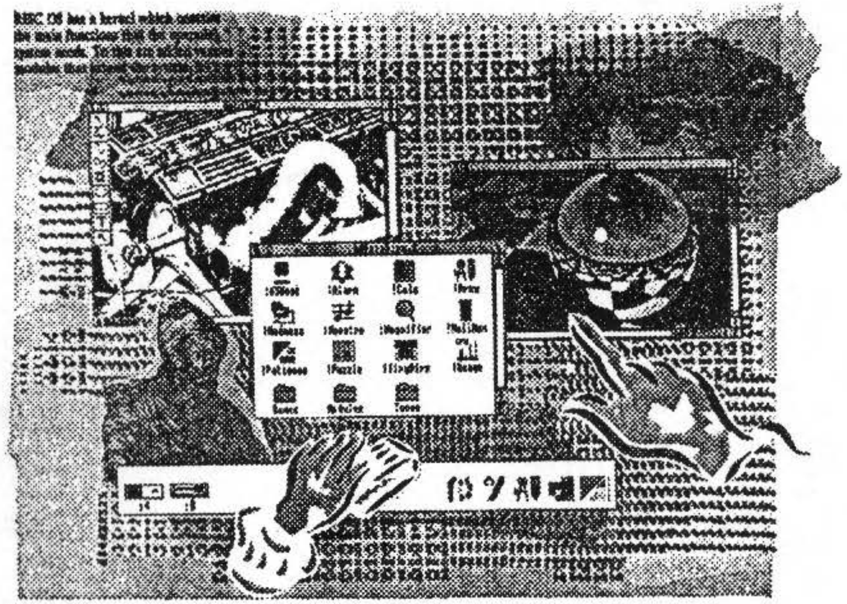
RISC OS

PROGRAMMER'S REFERENCE MANUAL

Volume V



Acorn

Copyright © Acorn Computers Limited 1991

Published by Acorn Computers Technical Publications Department

Neither the whole nor any part of the information contained in, nor the product described in, this manual may be adapted or reproduced in any material form except with the prior written approval of Acorn Computers Limited.

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

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

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

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

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

ACORN, ACORNSOFT, ACORN DESKTOP PUBLISHER, ARCHIMEDES, ARM, ARTHUR, ECONET, MASTER, MASTER COMPACT, THE TUBE, VIEW and VIEWSHEET are trademarks of Acorn Computers Limited.

ADOBE and POSTSCRIPT are trademarks of Adobe Systems Inc
AUTOCAD is a trademark of AutoDesk Inc
AMIGA is a trademark of Commodore-Amiga Inc
ATARI is a trademark of Atari Corporation
COMMODORE is a trademark of Commodore Electronics Limited
DBASE is a trademark of Ashton Tate Ltd
EPSON is a trademark of Epson Corporation
ETHERNET is a trademark of Xerox Corporation
HPGL and LASERJET are trademarks of Hewlett-Packard Company
LASERWRITER is a trademark of Apple Computer Inc
LOTUS 123 is a trademark of The Lotus Corporation
MS-DOS is a trademark of Microsoft Corporation
MULTISYNC is a trademark of NEC Limited
SUN is a trademark of Sun Microsystems Inc
SUPERCALC is a trademark of Computer Associates
TEX is a trademark of the American Mathematical Society

UNIX is a trademark of AT&T
VT is a trademark of Digital Equipment Corporation
1ST WORD PLUS is a trademark of GST Holdings Ltd

Published by Acorn Computers Limited
ISBN 1 85250 115 9
Edition 1
Part number 0470,295
Issue 1, October 1991

Contents

55 The Font Manager

Introduction

A *font* is a complete set of characters of a given type *style*. The font manager provides facilities for painting characters of various sizes and styles on the screen.

To allow characters to be printed in any size, descriptions of fonts can be held in files as size-independent outlines, or pre-computed at specific sizes. The font manager allows programs to request font types and sizes by name, without worrying about how they are read from the filing system or stored in memory.

The font manager also scales fonts to the desired size automatically if the exact size is not available. The fonts are, in general, proportionally spaced, and there are facilities to print justified text – that is, adjusting spaces between words to fit the text in a specified width.

An *anti-aliasing* technique can be used to print the characters. This technique uses up to 16 shades of colour to represent pixels that should only be partially filled-in. Thus, the illusion is given of greater screen resolution.

The font manager can use *hints*, which help it scale fonts to a low resolution while retaining maximum legibility.

RISC OS 2.0

References in this chapter to the RISC OS 2.0 font manager describe the **outline** font manager that is supplied with Release 1.02 of Acorn Desktop Publisher. The RISC OS 2.0 ROM contains an earlier version of this font manager called the **bitmap** font manager. This is no longer supported, and you should always use the outline font manager.

Overview

The font manager can be divided internally into the following components:

- Find and read font files
- Cache font data in memory
- Get a handle for a font style (many commands use this handle)
- Paint a string to the VDU memory
- Change the colours that the text is painted in
- Other assorted SWIs to handle scaling and measurements

Measurement systems

Much of the font manager deals with an internal measurement system, using millipoints. This is 1/1000th of a point, or 1/72000th of an inch. This system is an abstraction from the physical characteristics of the VDU. Text can therefore be manipulated by its size, rather than in terms of numbers of pixels, which will vary from mode to mode.

