configure WimpFlags, *Configure WimpMenuDragDelay

*Configure WimpMode

Sets the configured screen mode used

Syntax

```
*Configure WimpMode screen_mode
```

Parameter

screen_mode	the display mode that the computer should use after a power-on or hard reset, and when entering or leaving the desktop

Use

*Configure WimpMode sets the configured screen mode used by the machine when it is first switched on, or after a hard reset, and when entering or leaving the desktop. It is identical to the command *Configure Mode; the two commands alter the same value in CMOS RAM.

Under RISC OS 2.0, this command only sets the configured screen mode used for the Desktop; *Configure Mode sets the configured screen mode used for the command line. 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

*Configure Mode

Related SWIs

Wimp_SetMode

Related vectors

None

*Desktop

Initialises all desktop facilities, then starts the Desktop.

Syntax

```
*Desktop [command|-File filename]
```

Parameters

command	a *Command which will be passed to Wimp_StartTask when the Desktop starts up
filename	a valid pathname specifying a file, each line of which will be passed to Wimp_StartTask when the desktop starts up

Use

*Desktop initialises all desktop facilities, then starts the Desktop. The Desktop provides an environment in which Wimp programs can operate.

*Desktop automatically starts resident Wimp task modules such as the filers, the palette utility and the Task Manager. You can also run an optional * command or each line of a file of * commands. This is typically used to load applications such as Edit. Any * commands using files must specify them by their full pathname.

If you do run a file of * commands when you start the desktop, its first line should run the file !System!Boot, provided with your computer. This is needed by most desktop applications. If you want to start an application that uses fonts, the next line of the start-up file should run !Fonts.!Boot, again provided with your computer. Applications can then be started on the following lines.

The Desktop may also be configured as the default language, using the command:

```
*Configure Language 4
```

Examples

```
*Desktop
*Desktop !FormEd
*Desktop -File !DeskBoot
```

Related commands

*DeskFS, *Desktop_Filer, *Desktop_ADFSFiler et al.

Related SWIs

Wimp_StartTask

Related vectors

None

*Desktop_...

Commands to start up ROM-resident Desktop utilities

Syntax

```
*Desktop_ADFSFiler, *Desktop_Configure, *Desktop_Draw,
*Desktop_Edit, *Desktop_Filer, *Desktop_Free,
*Desktop_NetFiler, *Desktop_Paint, *Desktop_Palette,
*Desktop_Pinboard, *Desktop_RAMFSFiler,
*Desktop_ResourceFiler, *Desktop_TaskManager
```

Parameters

None

Use

*Desktop_... commands are used by the Desktop to start up ROM-resident Desktop utilities that appear automatically on the icon bar. However, they are for internal use only, and you should not use them; use *Desktop instead. If you do try to use these commands outside the desktop, an error is generated. For example, *Desktop_Palette will give the error message 'Use *Desktop to start the Palette utility'.

The reason why these commands have to be provided is that it is only possible to start a new Wimp task using a command line.

There is one *Desktop_... command that we've documented, because it appears in desktop boot files. This is *Desktop_SetPalette.

Related commands

*Desktop, *Desktop_SetPalette

Related SWIs

Wimp_StartTask

Related vectors

None

*Desktop_SetPalette

Alters the current Wimp palette

Syntax

```
*Desktop_SetPalette RGB0 … RGB15 RGBbor RGBms1 … RGBms3
```

Parameters

All parameters specify palette entries as 6 hex digits of the form *BBGGRR*.

RGB0 … RGB15	16 parameters giving the palette values for Wimp colours 0 - 15
RGBbor	1 parameter giving the palette value for the border
RGBms1 … RGBms3	3 parameters giving the palette values for pointer colours 1 - 3

Use

*Desktop_SetPalette alters the current Wimp palette.

Example

```
*Desktop_SetPalette FFFFFF DDDDDD BBBBBB 999999 777777
555555 333333 000000 994400 00EEEE 00CC00 0000DD BBEEEE
008855 00BBFF FFBB00 777777 FFFF00 990000 0000FFS
```

Related commands

None

Related SWIs

Wimp_SetPalette (SWI &400E4)

Related vectors

None

*IconSprites

Merges the sprites in a file with those in the Wimp sprite area

Syntax

```
*IconSprites filename
```

Parameters

filename	full name of file to load

Use

*IconSprites merges the sprites in a 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.

Under RISC OS 3 *IconSprites will first try to add a suffix which depends on the properties of the configured Wimp mode, and if this doesn't work will use the original filename as usual.

If the configured Wimp mode is a high resolution mono mode (i.e. bit 4 of the modeflags is set), then it will use the suffix '23'; otherwise the suffix is:

<OS units per pixel (x)><OS units per pixel (y)>'

For example:

Configured Wimp mode	Suffix
23	'23'
20	'22'
12	'24'

This allows applications to provide an alternative set of icons for high resolution mono modes (when using the new Wimp). For example, an application could provide a set of colour sprites in a file called `!Sprites`, and an alternative monochrome set in a file called `!Sprites23`, and then load one set or the other automatically by using `*Iconsprites <Obey$Dir>.Sprites`.

Example

```
*IconSprites <Obey$Dir>.!Sprites
```

Related commands

*SLoad, *SMerge, *SSave, *Pointer

Related SWIs

Wimp_SpriteOp

Related vectors

None

*Pointer

Turns the mouse pointer on or off

Syntax

```
*Pointer [0|1]
```

Parameters

0 or 1 or nothing

Use

*Pointer turns on or off the pointer that appears on screen to reflect the mouse position. If you give either no parameter or a parameter of 1, pointer 1 is set to the default shape held in the Wimp sprite ptr_default (a blue arrow) and the sprite colours are set to their default. The pointer is enabled. If you give a parameter of 0, the pointer is disabled.

Wimp programs that re-program the pointer should use shape 2. Pointer shapes 3 and 4 are used by the Hourglass module.

You can move the pointer with OS_Word 21,5 if the mouse and pointer are unlinked. You can read the pointer position at any time using OS_Word 21,6.

Example

```
*Pointer 0          turn off the pointer
```

Related commands

None

Related SWIs

OS_Word 21 (SWI &07), Wimp_SetPointerShape, Wimp_SpriteOp

Related vectors

None

*WimpMode

Changes the current screen mode used by the Desktop

Syntax

*WimpMode *screen_mode*

Parameters

screen_mode	the display mode that the Desktop should use

Use

*WimpMode changes the current screen mode used by the Desktop.

It does not alter the configured value, which will be used next time the computer is switched on, or after a hard reset, and when entering or leaving the desktop.

Example

```
*WimpMode 20
```

Related commands

*Configure WimpMode

Related SWIs

Wimp_SetMode

Related vectors

None

*WimpPalette

Uses a palette file to set the Wimp's colour palette

Syntax

*WimpPalette *filename*

Parameters

filename	pathname of a file of type &FED (Palette).

Use

*WimpPalette uses a palette file to set the Wimp's colour palette. Typically the file would have been saved using the Desktop's palette utility. If the file is not a Palette file, the error message 'Error in palette file' is generated. 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 backwards compatibility, but since the palette utility always saves files in format (2), you should use this in preference.

The RunType for Palette files is *WimpPalette %0, so you can also set a new palette from the Desktop simply by double-clicking on the file's icon.