OS coordinates

OS coordinates is the other system used. There are defined to be 180 OS units per inch. This is the coordinate system used by the VDU drivers, and is related to the physical pixel layout of the screen. Calls are provided to convert between these two systems, and even change the scaling factor between them.

Referencing fonts by name

A SWI is provided to scan through the list of available fonts. This allows a program to present the user with a list to select from. It is a good idea to cache this information as reading the font list with the SWI is a slow process. Another SWI will return a handle for a given font style. A handle is a byte that the font manager uses as an internal reference for the font style. This is like an Open command in a filing system. The equivalent of Close is also provided. This tells the font manager that the program has finished with the font.

There is a SWI to make a handle the currently selected one. This will be used implicitly by many calls in the font manager. It can be changed by commands within a string while painting to the VDU.

Cacheing

Cacheing is the technique of storing one or more fonts in a designated space in memory. The cacheing system decides what gets kept or discarded from its space. Two CMOS variables control how much space is used for cacheing. One sets the minimum amount, which no other part of the system will use. The other sets the maximum amount, which is the limit on what the font manager can expand the cache to.

You should adjust these settings to suit the font requirements of your application. If too little is allowed, then the system will have to continually re-load the fonts from file. If it is too large, then you will use up memory that could be used for other things.

The command *FontList is provided to show the total and used space in the cache, and what fonts are held in it. This is useful to check how the cache is occupied.

Colours

The anti-aliasing system uses up to 16 colours, depending on the screen mode. It will try, as intelligently as possible, to use these colours to shade a character giving the illusion of greater resolution.

Logical colours

The colour shades start with a background value, which is usually the colour that the character is painted onto. They progress up to a foreground colour, which is the desired colour for the character to appear in. This is usually what appears in the centre of the character. Both of these can be set to any valid logical colour numbers.

Palette

In between background and foreground colours can be a number of other logical colours. There is a call to program the palette so that these are set to graduating intermediate levels. The points of transition are called thresholds. The thresholds are set up so that the gradations produce a smooth colour change from background to foreground.

Painting

A string can be painted into the VDU memory. As well as printable characters which are displayed in the current font style, there are non-printing command characters, used in much the same way as those in the VDU driver. They can perform many operations, such as:

- changing the colour
- altering the write position in the x and y axes
- changing the font handle
- changing the appearance and position of the underlining

By using these command characters, a single string can be displayed with as many changes of these characteristics as required

Measuring

Many SWIs exist to measure various attributes of fonts and strings. With a font, you can determine the largest box needed to contain any character in the set. This is called its bounding box. You can also check the bounding box of an individual character.

With a string, you can measure its bounding box, or check where in the string the caret would be for a given coordinate. The caret is a special cursor used with fonts. It is usually displayed as a vertical bar with twiddles on each end.

VDU calls

A number of font manager operations can be performed through VDU commands. These have been kept for compatibility and you should not use them, as they may be phased out in future versions.

Technical Details

An easy way to introduce you to programming with the font manager is to use a simple example. It shows how to paint a text string on the screen using font manager SWIs. Further on in this section is a more detailed explanation of these and all other font SWIs.

Here is the sequence that you would use:

- Font_FindFont – to 'open' the font in the size required
- Font_SetFont – to make it the currently selected font and size
- Font_SetPalette – to set the range of colours to use
- Font_Paint – to paint the string on the screen
- Font_LoseFont – to 'close' the font

Measurement systems

Internal coordinates

The description of character and font sizes comes from specialist files called metrics files. The numbers in these files are held in units of 1/1000th of an em. An em is the size of a point multiplied by the point size of the font. For example, in a 10 point font, an em is 10 points, while in a 14 point font it is 14 points. The font manager converts 1000ths of ems into 1000ths of points, or millipoints, to use for its internal coordinate system. A millipoint is equal to 1/72000th of an inch. This has the advantage that rounding errors are minimal, since coordinates are only converted for the screen at the last moment. It also adds a level of abstraction from the physical characteristics of the target screen mode.

OS coordinates

Unfortunately, the coordinates provided for plot calls are only 16 bits, so this would mean that text could only be printed in an area of about 6/7ths of an inch.

Therefore, the font painter takes its initial coordinates from the user in the same coordinates as the screen uses, which are known as OS units. To make the conversion from OS units to points, the font painter assumes by default that there are 180 OS units to the inch. You can read and set this scale factor, which you may find useful to accurately calibrate the on screen fonts, or to build high resolution bitmaps.

Internal resolution

When the font painter moves the graphics point after printing a character, it does this internally to a resolution of millipoints, to minimise the effect of cumulative errors. The font painter also provides a justification facility, to save you the trouble of working the positions out yourself. The application can obtain the widths of characters to a resolution of millipoints.

SWIs

A pair of routines can be used to convert to and from internal millipoint coordinates to the external OS coordinates. Font_ConverttoOS (SWI &40088) will go from millipoints, while Font_Converttopoints (SWI &40089) will go to them.

Scaling factor

The scaling factor that the above SWIs (and many others in the font manager) use can be read with Font_ReadScaleFactor (SWI &4008F), or set with Font_SetScaleFactor (SWI &40090).