Example

```
*WimpPalette greyScale
```

Related commands

None

Related SWIs

Wimp_SetPalette

Related vectors

None

*WimpSlot

Changes the memory allocation for the current Wimp task

Syntax

```
*WimpSlot [-min] minsize[K] [[-max] maxsize[K]] [-next] size[K]
```

Parameters

minsize	the minimum amount of application space, in bytes or Kilobytes, that the current Wimp application requires
maxsize	the maximum amount of application space, in bytes or Kilobytes, that the current Wimp application requires
next	sets the size of the next slot

Use

*WimpSlot changes the memory allocation for the current Wimp task. It is typically used within Obey files called !Run, which the Filer uses to launch a new Wimp application. *WimpSlot 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 minsize bytes free, the error 'Application needs at least *minsize*K to start up' is generated.

Otherwise, if the current slot is smaller than minsize, then its size will be increased to minsize. If the current slot is already between minsize and maxsize, then it is unaltered. If a maxsize is specified, and the current slot is larger than maxsize, then its size will be reduced to maxsize.

The slot size that is set by this command will also apply to the application that the *Obey file finally invokes.

The next slot size is automatically saved in a desktop boot file. You can therefore alter the initial default slot either by dragging the **Next** slider in the Task manager's Task display window before saving a desktop boot file, or by editing the desktop boot file.

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

*WimpTask starts up a new task. It simply passes the supplied command to the SWI Wimp_StartTask. *WimpTask can only be called from active Wimp tasks (i.e. ones that have called Wimp_Initialise).

*WimpTask will exit via OS_Exit if you call it from outside a Wimp task.

In RISC OS 2.0 the command can only be used from within another task.

Example

```
*WimpTask myProg
```

Related commands

*WimpSlot

Related SWIs

Wimp_StartTask

Related vectors

None

*WimpWriteDir

Sets the direction of text entry for writable icons

Syntax

```
*WimpWriteDir 0|1
```

Parameters

0	write direction is the default for the current territory
1	write direction is the reverse of the default for the current territory

Use

*WimpWriteDir sets the direction of text entry for writable icons to either the default for the current territory, or the reverse of that.

Example

```
*WimpWriteDir 0
```

Related commands

None

Related SWIs

None

Related vectors

None

48 Pinboard

* Commands

*AddTinyDir

Adds a file, application or directory icon to the icon bar

Syntax

```
*AddTinyDir [object]
```

Parameters

object	a valid pathname specifying a file, application or directory

Use

*AddTinyDir adds a file, application or directory to the icon bar. If no pathname is given, it adds a blank directory icon to the icon bar. You can then later install a file, application or directory on the icon bar by dragging it to the blank icon.

Example

```
*AddTinyDir adfs::MHardy.$.!System
```

Related commands

*Pin, *RemoveTinyDir

Related SWIs

None

Related vectors

None

*BackDrop

Puts a sprite on the desktop background

Syntax

```
*BackDrop filename
```

Parameters

filename	a valid pathname, specifying a sprite file

Use

*BackDrop puts the first sprite in the given sprite file on the desktop background

Example

```
*BackDrop adfs::Disc4.$.Sprites.desert
```

Related commands

None

Related SWIs

None

Related vectors

None

*Pin

Adds a file, application or directory to the desktop pinboard

Syntax

```
*Pin object x y
```

Parameters

object	a valid pathname specifying a file, application or directory
x	the x-coordinate at which to pin the object's icon, given in OS units
y	the y-coordinate at which to pin the object's icon, given in OS units

Use

*Pin adds a file, application or directory to the desktop pinboard, positioning its icon at the given coordinates.

Example

```
*Pin adfs::MHardy.$.!System 200 200
```

Related commands

*AddTinyDir

Related SWIs

None

Related vectors

None

*Pinboard

Starts the pinboard

Syntax

```
*Pinboard
```

Parameters

None

Use

*Pinboard starts the pinboard. This command is provided for internal use only, so that the Task Manager can start the Pinboard when it starts the desktop.

Related commands

None

Related SWIs

None

Related vectors

None

*RemoveTinyDir

Removes a file, application or directory icon from the icon bar

Syntax

```
*RemoveTinyDir [object]
```

Parameters

object	a valid pathname specifying a file, application or directory

Use

*RemoveTinyDir removes a file, application or directory icon that was previously placed on the icon bar by a *AddTinyDir command. If no pathname is given, all such icons are removed from the icon bar.

Example

```
*RemoveTinyDir adfs::MHardy.$.!System
```

Related commands

*AddTinyDir, *Pin

Related SWIs

None

Related vectors

None

49 The Filter Manager

Introduction and Overview

SWI calls

Filter_RegisterPreFilter (SWI &42640)

Adds a new pre filter to the list of pre filters

On entry

R0 = pointer to 0 terminated filter name
R1 = address of filter
R2 = value to be passed in R12
R3 = task handle of task to which filter is applied (or 0 for all tasks)

On exit

All registers preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The address pointed to by R1 will be called whenever the task who's handle is passed in R3 calls Wimp_Poll with R12 equal to the value of R2 when this SWI is called.

The routine pointed to by R1 will be called with:

On entry

R0 = event mask as passed to Wimp_Poll
R1 = pointer to Event block as passed to Wimp_Poll
R2 = task handle of task that called Wimp_Poll

On Exit

It may clear bits in R0 to provide a new event mask
It must preserve all other registers

Related SWIs

Filter_DeRegisterPreFilter (SWI &42642)

Related vectors

None

Filter_RegisterPostFilter (SWI &42641)

Adds a new post filter to the list of post filters

On entry

R0 = pointer to 0 terminated filter name
R1 = address of filter
R2 = value to be passed in R12
R3 = task handle of task to which filter is applied (or 0 for all tasks)
R4 = event mask (1 bit masks the event out as for Wimp_Poll)

On exit

All registers preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The address pointed to by R1 will be called whenever the Wimp is about to return to the task who's handle is passed in R3 from Wimp_Poll with R12 equal to the value of R2 when this SWI is called.

On entry to the routine pointed to by R1:

On entry

R0 = event reason code (as from Wimp Poll)
R1 = pointer to User's event buffer
R2 = task handle for task to which the return is made
Task paged in

On Exit

The routine may modify the reason code in R0 and the contents of the buffer pointed to by R1 to provide a new event.

It must preserve R1 and R2.

Related SWIs

Filter_DeRegisterPostFilter (SWI &42643)

Related vectors

None

Filter_DeRegisterPreFilter (SWI &42642)

Removes a pre filter from the list of pre filters

On Entry

R0 = pointer to 0 terminated filter name
R1 = address of filter
R2 = value to be passed in R12
R3 = task handle of task to which filter was applied

All must be the same as those passed to Filter_RegisterPreFilter

On exit

All registers preserved
Filter deregistered

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI removes a pre filter from the list of pre filters.

Related SWIs

Filter_RegisterPreFilter (SWI &42640)

Related vectors

None

Filter_DeRegisterPostFilter (SWI &42643)

Removes a pre filter from the list of pre filters

On entry

R0 = pointer to 0 terminated filter name
R1 = address of filter
R2 = value to be passed in R12
R3 = task handle of task to which filter was applied

All must be the same as those passed to Filter_RegisterPreFilter

On exit

All registers preserved
Filter deregistered

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI removes a pre filter from the list of pre filters.