Font files

The font files relating to a font are all contained in a single directory and one or more *encoding* subdirectories:

Filename	Contents
IntMetrics	metrics information for default encoding
IntMetric0	metrics information for encoding /Base0
IntMetric*n*	metrics information for encoding to ISO 8859/*n*
encoding.x90y45	old format pixel file (4-bits-per-pixel) for *encoding*
encoding.f9999x9999	new format pixel file (4-bits-per-pixel) for *encoding*
encoding.b9999x9999	new format pixel file (1-bits-per-pixel) for *encoding*
Outlines	outline file for default encoding
Outlines0	outline file for encoding /Base0
Outlines*n*	outline file for encoding to ISO 8859/*n*

The '9999's referred to above mean 'any decimal number in the range 1 - 9999'. They refer to the pixel size of the font contained within the file, which is equal to:

(font size in 1/16ths of a point) × dots per inch / 72

so, for example, a file containing 4-bits-per-pixel 12 point text at 90 dots per inch would be called f240x240, because 12 × 16 × 90 / 72 = 240.

The formats of these files are detailed in *Appendix E: File formats* on page 6-391.

The default encoding for an alphabetic font (as opposed to symbol fonts, which have a fixed encoding) depends on the alphabet number of the current encoding. The encoding /Base0 includes all the characters supplied with a font; for an example of it, and of the Latin... encodings, see the file:

Resources:$.Fonts.Encodings

For details of the different ISO 8859 character sets, see *Table D: Character sets* on page 6-491.

The minimal requirement for a font is that it should contain an IntMetrics file, and an Outlines file (which we strongly urge you to include) or an x90y45 file. In addition, it can have any number of f9999x9999 or b9999x9999 files, to speed up the cacheing of common sizes.

Master and slave fonts

If outline data or scaled 4-bpp data is to be used as the source of font data it is first loaded into a 'master' font in the cache, which can be shared between many 'slave' fonts at various sizes. There can be only one master font for a given font name, regardless of size, whereas each size of font requires a separate slave font. If the data is loaded directly from the disc into the slave font, the master font is not required.

Referencing fonts by name

The font manager uses the path variable Font$Path when it searches for fonts. This contains a list of full pathnames - each of which has (as in all ...$Path variables) a trailing '.' - which are, in turn, placed before the requested font name. The font manager uses the first directory that matches, provided it also contains an IntMetrics file. Because the variable is a list of path names, you can keep separate libraries of fonts.

Early versions of the font manager used the variable Font$Prefix to specify a single font directory. For compatibility, the font manager looks when it is initialised to see if Font$Path has been defined - if not, it initialises it as follows:

```
*SetMacro Font$Path <Font$Prefix>.
```

This ensures that the old Font$Prefix directory is searched if you haven't explicitly set up the font manager to look elsewhere. The trailing '.' is needed, as Font$Prefix does not include one, and Font$Path requires one.

*FontCat will list all the fonts that can be found using Font$Path.

Changing the font path

Applications which allow the user access to fonts should call Font_ListFonts repeatedly to discover the list of fonts available. This is normally done when the program starts up. The same call can be used with different parameters to build a menu of available fonts (but not under RISC OS 2.0).

The commands *FontInstall, *FontRemove and *FontLibrary add directories to Font$Path, or remove them. Service_FontsChanged is then issued to notify applications update their list of fonts available by calling Font_ListFonts again. These commands are not available under RISC OS 2.0, but where possible, you should use them.

RISC OS 2.0

Under RISC OS 2.0 families of fonts are often found in a separate font 'application' directory, the !Run file of which RMEnsures the correct font manager module from within itself, and then either adds itself to Font$Path or resets Font$Path and Font$Prefix so that it is the only directory referenced.

In order to ensure that the user can access the new fonts available, applications running under RISC OS 2.0 should check whether the value of Font$Path or Font$Prefix has changed since the list of fonts was last cached, and recache the list if so. A BASIC program could accomplish this as follows:

```
size% = &200
DIM buffer% size%         : REM this could be a scratch buffer

...

SYS "OS_GSTrans","<Font$Prefix> and <Font$Path>",buffer%,size%-1 TO ,,length%
buffer%?length% = 13    :REM ensure there is a terminator (13 for BASIC)
IF $buffer%<>oldfontpath$ THEN
   oldfontpath$ = $buffer%
   PROCcache_list_of_fonts
ENDIF
```

Note that if the buffer overflows the string is simply truncated, so it is possible that the check may miss some changes to Font$Prefix. However, since new elements are normally added to the front of Font$Path, this will probably not matter.

The application could scan the list of fonts when it started up, remembering the value of Font$Path and Font$Prefix in `oldfontpath$`, and then make the check described above just before the menu tree containing the list of fonts was about to be opened.

Alternatively the application could scan the list of fonts only when required, by setting `oldfontpath$=""` when it started up, and checking for Font$Path changing only when the font submenu is about to be opened (using the Message_MenuWarning message protocol.)

Opening and closing a font

In order to use a font, Font_FindFont (SWI &40081) must be used. This returns a handle for the font, and can be considered conceptually like a file open. In order to close it, Font_LoseFont (SWI &40082) must be used.

Handles

Font_ReadDefn (SWI &40083) will read the description of a handle, as it was created with Font_FindFont.

In order for a handle to be used, it should be set as the current handle with Font_SetFont (SWI &4008A). This setting stays until changed by another call to this function, or while painting, by a character command to change the handle.

Font_CurrentFont (SWI &4008B) will tell you what the handle of the currently selected font is.

Cacheing

Setting cache size

The size of the cache can be set with two commands. *Configure FontSize sets the minimum that will be reserved. This allocation is protected by RISC OS and will not be used for any other purpose. Running the Task Display from the desktop and sliding the bar for font cache will change this setting until the next reset.