Related SWIs

Filter_RegisterPostFilter (SWI &42641)

Related vectors

None

* Commands

*Filters

Lists all currently active pre- and post-Wimp_Poll filters

Syntax

```
*Filters
```

Parameters

None

Use

*Filters lists all currently active pre- and post-Wimp_Poll filters.

Example

```
*Filters

Filters called on entry to Wimp_Poll:
Filter          Task

Usage           All tasks


Filters called on exit from Wimp_Poll:
Filter          Task            Mask

Usage           All tasks       00000000
```

Related commands

None

Related SWIs

None

Related vectors

None

50 The TaskManager module

Introduction and Overview

TaskManager_TaskNameFromHandle (SWI &42680)

Finds the name of a task

On entry

R0 = Task Handle

On exit

R0 = pointer to Task name

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI enables you to find the name of a task given its task handle.
You must copy the name into your workspace if you want to keep it.

Related SWIs

None

Related vectors

None

TaskManager_EnumerateTasks (SWI &42681)

Enumerates all the currently active tasks

On entry

R0 = 0 for first call or value from last call
R1 = pointer to Word aligned Buffer
R2 = Buffer length

On exit

R0 = <0 if no more entries, else the value to pass to next call
R1 = pointer to first unused word in buffer
R2 = number of unused bytes in buffer
[R1] - filled with entries of the form:
[0] = Task handle
[4] = pointer to Task name (should be copied away and not used in place)
[8] = amount of memory (in K) used by the task
[12] = flags:
bit 0 1 module task
0 application task
bit 1 1 slot bar can be dragged
0 slot bar cannot be dragged
bits 2-31 reserved and are currently 0

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI provides a way of enumerating all the currently active tasks.

Related SWIs

None

Related vectors

None

* Commands

*ChangeDynamicArea

Changes the size of the font cache, system sprite area and/or RAM disc

Syntax

```
*ChangeDynamicArea [-FontSize n[K]] [-SpriteSize n[K]] [-RamFsSize n[K]]
```

Parameters

n	Size of the area to be set, in kilobytes

Use

*ChangeDynamicArea changes the size of the font cache, system sprite area and/or RAM disc. Its main use is in desktop boot files.

Example

```
*ChangeDynamicArea -SpriteSize 32K -RamFsSize 100K
```

Related commands

None

Related SWIs

OS_ChangeDynamicArea (SWI &2A)

Related vectors

None

51 TaskWindow

Introduction and Overview

The TaskWindow module is intended to allow programs which do not call SWI Wimp_Poll to be pre-emptively scheduled in the RISC OS desktop. In the following sections Child refers to the task created from a call to *TaskWindow and Parent refers to the task being used to display the Child's output. Any screen output produced by the Child is intercepted and sent in Wimp messages to the Parent.

SWI calls

TaskWindow_TaskInfo (&43380)

Obtains information from the TaskWindow module

On entry

R0 = Information index

On exit

Registers values depend on value of R0 on entry (see below)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI is used to obtain information from the TaskWindow module which is not readily available otherwise. The value in R0 on entry indicates which item of information is required. The registers on exit return the requested information.

Values of R0 are:

On entry	On exit
0	R0 is non-zero if the calling task is running in a task window otherwise it returns 0

Related SWIs

Wimp_ReadSysInfo (SWI &400F2)
with R0 = 3 on entry

Related vectors

None

Messages

TaskWindow_Input (&808C0)

This message is used to send input data from Parent to Child.

R1+20 size of input data
R1+24 pointer to input data

Input can also be sent via a normal RAM transfer protocol, i.e. send a Message_DataSave, then perform the following two steps until all the data has been sent and received:

1 wait for Message_RAMFetch

2 send back Message_RAMTransmit

See the section entitled *Memory data transfer* on page 4-310 for a full description of this protocol.

TaskWindow_Output (&808C1)

This message is sent to the Parent when one of its children has produced output.

R1+20 size of output data
R1+24... output data

TaskWindow_Ego (&808C2)

This message is sent to the Parent, to inform him of the Child's task-id.

R1+4 Child's task-id (as filled in by Wimp)
R1+20 Parent's txt-handle (as passed to *TaskWindow or *ShellCLI_Task)

Note that this is the only time the txt-handle is used. It allows the Parent to identify which Child is announcing its task-id.

TaskWindow_Morio (&808C3)

This message is sent to the Parent when the Child exits.

No data (all necessary information is in the wimp message header).

TaskWindow_Morite (&808C4)

This message is sent by the Parent to kill the Child.

No data (all necessary information is in the wimp message header).

TaskWindow_NewTask (&808C5)

This message is broadcast by an external task which requires an application (e.g. Edit) to start up a task window. If the receiving application wishes to deal with this request, it should first acknowledge the Wimp message, then issue a SWI Wimp_StartTask with R1+20... as the command.

R1+20... the command to run

TaskWindow_Suspend (&808C6)

This message is sent by the Parent to suspend a Child.

No data (all necessary information is in the wimp message header).

TaskWindow_Resume (&808C7)

This message is sent by the Parent to resume a suspended Child.

No data (all necessary information is in the wimp message header).

* Commands

*ShellCLI_Task

Runs an application in a window

Syntax

`*ShellCLI_Task` *xxxxxxxx xxxxxxxx*

Parameters

xxxxxxxx	an 8 digit hex. number giving the task handle of the parent task
xxxxxxxx	an 8 digit hex. number giving a handle which may be used by the parent task to identify the task

Use

*ShellCLI_Task runs an application in a window. This command is intended for use only within desktop applications.

Use of this command is deprecated. Its functionality is subsumed within *TaskWindow.

Related commands

*ShellCLI_TaskQuit

Related SWIs

None

Related vectors

None

*ShellCLI_TaskQuit

Quits the current task window

Syntax

`*ShellCLI_TaskQuit`

Parameters

None

Use

*ShellCLI_TaskQuit quits the current task window. This command is intended for use only within desktop applications.

Related commands

*ShellCLI_Task

Related SWIs

None

Related vectors

None

*TaskWindow

Starts a background task, which will obtain a task window if necessary

Syntax

```
*TaskWindow [command] [[-wimpslot] nK] [[-name] taskname] [-ctrl]
            [-display] [-quit] [-task &xxxxxxxx] [-txt &xxxxxxxx]
```

Parameters

command	command to execute as a background task
n	size of memory to allocate to task
taskname	name of task
-ctrl	allow control characters through
-display	open the task window immediately, rather than waiting for a character to be printed
-quit	make that task quit after the command, even if the task window has been opened
-task &xxxxxxxx	an eight digit hex. number giving the Wimp Task-id of the calling task
-txt &xxxxxxxx	an eight digit hex. number giving the handle for the Parent to identify the Child by

Use

*TaskWindow starts a background task, which will obtain a task window if it needs to get input, or to output a character to the screen.

Any fields comprising more than one word must be enclosed in double quotes.

This command can only be issued via a SWI Wimp_StartTask or a *WimpTask command.

If -txt and -task are not used, then before starting the task, a TaskWindow_NewTask message is broadcast to find an application (e.g. Edit) that can provide a window in which to show the task's output. An application task which receives this broadcast, and which wishes to receive output from the task, should acknowledge the message and then SWI Wimp_StartTask the command given in the message block.

Example

```
*TaskWindow "Cat Ram:$" -ctrl -display -quit
```


Related commands

None

Related SWIs

None

Related vectors

None

52 ShellCLI

Introduction

This module provides a single * Command that allows you to invoke a command shell from a Wimp program.

It also has two SWIs for its own internal use. **You must not use them** in your own code.

SWI Calls

Shell_Create (SWI &405C0)

This SWI call is for use by the ShellCLI module only. **You must not use it** in your own code.

Shell_Destroy (SWI &405C1

This SWI call is for use by the ShellCLI module only. **You must not use it** in your own code

* Commands

*ShellCLI

Invokes a command shell from a Wimp program

Syntax

```
*ShellCLI
```

Parameters

None

Use

*ShellCLI invokes a command shell from a Wimp program, starting it as a Wimp task. It prompts the user with *, and passes each line that the user types to the command line interpreter, OS_CLI. This is repeated until the user enters a blank line, whereupon control is returned to the Wimp program. The Task Manager uses this command to implement its *Command (F12) menu item.

You must call *ShellCLI using *WimpTask or the SWI Wimp_StartTask, rather than using the command line or the SWI OS_CLI. You can only call Wimp_StartTask or *WimpTask from within an active task.

The command uses the two SWIs Shell_Create and Shell_Destroy; it is the only user of these SWIs.

Example

```
*WimpTask ShellCLI
```

Related commands

None

Related SWIs

Shell_Create, Shell_Destroy

Related vectors

None

53 !Configure

Part 5 – System extensions

54 ColourTrans

Introduction

ColourTrans allows a program to select the physical red, green and blue colours that it wishes to use, given a particular output device and palette. ColourTrans then calculates the best colour available to fit the required colour.

Thus, an application doesn't have to be aware of the number of colours available in a given mode.

It can also intelligently handle colour usage with sprites and the font manager, and is the best way to set up colours when printing.

Finally, it supports colour calibration, so that you can make different output devices produce the same colours. (This feature is not supported by RISC OS 2.0)

Before reading this chapter, you should be familiar with the VDU, sprite and font manager principles.

Overview

The ColourTrans module is provided in RAM in Release 2.0 of RISC OS in System:Modules.Colours, but is in the ROM for later releases of the OS. Any application which uses it should ensure it is present using the *RMEnsure command, say from an Obey file:

```
RMEnsure ColourTrans 0.51 RMLoad System:Modules.Colours
RMEnsure ColourTrans 0.51 Error You need ColourTrans 0.51 or later
```

Definition of terms

Here are some terms you should know when using this chapter.

GCOL is like the colour parameter passed to VDU 17. It uses a simple format for 256 colour modes.

Colour number is what is written into screen memory to achieve a given colour in a particular mode.

Palette entry is a word that contains a description of a physical colour in red, green and blue levels. Usually, this term refers to the required colour that is passed to a ColourTrans SWI.

Palette pointer is a pointer to a list of palette entries. The table would have one entry for each logical colour in the requested mode. In 256 colour mode, only 16 entries are needed, as there are only 16 palette registers.

Closest colour is the colour in the palette that most closely matches the palette entry passed. Furthest colour is the one furthest from the colour requested. These terms refer to a least-squares test of closeness.

Finding a colour

There are many SWIs that will find the best fit colour in the palette for a set of parameters. Here is a list of the different kinds of parameters that can return a best fit colour:

- Given palette entry, return nearest or furthest GCOL
- Given palette entry, return nearest or furthest colour number
- Given palette entry, mode and palette pointer, return nearest or furthest GCOL
- Given palette entry, mode and palette pointer, return nearest or furthest colour number

Setting a colour

Some SWIs will set the VDU driver GCOL to the calculated GCOL after finding it.

- Given palette entry, return nearest GCOL, and set that colour
- Given palette entry, return furthest GCOL, and set that colour

Conversion

There is a pair of SWIs to convert GCOLs to and from colour numbers. Note that this only has meaning for 256 colour modes.

Sprites and Fonts

The colour control commands in sprites and the font manager can be controlled from ColourTrans. Thus, the selection of logical colours within these modules is handled by ColourTrans, rather than an application selecting an explicit range.

Using other palette SWIs

If an application has to use other SWIs to change the palette, then there is a SWI in ColourTrans to inform it. This is because ColourTrans maintains a cache used for mapping colours. If the palette has independently changed, then it has no way of telling.

If the screen mode has changed there is no need to use this call, since the ColourTrans module detects this itself – but, under RISC OS 2, if output is switched to a sprite (and ColourTrans will be used) then the SWI must also be called.

Wimp

If you are using the Wimp interface, then the ColourTrans calls are fine to use, because they never modify the palette.

Printing

Because ColourTrans allows an application to request an RGB colour rather than a logical colour, it is ideal for use with the printer drivers, where a printer may be able to represent some RGB colours more accurately then the screen.

Colour calibration

There is a major problem in working with colour documents. This is that, if the user selects some colours on the screen, they may well come out as different colours on a printer or other final output device. Colour calibration is a way to get round this problem.

Colour calibration involves calibrating the screen colours with a fixed standard set of colours, and also calibrating the output device colours to the same fixed set of colours. Then, when an application draws to the screen, it does so in standard colours which are converted by the OS to screen colours. If the application draws to the printer it again does so in standard colours, but this time they are converted to printer colours.

So, for the user, calibrating the colours will give constant colour reproduction throughout the system, for the cost of calibrating the devices in the first place.

Colour calibration is not available in RISC OS 2.0.

Technical Details

Colours

Two different colour systems are used in 256 colour modes. The GCOL form is much easier to use, while the colour number is optimised for the hardware. In all other colour modes, they are identical.

The palette entry used to request a given physical colour is in the same format as that used to set the anti-alias palette in the font manager.

GCOL

The 256 colour modes use a byte that looks like this:

Bit	Meaning
0	Tint bit 0 (red+green+blue bit 0)
1	Tint bit 1 (red+green+blue bit 1)
2	Red bit 2
3	Red bit 3 (high)
4	Green bit 2
5	Green bit 3 (high)
6	Blue bit 2
7	Blue bit 3 (high)

This format is converted into the internal 'colour number' format when stored, because that is what the VIDC hardware recognises.

Colour number

The 256 colour mode in the colour number looks like this:

Bit	Meaning
0	Tint bit 0 (red+green+blue bit 0)
1	Tint bit 1 (red+green+blue bit 1)
2	Red bit 2
3	Blue bit 2
4	Red bit 3 (high)
5	Green bit 2
6	Green bit 3 (high)
7	Blue bit 3 (high)

In fact the bottom 4 bits of the colour number are obtained via the palette, but the default palette in 256 colour modes is set up so that the above settings apply, and this is not normally altered.

Palette entry

The palette entry is a word of the form &BBGGRR00. That is, it consists of four bytes, with the palette value for the blue, green and red gun in the top three bytes. Bright white, for instance would be &FFFFFF00, while half intensity cyan would be &77770000. The current graphics hardware only uses the upper nibbles of these colours, but for upwards compatibility the lower nibble should contain a copy of the upper nibble.