Above this amount, *Configure FontMax sets a maximum amount of memory for font cacheing. The difference between FontSize and FontMax is taken from unallocated free memory as required to accommodate fonts currently in use. If other parts of the system have used up all this memory, then fonts will be limited to FontSize. If there is plenty of free unallocated memory, then FontMax will stop font requirements from filling up the system with cached fonts.

Cache size

*FontList will generate a list of the size and free space of the cache, as well as a list of the fonts currently cached. Font_CacheAddr (SWI &40080) can be used in a program to get the cache size and free space.

Font_LoseFont

When a program calls Font_LoseFont, the font may not be discarded from memory. The cacheing system decides when to do this. A usage count is kept, so that it knows when no task is currently using it. An 'age' is also kept, so that the font manager knows when it hasn't been used for some time.

Cache formats

The cache format, and the algorithms used for cacheing characters, change from release to release. You must not directly access the cache.

Saving and loading the cache

You can use the commands *SaveFontCache and *LoadFontCache to save the font cache in a known state, and to reload it later. This can be a useful speed-up for your applications.

Colours

Colour selection with the font manager involves the range of logical colours that are used by the anti-aliasing software and the physical colours that are displayed.

Logical colours

The logical colour range required is set by Font_SetFontColours (SWI &40092). This sets the background colour, the foreground colour and the range of colours in between.

Physical colours

Font_SetPalette (SWI &40093) duplicates what Font_SetFontColours does, and uses two extra parameters. These specify the foreground and background physical colours, using 4096 colour resolution. Given a range of logical colours and the physical colours for the start and finish of them, this SWI will program the palette with all the intermediate values.

Wimp environment

It must be strongly emphasised that if the program you are writing is going to run under the wimp environment then you must not use Font_SetPalette. It will damage the Wimp's colour information. It is better to use Wimp_SetFontColours (SWI &400F3) or ColourTrans_SetFontColours (SWI &4074F) to use colours that are already in the palette.

Thresholds

The setting of intermediate levels uses threshold tables. These can be read with Font_ReadThresholds (SWI &40094) or set with Font_SetThresholds (SWI &40095). They use a lookup table that is described in Font_ReadThresholds.

Painting

Font_Paint (SWI &40086) is the central SWI that puts text onto the screen. It commences painting with the current handle, set with Font_SetFont. Printable characters it displays appropriately, using the current handle. Using Font_Paint, you can justify the text, back it with a rubout box, transform it, and/or apply kerning to its characters.

A number of embedded command sequences (introduced by control characters) change the way the string is painted:

Number	Effect
9	x coordinate change in millipoints
11	y coordinate change in millipoints
17	change foreground or background colour
18	change foreground, background and range of colours
19	set colours using ColourTrans_SetFontColours (not in RISC OS 2.0)
21	comment string that is not displayed
25	change underline position and thickness
26	change font handle
27	set new transformation matrix (not in RISC OS 2.0)
28	set new transformation matrix (not in RISC OS 2.0)

Note that these are **not** compatible with VDU commands. Any non-printing characters not in the above list will generate an error, apart from 0, 10 and 13 (which are the only valid terminators).

Measuring

There are a number of calls to return information about a string or character. Most of these are obsolete calls from earlier versions of the font manager, which are still supported for backward compatibility.

To get information on a string, you should call Font_ScanString. To get information on a character, you should call Font_CharBBox.

After using Font_ScanString, you can call Font_FutureFont (SWI &4008C). This will return what the font and colours would be if the string was passed through Font_Paint.

Caret

If the pointer is clicked on a string, and the caret needs to be placed on a character, it is necessary to calculate where on the string it would be. Again, Font_ScanString can do this.

You can plot the caret at a given height, position and colour using Font_Caret (SWI &40087). Its height should be adjusted to suit the point size of the font it is placed with. The information returned from Font_ScanString would be appropriate for this adjustment.

Mixing fonts' metrics and characters

Where you are using an external printer (eg. PostScript) which has a larger range of fonts than those available on the screen, it can often be useful to use a similar-looking font on the screen, using the appropriate metrics (ie spacing) for the printer font.

The font manager provides a facility whereby a font can be created which has its own IntMetrics file, matching the appropriate font on the printer, but uses another font's characters on the screen.

This is done by putting a file called 'Outlines' in the font's directory which simply contains the name of the appropriate screen font to use. The font manager will use the IntMetrics file from the font's own directory, but will look in the other font's directory for any bitmap or outline information.

Service Calls

Service_FontsChanged (Service Call &6E)

New FontSPath detected

On entry

R1 = &6E (reason code)

On exit

All registers preserved

Use

This is issued by the Font manager to notify any applications that Font_ListFonts should be called to update the list of fonts available.

SWI Calls

Font_CacheAddr (SWI &40080)

Get the version number, font cache size and amount used

On entry

—

On exit

R0 = version number
R2 = total size of font cache (bytes)
R3 = amount of font cache used (bytes)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The version number returned is the actual version multiplied by 100. For example, version 2.42 would return 242.

This call also returns the font cache size and the amount of space used in it.

*FontList can be used to display the font cache size and space.

Related SWIs

None

Related vectors

None

Font_FindFont (SWI &40081)

Get the handle for a font

On entry

R1 = pointer to font name (terminated by a Ctrl char)
R2 = x point size × 16 (ie in 1/16ths point)
R3 = y point size × 16 (ie in 1/16ths point)
R4 = x resolution in dots per inch (0 ⇒ use default, −1 ⇒ use current)
R5 = y resolution in dots per inch (0 ⇒ use default, −1 ⇒ use current)

On exit

R0 = font handle
R1 - R3 preserved
R4 = x resolution in dots per inch
R5 = y resolution in dots per inch

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call returns a handle to a font whose name, point size and screen resolution are passed. It also sets it as the current font, to be used for future calls to Font_Paint etc.

The VDU command:

```
VDU 23,26,font_handle,pt_size,x_dpi,y_dpi,x_scale,y_scale,0,0,font_name
```

is an equivalent command to this SWI. As with all VDU font commands, it has been kept for compatibility with earlier versions of the operation system and must not be used.

The font name can also have various qualifiers added to it, which are a '\' followed by an identifying letter and the value associated with the qualifier. These qualifiers are not supported by RISC OS 2.0. If the string does not start with a '\', it is assumed that it is a font identifier.

The strings following qualifiers must not contain '\', as this denotes the start of the next qualifier.

The possible qualifiers are:

\F<identifier>	font identifier (as for earlier implementations of Font_FindFont)
\f<t> <name>	territory number for font name, followed by the font name
\E<identifier>	encoding identifier
\e<t> <name>	territory number for encoding name, followed by the encoding name
\M<matrix>	transformation matrix to apply to this font

where:

- <identifier> is a string of ASCII characters, in the range 33 to 126 inclusive, which must represent a legal filename (although it can contain '.'s).
- <name> is the name of the font/encoding, expressed in the language of the current territory, and using the alphabet of the current territory, and terminated by an end-of-string.
- <t> is the territory number of the current territory, ie the language in which the font/encoding name is expressed. It is followed by a space character, to separate it from the following <name>.
- <matrix> is a set of 6 signed decimal integers which represent the values of the 6 words that go into making a draw-type matrix: the first four numbers are in fact 16-bit fixed point, and the last two are offsets in 1/1000th of an em.

The font identifier is the name of the font directory without the Font$Path prefix, and is invariant in any territory. The font name is the name of the font (ie the one displayed to the user) in the given territory.

If Font_FindFont fails to find the font, an error message 'Font '<name>' not found' is returned, where <name> is the font name if the current territory is the same as the one in the string, and is the font identifier otherwise.

Applications should store the entire string returned from Font_DecodeMenu in the document, so that if a user loads the document without having the correct fonts available, the font name – rather than the identifier – can be returned, as long as the user is in the same territory.

The '\E' (encoding) field indicates the appropriate encoding for the font itself. This field is only supplied by Font_DecodeMenu if the font is deemed to be a 'language' font, ie one whose encoding depends on the territory. Other fonts are thought of as 'Symbol' fonts, which have a fixed encoding.

Note that Font_DecodeMenu will return a font identifier of the following form:

\F<fontid>\f<territory> <fontname>

To apply a particular encoding to a font, remember to eliminate the existing encoding fields (if present) first. Note that no field is allowed to contain a '\'.

\E<encid>\e<t> <encname>\F<fontid>\f<t> <fontname>

Since <fontid>\f<t> <fontname> is also accepted by Font_FindFont, when prepending '\E<encid>\e<t> <encname>' on the front, you should also put '\F' on the front of the original string if it did not start with '\'.

In BASIC, this looks like:

```
REM original$ is the original string passed to Font_FindFont
REM encoding$ is the string returned from Font_DecodeMenu
REM           typically "\E<enc_id>\e <territory> <enc_name>"
REM result is the new string to be passed to Font_FindFont

DEF FNapply_encoding_to_font(original$,encoding$)
IF LEFT$(original$,1)<>"\" THEN original$ = "\F"+original$
original$ = FNremove(original$,"\E")
original$ = FNremove(original$,"\e")
= encoding$ + original$

REM this function removes the specified field from the string
REM eliminates all characters from b$ to "\"

DEF FNremove(a$,b$)
LOCAL I%,J%
I% = INSTR(a$,b$)
IF I%=0 THEN =a$                  :REM nothing to eliminate
J% = INSTR(a$+"\","\",I%+1)       :REM searches from I%+1
= LEFT$(a$,I%-1)+MID$(a$,J%)
```

In fact it is not strictly necessary to remove the original encoding fields from the font identifier, since an earlier occurrence of a field overrides a later one; but if this is not done then the length of the total string will continue to grow every time an encoding is altered.

Related SWIs

Font_LoseFont (SWI &40082)

Related vectors

None

Font_LoseFont (SWI &40082)

Finish use of a font

On entry

R0 = font handle

On exit

R0 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call tells the font manager that a particular font is no longer required.

Related SWIs

Font_FindFont (SWI &40081)

Related vectors

None

Font_ReadDefn (SWI &40083)

Read details about a font

On entry

R0 = font handle
R1 = pointer to buffer to hold font name

On exit

R1 = pointer to buffer (now contains font name)
R2 = x point size × 16 (ie in 1/16ths point)
R3 = y point size × 16 (ie in 1/16ths point)
R4 = x resolution (dots per inch)
R5 = y resolution (dots per inch)
R6 = age of font
R7 = usage count of font

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call returns a number of details about a font. The usage count gives the number of times that Font_FindFont has found the font, minus the number of times that Font_LoseFont has been used on it. The age is the number of font accesses made since this one was last accessed.