Finding a colour

The SWIs that find the best fit have generally self explanatory names. As shown in the overview, they follow a standard pattern. All of the SWI names that follow are prefixed with ColourTrans_. They are as follows:

ReturnGCOL (SWI &40742)
Given palette entry, return nearest GCOL

ReturnOppGCOL (SWI &40747)
Given palette entry, return furthest GCOL

ReturnColourNumber (SWI &40744)
Given palette entry, return nearest colour number

ReturnOppColourNumber (SWI &40749)
Given palette entry, return furthest colour number

ReturnGCOLForMode (SWI &40745)
Given palette entry, mode and palette pointer, return nearest GCOL

ReturnOppGCOLForMode (SWI &4074A)
Given palette entry, mode and palette pointer, return furthest GCOL

ReturnColourNumberForMode (SWI &40746)
Given palette entry, mode and palette pointer, return nearest colour number

ReturnOppColourNumberForMode (SWI &4074B)
Given palette entry, mode and palette pointer, return furthest colour number

Palette pointers

Where a palette pointer is used, certain conventions apply:

- a palette pointer of –1 means the current palette is used
- a palette pointer of 0 means the default palette for the specified mode.

Where modes are used:

- mode –1 means the current mode.

Best fit colour

These calls use a simple algorithm to find the colour in the palette that most closely matches the high resolution colour specified in the palette entry. It calculates the *distance* between the colours, which is a weighted least squares function. If the desired colour is (R_d, B_d, G_d) and a trial colour is (R_t, B_t, G_t), then:

$$\text{distance} = \text{redweight} \times (R_t - R_d)^2 + \text{greenweight} \times (G_t - G_d)^2 + \text{blueweight} \times (B_t - B_d)^2$$

where redweight = 2, greenweight = 3 and blueweight = 1. These weights are set for the most visually effective solution to this problem.

Setting a colour

ColourTrans_SetGCOL (SWI &40743) will act like ColourTrans_ ReturnGCOL, except that it will set the graphics system GCOL to be as close to the colour you requested as it can. Note that ECF patterns will not yet be used in monochrome modes to reflect grey shades, as they are with Wimp_SetColour.

Similarly, ColourTrans_SetOppGCOL (SWI &40748) will set the graphics system GCOL with the opposite of the palette entry passed.

Conversion

To convert between the GCOL and colour number format in 256 colour modes, the SWIs ColourTrans_GCOLToColourNumber (SWI &4074C) and ColourTrans_ColourNumberToGCOL (SWI &4074D) can be used.

Sprites and Fonts

ColourTrans_SelectTable (SWI &40740) will set up a translation table in the buffer. ColourTrans_SelectGCOLTable (SWI &40741) will set up a list of GCOLs in the buffer. See the section entitled *Pixel translation table* on page 2-252 for a definition of these tables (although the latter call does not in fact relate to sprites).

ColourTrans_ReturnFontColours (SWI &4074E) will try and find the best set of logical colours for an anti-alias colour range. ColourTrans_SetFontColours (SWI &4074F) also does this, but sets the font manager plotting colours as well. It calls Font_SetFontColours, or Font_SetPalette in 256 colour modes – but it works out which logical colours to use beforehand. See the section entitled *Colours* on page 5-3 for details of using colours and anti-aliasing colours; see also the descriptions of the relevant commands later in the same chapter, on page 5-48 and page 5-50.

Using other palette SWIs

If a program has changed the palette, then ColourTrans_InvalidateCache (SWI &40750) must be called. This will reset its internal cache. This applies to Font_SetFontColours or Wimp_SetPalette or VDU 19 or anything like that, but not to mode change, since this is detected automatically.

Under RISC OS 2 you must also call this SWI if output has been switched to a sprite, and ColourTrans is to be called while the output is so redirected. You must then call it again after output is directed back to the screen. Later versions of RISC OS automatically do this for you.

Colour calibration

Colour calibration is performed by ColourTrans using a calibration table that maps from device colours to standard colours.

The palette in RISC OS maps logical colours to device colours (also known as physical colours). When you ask RISC OS to select a colour for you, it takes this palette and uses a calibration table to convert the device colours to standard colours, giving a (transient) palette that maps logical colours to standard colours. It then chooses the closest standard colour to the one that you have specified.

Calibration tables

A calibration table is a one-to-one map that fills the device colour space, but does not necessarily fill the standard colour space. In fact, it consists of three separate mappings: one for each component of the device space (Red, Green and Blue on a monitor, for example). Each mapping consists of a series of device component/standard colour pairs.

The pairs are stored as 32-bit words, in the form &BBGGRRDD, where DD is the amount of the device component (from 0 to 255), and BBGGRR is the standard colour corresponding to that amount. The two other device components are presumed to be zero.

The format of the table is:

Word	Meaning
0	Number of pairs of component 1 (*n*1)
1	Number of pairs of component 2 (*n*2)
2	Number of pairs of component 3 (*n*3)
3	*n*1 words giving pairs for component 1
3 + *n*1	*n*2 words giving pairs for component 2
3 + *n*1 + *n*2	*n*3 words giving pairs for component 3

The length of the table is therefore 3 + *n*1 + *n*2 + *n*3 words.

Within each of the three sets of mappings, the words must be sorted in ascending order of device component. To fill the device colour space, there must be entries for device components of 0 and 255, so there must be at least two pairs for each component.

As an example, a minimal calibration table might be:

Word	Meaning
&00000002	2 pairs of red component
&00000002	2 pairs of green component
&00000002	2 pairs of blue component
&02010300	Device colour 000000 corresponds to standard colour 020103
&0203FDFF	Device colour 0000FF corresponds to standard colour 0203FD
&02010300	Device colour 000000 corresponds to standard colour 020103
&03FC02FF	Device colour 00FF00 corresponds to standard colour 03FC02
&02010300	Device colour 000000 corresponds to standard colour 020103
&FF0302FF	Device colour FF0000 corresponds to standard colour FF0302
	(In this column both device and standard colours are given in the format &BBGGRR)

The default mapping for the screen is that device colours and standard colours are the same. This produces the same effect as earlier uncalibrated versions of ColourTrans.

To convert a specific device colour to a standard colour, ColourTrans splits the device colour into its three component parts. Then, for each component, it uses linear interpolation between the two device components 'surrounding' the required device component. The standard colours thus obtained for each component are then summed to give the final calibrated standard colour.

Colour calibration is not available in RISC OS 2.0.

Service Calls

Service_CalibrationChanged (Service Call &5B)

Screen calibration is changed

On entry

R1 = &5B (reason code)

On exit

All registers preserved

This service call should not be claimed

Use

This service is issued by the ColourTrans module when the ClourTrans_SetCalibration SWI has been issued.

It is noticed by the Palette utility in the desktop, which broadcasts a Message_PaletteChange.

Service_InvalidateCache (Service Call &82)

Broadcast whenever the cache is flushed within ColourTrans

On entry

R1 = &82 (reason code)

On exit

All registers preserved

Use

This service is broadcast whenever the cache is flushed within ColourTrans. You should never claim it.