Note that the x resolution in a 132 column mode will be the same as an 80 column mode. This is because it is assumed that it will be used on a monitor that displays it correctly, which is not the case with all monitors.

Related SWIs

None

Related vectors

None

Font_ReadInfo (SWI &40084)

Get the font bounding box

On entry

R0 = font handle

On exit

R1 = minimum x coordinate in OS units for the current mode (inclusive)
R2 = minimum y coordinate in OS units for the current mode (inclusive)
R3 = maximum x coordinate in OS units for the current mode (exclusive)
R4 = maximum y coordinate in OS units for the current mode (exclusive)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call returns the minimal area covering any character in the font. This is called the font bounding box.

You should use the SWI Font_CharBBox (see page 5-41) in preference to this one.

Related SWIs

Font_CharBBox (SWI &4008E), Font_StringBBox (SWI &40097)

Related vectors

None

Font_StringWidth (SWI &40085)

Calculate how wide a string would be

On entry

R1 = pointer to string
R2 = maximum x offset before termination in millipoints
R3 = maximum y offset before termination in millipoints
R4 = 'split' character (–1 for none)
R5 = index of character to terminate by

On exit

R1 = pointer to character where the scan terminated
R2 = x offset after printing string (up to termination)
R3 = y offset after printing string (up to termination)
R4 = no of 'split' characters in string (up to termination)
R5 = index into string giving point at which the scan terminated

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is used to calculate how wide a string would be.

The 'split' character is one at which the string can be split if any of the limits are exceeded. If R4 contains –1 on entry, then on exit it contains the number of printable (as opposed to 'split') characters found.

The string is allowed to contain command sequences, including font-change (26,<font handle>) and colour-change (17,<colour>). After the call, the current font foreground and background call are unaffected, but a call can be made to Font_FutureFont to find out what the current font would be after a call to Font_Paint.

The string width function terminates as soon as R2, R3 or R5 are exceeded, or the end of the string is reached. It then returns the state it had reached, either:

- just before the last 'split' char reached
- if the 'split' char is –1, then before the last char reached
- if R2, R3 or R5 are not exceeded, then at the end of the string.

By varying the entry parameters, the string width function can be used for any of the following purposes:

- finding the cursor position in a string if you know the coordinates (although Font_FindCaret is better for this)
- finding the cursor coordinates if you know the position
- working out where to split lines when formatting (set R4=32)
- finding the length of a string (eg for right-justify)
- working out the data for justification (as the font manager does).

You should use the SWI Font_ScanString (see page 5-79) in preference to this one.

Related SWIs

Font_FutureFont (SWI &4008C), Font_ScanString (SWI &400A1)

Related vectors

None

Font_Paint (SWI &40086)

Write a string to the screen

On entry

R0 = initial font handle (1 - 255) or 0 for current handle – if bit 8 of R2 is set
R1 = pointer to string
R2 = plot type:
- bit 0 set ⇒ use graphics cursor justification coordinates (bit 5 must be clear); else use R5 to justify (if bit 5 is set) or don't justify
- bit 1 set ⇒ plot rubout box using either graphics cursor rubout coordinates (if bit 5 is clear) or R5 (if bit 5 is set); else don't plot rubout box
- bits 2,3 reserved (must be zero)
- bit 4 set ⇒ coordinates are in OS units; else in millipoints
- bit 5 set ⇒ use R5 as indicated below (bit 4 must be clear)
- bit 6 set ⇒ use R6 as indicated below (bit 4 must be clear)
- bit 7 set ⇒ use R7 as indicated below
- bit 8 set ⇒ use R0 as indicated above
- bit 9 set ⇒ perform kerning on the string
- bit 10 set ⇒ writing direction is right to left; else left to right

R3 = start x-coordinate (in OS coordinates or millipoints, depending on bit 4 of R2)
R4 = start y-coordinate (in OS coordinates or millipoints, depending on bit 4 of R2)
R5 = pointer to coordinate block – if bit 5 of R2 is set
R6 = pointer to transformation matrix – if bit 6 of R2 is set
R7 = length of string – if bit 7 of R2 is set

On exit

R1 - R7 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call writes a string to the screen, optionally justifying it, backing it with a rubout box, transforming it, and/or applying kerning to its characters.

RISC OS 2.0 ignores the values of R0 and of R5 - R7, and behaves as though bits 2,3 and 5 - 31 inclusive of R2 are clear.

Setting graphics cursor coordinates

To set the graphics cursor justification coordinates, you must have previously called a VDU 25 move command. Likewise, to set the graphics cursor rubout coordinates, you must have called VDU 25 twice, to describe the rectangle to clear: first the lower-left coordinate (which is inclusive), then the upper-right coordinate (which is exclusive). Thus, to use both the justification and rubout coordinates, you must have made three VDU 25 moves, with the justify coordinates being last. The font manager rounds all these coordinates to the nearest pixel.

The coordinate block

The coordinate block pointed to by R5 contains eight words giving additional spacing to use to achieve justification, and coordinates for the rubout box. The values are in OS units (bit 4 of R2 is set) or millipoints (bit 4 of R2 is clear):

Offset	Value
0	additional x, y offset on space
8	additional x, y offset between each letter
16	x, y coordinates for bottom left of rubout box (inclusive)
24	x, y coordinates for top right of rubout box (exclusive)

Justification

Justification can be done in one of two ways, depending on the value of bits 0 and 5 of R2:

- If bit 0 of R2 is set, the text is justified between the start coordinates (given in R3, R4) and the graphics cursor justification coordinates (see above). In fact, the justification y-coordinate is ignored as being too inaccurate, and the start y-coordinate used for both ends of the text.
- If bit 0 of R2 is clear, and bit 5 set, the additional offsets pointed to by R5 are used instead. Left justification can be achieved by simply setting these four words to zero.

The rubout box

Similarly, there are two different ways to plot a rubout box. Bit 1 of R2 must be set; then:

- If bit 5 of R2 is clear, the graphics cursor rubout box coordinates get used. In this case they are treated as inclusive, as in graphics window setting.
- If bit 5 of R2 is set, the coordinates in the block pointed to by R5 are used instead. In this case pixels are filled only if the pixel centre is enclosed, as in Draw_Fill.

Transformation matrices

The buffer pointed to by R6 contains a transformation matrix, held as six words. The first four words are 32-bit signed numbers, with a fixed point after bit 16 (ie 1 is represented by 1 << 16, which is 65536). The translations are in OS units (bit 4 of R2 is set) or millipoints (bit 4 of R2 is clear):

Offset	Value
0	four fixed point multipliers of transformation matrix
16	x, y coordinates for translation element of transformation matrix

Subsequent matrices can be included within the string (not in RISC OS 2.0); they alter the matrix to the specified value, rather than being concatenated with any previous matrix. Such changes are made by including one of the following control sequences:

27, <align>, <m1>, <m2>, <m3>, <m4>
28, <align>, <m1>, <m2>, <m3>, <m4>, <m5>, <m6>

where <align> means 'advance until the pointer is word-aligned'. The equation for this is:

pointer = (pointer + 3) AND NOT 3

m1 - m4 are little-endian 32-bit signed numbers with a fixed point after bit 16 (ie 1 is represented as 1 << 16, which is 65536).

m5 and m6 are the offsets, which are in millipoints (even if R2 bit 4 is set). These values are assumed to be 0 if the 27,m1...m4 code is used.

To restore the unit matrix, use 27,<align>,65536,0,0,65536.

Note that underlining and rubout do not work correctly if the x-axis is transformed so that it is no longer on the output x-axis, or has its direction reversed. The effect when doing this should **not** be relied on.

Text direction

If bit 10 of R2 is set, then text is written right to left, rather than left to right. In this case the width of each character is subtracted from the position of the current point before painting the character, rather than the width being added after painting it. Rubout and underline are also filled in from right to left.

When kerning, the kern pairs stored in the metrics file indicate the left and right hand characters of a pair, and the additional offset to be applied between the characters if this pair is found. Note that if the main writing direction is right to left, then the right hand character is encountered first, and the left hand one is encountered next.

String length

Note that the character at [R1,R7] may be accessed, to determine the character offset due to kerning (which in turn affects the underline width). This will not be a problem if the string has a terminator, and the R7=length facility is only used to extract substrings.

Changing colour

You can change the colour used by including this control sequence in the string:

19,<r>,<g>,<b>,<R>,<G>,<B>,<max>

This results in a call to ColourTrans_SetFontColours (see page 4-412). Again, RISC OS 2.0 does not support this control sequence.

Other control sequences

There are other control sequences that are supported by all versions of RISC OS, and that are similar to certain VDU sequences:

9,<dx low>,<dx middle>,<dx high>
11,<dy low>,<dy middle>,<dy high>
17,<foreground colour> (+&80 for background colour)
18,<background>,<foreground>,<font colour offset>
21,<comment string>,<terminator (any Ctrl char)>
25,<underline position>,<underline thickness>
26,<font handle>

After the call, the current font and colours are updated to the last values set by command characters.

Control sequences 9 and 11 allow for movement within a string. This is useful for printing superscripts and subscripts, as well as tabs, in some cases. They are each followed by a 3-byte sequence specifying a number (low byte first, last byte sign-extended), which is the amount to move by in millipoints. Subsequent characters are plotted from the new position onwards.

An example of moving in the Y direction (character 11) would look like the following example, where *chr()* is a function that converts a number into a character and *move* is the movement in millipoints:

```
MoveString =        chr(11)+chr(move AND &FF)+
                    chr((move AND &FF00) >> 8)+
                    chr((move AND &FF0000) >> 16)
```

Control sequence 17 will act as if the foreground or background parameters passed to Font_SetFontColours (SWI &40092) had been changed. Control sequence 18 allows all three parameters to that SWI to be set. See that SWI for a description of these parameters.

The *underline position* within control sequence 25 is the position of the top of the underline relative to the baseline of the current font, in units of 1/256th of the current font size. It is a sign-extended 8 bit number, so an underline below the baseline can be achieved by setting the underline position to a value greater than 127. The *underline thickness* is in the same units, although it is not sign-extended.

Note that when the underline position and height are set up, the position of the underline remains unchanged thereafter, even if the font in use changes. For example, you do not want the thickness of the underline to change just because some of the text is in italics. If you actually want the thickness of the underline to change, then another underline-defining sequence must be inserted at the relevant point. Note that the underline is always printed in the same colour as the text, and that to turn it off you must set the underline thickness to zero.

Subpixel scaling

This is quite simple if neither x or y scaling is performed, and also if both x and y scaling is performed: the subpixel scaling directions relate to the output device axes.

When just horizontal or just vertical subpixel scaling is performed, it is sometimes necessary to swap over the sense of which is horizontal and which is vertical, in order to determine the 'size' of the font.