SWI Calls

ColourTrans_SelectTable (SWI &40740)

Sets up a translation table in a buffer

On entry

R0 = source mode or (if ≥ 256) pointer to sprite area
R1 = source palette pointer or (if R0 ≥ 256) pointer to sprite name/sprite in area pointed to by R0 (as specified by R5)
R2 = destination mode, or –1 for current mode
R3 = destination palette pointer, or –1 for current palette, or 0 for default for the mode
R4 = pointer to buffer
R5 = flags (used if R0 ≥ 256):
bit 0 set ⇒ R1 = pointer to sprite; else R1 = pointer to sprite name
bit 1 set ⇒ use current palette if sprite doesn't have one; else use default
all other bits reserved (must be zero)

On exit

R0 - R4 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets up a translation table in a buffer – that is, a set of colour numbers as used by scaled sprite plotting. You may specify the source mode palette either directly, or (except in RISC OS 2) by specifying a sprite. See the section entitled *Pixel translation table* on page 2-252 for details of such tables.

You should use this call rather than any other to set up translation tables for sprites, as it copes correctly with sprites that have a 256 colour palette.

In RISC OS 2, R0 must be less than 256, and so R5 is unused. Consequently, to use a sprite as the source you first have to copy its palette information out from its header.

Related SWIs

None

Related vectors

ColourV

ColourTrans_SelectGCOLTable (SWI &40741)

Sets up a list of GCOLs in a buffer

On entry

R0 = source mode or (if ≥ 256) pointer to sprite area
R1 = source palette pointer or (if R0 ≥ 256) pointer to sprite name/sprite in area pointed to by R0 (as specified by R5)
R2 = destination mode, or –1 for current mode
R3 = destination palette pointer, or –1 for current palette, or 0 for default for the mode
R4 = pointer to buffer
R5 = flags (used if R0 ≥ 256):
bit 0 set ⇒ R1 = pointer to sprite; else R1 = pointer to sprite name
bit 1 set ⇒ use current palette if sprite doesn't have one; else use default
all other bits reserved (must be zero)

On exit

R0 - R4 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrant

SWI is not re-entrant

Use

This call, given a source mode and palette (either directly, or – except in RISC OS 2 – from a sprite), a destination mode and palette, and a buffer, sets up a list of GCOLs in the buffer. The values can subsequently be used by passing them to GCOL and Tint.

In RISC OS 2, R0 must be less than 256, and so R5 is unused. Consequently, to use a sprite as the source you first have to copy its palette information out from its header.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ReturnGCOL (SWI &40742)

Gets the closest GCOL for a palette entry

On entry

R0 = palette entry

On exit

R0 = GCOL

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, given a palette entry, returns the closest GCOL in the current mode and palette.

It is equivalent to ColourTrans_ReturnGCOLForMode for the given palette entry, with parameters of -1 for both the mode and palette pointer.

The colours are not calibrated in RISC 2.0, but are calibrated in later versions.

Related SWIs

ColourTrans_ReturnGCOLForMode (SWI &40745)

Related vectors

ColourV

ColourTrans_SetGCOL (SWI &40743)

Sets the closest GCOL for a palette entry

On entry

R0 = palette entry
R3 = flags
R4 = GCOL action

On exit

R0 = GCOL
R2 = $\log_2$ of bits-per-pixel for current mode
R3 = initial value AND &80
R4 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, works out the closest GCOL in the current mode and palette, and sets it. The flags in R3 have the following meaning:

Value of R3	Meaning
bit 7 = 1	set background colour
bit 7 = 0	set foreground colour
bit 8 = 1	use ECFs to give a better approximation to the colour
bit 8 = 0	don't use ECFs

The remaining bits of R3 and the top three bytes of R4 are reserved, and should be set to zero to allow for future expansion. Bit 8 of R3 is ignored in RISC OS 2.0, which does not support ECF patterns with this call.

The colours are not calibrated in RISC 2.0, but are calibrated in later versions.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ReturnColourNumber (SWI &40744)

Gets the closest colour for a palette entry

On entry

R0 = palette entry

On exit

R0 = colour number

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, returns the closest colour number in the current mode and palette.

The colours are not calibrated in RISC 2.0, but are calibrated in later versions.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ReturnGCOLForMode (SWI &40745)

Gets the closest GCOL for a palette entry

On entry

R0 = palette entry
R1 = destination mode, or -1 for current mode
R2 = palette pointer, or -1 for current palette, or 0 for default for the mode

On exit

R0 = GCOL
R1 = preserved
R2 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, a destination mode and palette, returns the closest GCOL.

The colours are not calibrated in RISC 2.0, but are calibrated in later versions.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ReturnColourNumberForMode (SWI &40746)

Gets the closest colour for a palette entry

On entry

R0 = palette entry
R1 = destination mode, or -1 for current mode
R2 = palette pointer, or -1 for current palette, or 0 for default for the mode

On exit

R0 = colour number
R1 = preserved
R2 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, a destination mode and palette, returns the closest colour number.

The colours are not calibrated in RISC 2.0, but are calibrated in later versions.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ReturnOppGCOL (SWI &40747)

Gets the furthest GCOL for a palette entry

On entry

R0 = palette entry

On exit

R0 = GCOL

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, given a palette entry, returns the furthest GCOL in the current mode and palette.

It is equivalent to ColourTrans_ReturnOppGCOLForMode for the given palette entry, with parameters of -1 for both the mode and palette pointer.

The colours are not calibrated in RISC 2.0, but are calibrated in later versions.

Related SWIs

None

Related vectors

ColourV

ColourTrans_SetOppGCOL (SWI &40748)

Sets the furthest GCOL for a palette entry

On entry

R0 = palette entry
R3 = 0 for foreground or 128 for background
R4 = GCOL action

On exit

R0 = GCOL
R2 = $\log_2$ of bits-per-pixel for current mode
R3 = initial value AND &80
R4 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, works out the furthest GCOL in the current mode and palette, and sets it

The top three bytes of R3 and R4 should be zero, to allow for future expansion.

The colours are not calibrated in RISC 2.0, but are calibrated in later versions.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ReturnOppColourNumber (SWI &40749)

Gets the furthest colour for a palette entry

On entry

R0 = palette entry

On exit

R0 = colour number

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, returns the furthest colour number in the current mode and palette.

The colours are not calibrated in RISC 2.0, but are calibrated in later versions.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ReturnOppGCOLForMode (SWI &4074A)

Gets the furthest GCOL for a palette entry

On entry

R0 = palette entry
R1 = destination mode or -1 for current mode
R2 = palette pointer, -1 for current palette or 0 for default for the mode

On exit

R0 = GCOL
R1 = preserved
R2 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, a destination mode and palette, returns the furthest GCOL.

The colours are not calibrated in RISC 2.0, but are calibrated in later versions.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ReturnOppColourNumberForMode (SWI &4074B)

Gets the furthest colour for a palette entry

On entry

R0 = palette entry
R1 = destination mode or -1 for current mode
R2 = palette pointer, -1 for current palette or 0 for default for the mode

On exit

R0 = colour number
R1 = preserved
R2 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, a destination mode and palette, returns the furthest colour number.

The colours are not calibrated in RISC 2.0, but are calibrated in later versions.

Related SWIs

None

Related vectors

ColourV

ColourTrans_GCOLToColourNumber (SWI &4074C)

Translates a GCOL to a colour number

On entry

R0 = GCOL

On exit

R0 = colour number

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call changes the value passed from a GCOL to a colour number.

You should only call this SWI for 256 colour modes; the results will be meaningless for any others.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ColourNumberToGCOL (SWI &4074D)

Translates a colour number to a GCOL

On entry

R0 = colour number

On exit

R0 = GCOL

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 changes the value passed from a colour number to a GCOL.

You should only call this SWI for 256 colour modes; the results will be meaningless for any others.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ReturnFontColours (SWI &4074E)

Finds the best range of anti-alias colours to match a pair of palette entries

On entry

R0 = font handle, or 0 for the current font
R1 = background palette entry
R2 = foreground palette entry
R3 = maximum foreground colour offset (0 - 14)

On exit

R0 = preserved
R1 = background logical colour (preserved if in 256 colour mode)
R2 = foreground logical colour
R3 = maximum sensible colour offset (up to R3 on entry)

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, given background and foreground colours and the number of anti-aliasing colours desired, finds the maximum range of colours that can sensibly be used. So for the given pair of palette entries, it finds the best fit in the current palette, and then inspects the other available colours to deduce the maximum possible amount of anti-aliasing up to the limit in R3.

If anti-aliasing is desirable, you should set R3 = 14 on entry; otherwise set R3 = 0 for monochrome.

The values in R1 - R3 on exit are suitable for passing to Font_SetFontColours, or including in a font string in a command (18) sequence.

Note that in 256 colour modes, you can only set 16 colours before previously returned information becomes invalid. Therefore, if you are using this SWI to obtain information to subsequently pass to the font manager, do not use more than 16 colours.

Also note that in 256 colour modes, the font manager's internal palette will be set, with all 16 entries being cycled through by ColourTrans.

The colours are not calibrated in RISC 2.0, but are calibrated in later versions.

See page 5-48 of the chapter entitled *The Font Manager* for further details of the parameters used in this call.

Related SWIs

ColourTrans_SetFontColours (SWI &4074F),
Font_SetFontColours (SWI &40092)

Related vectors

ColourV

ColourTrans_SetFontColours (SWI &4074F)

Sets the best range of anti-alias colours to match a pair of palette entries

On entry

R0 = font handle, or 0 for the current font
R1 = background palette entry
R2 = foreground palette entry
R3 = maximum foreground colour offset (0 - 14)

On exit

R0 preserved
R1 = background logical colour (preserved if in 256 colour mode)
R2 = foreground logical colour
R3 = maximum sensible colour offset (up to R3 on entry)

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, given a pair of palette entries, finds the best available range of anti-alias colours in the current palette, and sets the font manager to use these colours.

The colours are not calibrated in RISC 2.0, but are calibrated in later versions.

Related SWIs

ColourTrans_ReturnFontColours (SWI &4074F)

Related vectors

ColourV

ColourTrans_InvalidateCache (SWI &40750)

Informs ColourTrans that the palette has been changed by some other means

On entry

—

On exit

—

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call must be issued whenever the palette has changed since ColourTrans was last called. Note that colour changes due to a mode change are detected. You only need to use this if another of the palette change operations was used.

Under RISC OS 2 you must also call this SWI if output has been switched to a sprite, and ColourTrans is to be called while the output is so redirected. You must then call it again after output is directed back to the screen. For example, the palette utility on the icon bar calls this SWI when you finish dragging one of the RGB slider bars. Later versions of RISC OS automatically do this for you.

Related SWIs

None

Related vectors

ColourV

ColourTrans_SetCalibration (SWI &40751)

Sets the calibration table for the screen

On entry

R0 = pointer to calibration table

On exit

—

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call copies the calibration table pointed to by R0 into the RMA as the new calibration table for the screen. If the call fails due to lack of room in the RMA then the calibration will be set to the default calibration for the screen, and the 'No room in RMA' error will be passed back. Another possible error is 'Bad calibration table', given if the device component pairs do not cover the full range 00 to &FF.

This call is not available in RISC OS 2.0.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ReadCalibration (SWI &40752)

Reads the calibration table for the screen

On entry

R0 = 0 to read size of table, or pointer to buffer

On exit

R0 preserved
R1 = size of table (if R0 = 0 on entry)

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 the calibration table for the screen into the buffer pointed to by R0, which should be large enough to contain the complete table. Ideally you should first issue this call with R0=0 to read the size of the table, then allocate space, and then issue this call again to read the table.

This call is not available in RISC OS 2.0.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ConvertDeviceColour (SWI &40753)

Converts a device colour to a standard colour

On entry

R1 = 24-bit device colour (&BBGGRR00 for the screen)
R3 = 0 to use the current screen calibration, or pointer to calibration table to use

On exit

R2 = 24-bit standard colour (&BBGGRR00)

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 allows applications to read, say, screen colours, and find the standard colours to which they correspond.

This call is not available in RISC OS 2.0.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ConvertDevicePalette (SWI &40754)

Converts a device palette to standard colours

On entry

R0 = number of colours to convert
R1 = pointer to table of 24-bit device colours
R2 = pointer to table to store standard colours
R3 = 0 to use the current screen calibration, or pointer to calibration table to use

On exit

R0 - R3 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call allows printer drivers to use the same calibration calculation code for their conversions between device and standard colours as the screen does. The printer device palette can be set up and then converted using this call to the standard colours using the printer's calibration table. This call is mainly provided to ease the load on the writers of printer drivers.

This call is not available in RISC OS 2.0.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ConvertRGBToCIE (SWI &40755)

Converts RISC OS RGB colours to industry standard CIE colours

On entry

R0 = red component
R1 = green component
R2 = blue component

On exit

R0 = CIE X tristimulus value
R1 = CIE Y tristimulus value
R2 = CIE Z tristimulus value

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 converts RISC OS RGB colours to industry standard CIE colours, allowing easy interchange with other systems. The CIE standard that is output is the XYZ tristimulus values.

All parameters are passed as fixed point 32 bit numbers in the range 0 - 1, with 16 bits below the point and 16 bits above the point.

This call is not available in RISC OS 2.0.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ConvertCIEToRGB (SWI &40756)

Converts industry standard CIE colours to RISC OS RGB colours

On entry

R0 = CIE X tristimulus value
R1 = CIE Y tristimulus value
R2 = CIE Z tristimulus value

On exit

R0 = red component
R1 = green component
R2 = blue component

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 converts industry standard CIE colours to RISC OS RGB colours, allowing easy interchange with other systems. The CIE standard that is accepted is the XYZ tristimulus values.

All parameters are passed as fixed point 32 bit numbers in the range 0 - 1, with 16 bits below the point and 16 bits above the point.

This call is not available in RISC OS 2.0.

Related SWIs

None

Related vectors

ColourV

ColourTrans_WriteCalibrationToFile (SWI &40757)

Saves the current calibration to a file

On entry

R0 = flags
R1 = file handle of file to save calibration to

On exit

R0 corrupted

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call saves the current calibration to a file. It does so by creating a list of * Commands which will recreate the current calibration.

If bit 0 of R0 is clear then the calibration will only be saved if it is not the default calibration. If bit 0 of R0 is set then the calibration will be saved even if it is the default calibration.

This call is not available in RISC OS 2.0.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ConvertRGBToHSV (SWI &40758)

Converts RISC OS RGB colours into corresponding hue, saturation and value

On entry

R0 = red component
R1 = green component
R2 = blue component

On exit

R0 = hue
R1 = saturation
R2 = value

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 converts RISC OS RGB colours into corresponding hue, saturation and value.

All parameters are passed as fixed point 32 bit numbers, with 16 bits below the point and 16 bits above the point. Hue ranges from 0 - 360 with no fractional element, whilst the remaining parameters are in the range 0 - 1 and may have fractional elements.

When dealing with achromatic colours, hue is undefined.

This call is not available in RISC OS 2.0.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ConvertHSVToRGB (SWI &40759)

Converts hue, saturation and value into corresponding RISC OS RGB colours

On entry

R0 = hue
R1 = saturation
R2 = value

On exit

R0 = red component
R1 = green component
R2 = blue component

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 converts hue, saturation and value into corresponding RISC OS RGB colours.

All parameters are passed as fixed point 32 bit numbers, with 16 bits below the point and 16 bits above the point. Hue ranges from 0 - 360 with no fractional element, whilst the remaining parameters are in the range 0 - 1 and may have fractional elements.

An error is generated if a hue value is defined when the other values indicate that the colour is achromatic.

This call is not available in RISC OS 2.0.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ConvertRGBToCMYK (SWI &4075A)

Converts RISC OS RGB colours into the CMYK model

On entry

R0 = red component
R1 = green component
R2 = blue component

On exit

R0 = cyan component
R1 = magenta component
R2 = yellow component
R3 = key (black) component

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 converts RISC OS RGB colours into the CMY (cyan/magenta/yellow) model with a K (key – ie black) additive, allowing easy preparation of colour separations.

All parameters are passed as fixed point 32 bit numbers in the range 0 - 1, with 16 bits below the point and 16 bits above the point. The 'K' acts as a black additive and is a value equally subtracted or added to the given CMY values.

This call is not available in RISC OS 2.0.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ConvertCMYKToRGB (SWI &4075B)

Converts from the CMYK model to RISC OS RGB colours

On entry

R0 = cyan component
R1 = magenta component
R2 = yellow component
R3 = key (black) component

On exit

R0 = red component
R1 = green component
R2 = blue component

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 converts from the CMY (cyan/magenta/yellow) model with a K (key – ie black) additive to RISC OS RGB colours, allowing easy conversion from colour separations

All parameters are passed as fixed point 32 bit numbers in the range 0 - 1, with 16 bits below the point and 16 bits above the point. The 'K' acts as a black additive and is a value equally subtracted or added to the given CMY values.

This call is not available in RISC OS 2.0.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ReadPalette (SWI &4075C)

Reads either the screen's palette, or a sprite's palette

On entry

R0 = source mode or (if ≥ 256) pointer to sprite area
R1 = source palette pointer or (if R0 ≥ 256) pointer to sprite name/sprite in area pointed to by R0 (as specified by R4)
R2 = pointer to buffer, or 0 to return required size in R3
R3 = size of buffer (if R2 ≠ 0)
R4 = flags (used if R0 ≥ 256):
bit 0 set ⇒ R1 = pointer to sprite; else R1 = pointer to sprite name
bit 1 set ⇒ return flashing colours; else don't
all other bits reserved (must be zero)

On exit

R2 = pointer to next free word in buffer
R3 = remaining size of buffer

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 either the screen's palette, or a sprite's palette. It is the recommended way of doing so. It provides a way for applications to enquire about the palette and always read the absolute values, no matter what the hardware is capable of.

All palette entries are returned as **true** 24bit RGB, passing through the calibration if required. In 256 colour modes the palette is returned fully expanded (ie 256 palette entries, rather than the base 16 entries used by VIDC).

This call is not available in RISC OS 2.

Related SWIs

None

Related vectors

PaletteV

ColourTrans_WritePalette (SWI &4075D)

Writes to either the screen's palette, or to a sprite's palette

On entry

R0 = -1 to write current mode's palette, or pointer to sprite area
R1 = -1 to write current palette, else ignored (if R0 = -1); or (if R0 ≥ 0) pointer to sprite name/sprite in area pointed to by R0 (as specified by R4)
R2 = pointer to palette to write
R3 reserved (must be zero)
R4 = flags (used if R0 ≥ 0):
bit 0 set ⇒ R1 = pointer to sprite; else R1 = pointer to sprite name
bit 1 set ⇒ flashing colours in table; else not present
all other bits reserved (must be zero)

On exit

—

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call writes to either the screen's palette, or to a sprite's palette.

256 colour palettes are first compacted to the base 16 entries used by VIDC – but only if the compacted palette expands via the tint mechanism to the original palette. Otherwise the full 256 colours are written.

This call is not available in RISC OS 2.

Related SWIs

None

Related vectors

PaletteV

ColourTrans_SetColour (SWI &4075E)

Changes the foreground or background colour to a GCOL number

On entry

R0 = GCOL number
R3 = flags:
bit 7 set ⇒ set background, else foreground
R4 = GCOL action

On exit

All registers preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call changes the foreground or background colour to a GCOL number (as returned from ColourTrans_ReturnGCOL). You should only use it for GCOL numbers returned for the current mode.

This call is not available in RISC OS 2.

Related SWIs

ColourTrans_ReturnGCOL (SWI &40742)

Related vectors

PaletteV

ColourTrans_MiscOp (SWI &4075F)

This call is for internal use only. It is not available in RISC OS 2.

* Commands

*ColourTransMap

Sets up a calibration table from its parameters

Syntax

`*ColourTransMap` *RRGGBBDD RRGGBBDD RRGGBBDD RRGGBBDD etc.*

Parameters

RRGGBBDD	8 hex digits, such that &RRGGBBDD is the number to be placed in the calibration table

Use

*ColourTransMap sets up a calibration table from its parameters. The number of parameters passed for each component must have been specified in a previous *ColourTransMapSize command.

The main purpose of this command is to enable the Task Manager to save the calibration when a desktop save is done.

This command is not available in RISC OS 2.0.

Example

```
*ColourTransMap 01000000 FF0000FF 00020000 00FE00FF etc
```

Related commands

*ColourTransMapSize

Related SWIs

ColourTrans_WriteCalibrationToFile (SWI &40757)

Related vectors

ColourV

*ColourTransMapSize

Sets how parameters will be passed in the next *ColourTransMap command

Syntax

```
*ColourTransMapSize n1 n2 n3
```

Parameters

n1	number of parameters to be passed in *ColourTransMap for component 1
n2	number of parameters to be passed in *ColourTransMap for component 2
n3	number of parameters to be passed in *ColourTransMap for component 3

Use

*ColourTransMapSize sets the number of parameters that will be passed in the next *ColourTransMap command for each component. It hence also sets the size of the resultant calibration table, which will be (3 + n1 + n2 + n3) words long.

The main purpose of this command is to enable the Task Manager to save the calibration when a desktop save is done.

This command is not available in RISC OS 2.0.

Example

```
*ColourTransMapSize 8 10 8
```

Related commands

*ColourTransMap

Related SWIs

ColourTrans_WriteCalibrationToFile (SWI &40757)

Related vectors

ColourV