This goes for the other FontMax<n> thresholds too, such as FontMax2, which determines whether characters should be anti-aliased. FontMax3 determines whether characters should be cached or not, and this must relate to the amount of memory taken up by the bitmaps.

Scaffolding

Clearly it is not possible to apply scaffolding to characters which are transformed such that its new axes do not lie on the old ones. However, if the axes are mapped onto each other (eg a scale, rotation or reflection about an axis or 45-degree line) then scaffolding can still be applied. This can involve swapping over the x and y scaffolding. If a font is sheared, then scaffolding may be applied in one direction but not the other.

Bounding boxes

The bounding box of a transformed character cannot be determined purely by transforming the original bounding box of the character outline, since bounding boxes are axis-aligned rectangles, and character outlines are not, so the bounding box of the transformed character is typically smaller than that of the transformed bounding box.

Taking the bounding box of the transformed original bounding box is sufficient to work out a large enough box for outline to bitmap conversion, since not much memory is wasted (only one character is done at a time, and the character is 'shrink-wrapped' after conversion).

Bitmap fonts

If a font has an encoding applied to it, then Font_Paint looks inside <fontname>.<encoding> to find the bitmap files. This is because bitmap files are specific to one encoding.

Note that Font_MakeBitmap also generates its bitmap files inside the appropriate encoding subdirectory.

If the font has no encoding applied, the bitmap files are inside the font directory, as before.

Note that this means that encoding names must not clash with any of the filenames that normally reside within font directories, ie:

IntMetrics[<n>] Outlines[<n>]	<n> is optional and the prefix is truncated so it all fits in 10 characters
x90y45	
b<n>x<n> f<n>x<n>	<n> is a number from 1 - 9999

Equivalent VDU command

The VDU command VDU 25,&D0–&D7,*x_coordinate,y_coordinate,text_string* is an equivalent command to this SWI. As with all VDU font commands, it has been kept for compatibility with earlier versions of the operation system and must not be used.

Related SWIs

Font_StringWidth (SWI &40085)

Related vectors

None

Font_Caret (SWI &40087)

Define text cursor for font manager

On entry

R0 = colour (exclusive ORd onto screen)
R1 = height (in OS coordinates)
R2 bit 4 = 0 ⇒ R3, R4 in millipoints
= 1 ⇒ R3, R4 in OS coordinates
R3 = x coordinate (in OS coordinates or millipoints)
R4 = y coordinate (in OS coordinates or millipoints)

On exit

R0 = preserved
R1 = preserved
R2 = preserved
R3 = preserved
R4 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The 'caret' is a symbol used as a text cursor when dealing with anti-aliased fonts. The height of the symbol, which is a vertical bar with 'twiddles' on the end, can be varied to suit the height of the text, or the line spacing.

The colour is in fact Exclusive ORd onto the screen, so in 256-colour modes it is equal to the values used in a 256-colour sprite.

Related SWIs

None

Related vectors

None

Font_ConverttoOS (SWI &40088)

Convert internal coordinates to OS coordinates

On entry

R1 = x coordinate (in millipoints)
R2 = y coordinate (in millipoints)

On exit

R1 = x coordinate (in OS units)
R2 = y coordinate (in OS units)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call converts a pair of coordinates from millipoints to OS units, using the current scale factor. (The default is 400 millipoints per OS unit.)

Related SWIs

Font_Converttopoints (SWI &40089), Font_ReadScaleFactor (SWI &4008F), Font_SetScaleFactor (SWI &40090)

Related vectors

None

Font_Converttopoints (SWI &40089)

Convert OS coordinates to internal coordinates

On entry

R1 = x coordinate (in OS units)
R2 = y coordinate (in OS units)

On exit

R0 is corrupted
R1 = x coordinate (in millipoints)
R2 = y coordinate (in millipoints)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call converts a pair of coordinates from OS units to millipoints, using the current scale factor. (The default is 400 millipoints per OS unit.)

Related SWIs

Font_ConverttoOS (SWI &40088), Font_ReadScaleFactor (SWI &4008F), Font_SetScaleFactor (SWI &40090)

Related vectors

None

Font_SetFont (SWI &4008A)

Select the font to be subsequently used

On entry

R0 = handle of font to be selected

On exit

R0 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets up the font which is used for subsequent painting or size-requesting calls (unless overridden by a command 26,<font> sequence in a string passed to Font_Paint).

You can also set the font by passing its handle in R0 when calling Font_Paint (see page 5-24). Where possible, you should do so in preference to using this SWI.

Related SWIs

Font_SetFontColours (SWI &40092), Font_CurrentFont (SWI &4008B), Font_Paint (SWI &40086)

Related vectors

None

Font_CurrentFont (SWI &4008B)

Get current font handle and colours

On entry

–

On exit

R0 = handle of currently selected font
R1 = current background logical colour
R2 = current foreground logical colour
R3 = foreground colour offset

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call returns the state of the font manager's internal characteristics which will apply at the next call to Font_Paint.

The value in R3 gives the number of colours that will be used in anti-aliasing. The colours are f, f+1… f+offset, where 'f' is the foreground colour returned in R2, and offset is the value returned in R3. This can be negative, in which case the colours are f, f–1… f–|offset|. Negative offsets are useful for inverse anti-aliased fonts.

Offsets can range between –14 and +14. This gives a maximum of 15 foreground colours, plus one for the font background colour. If the offset is 0, just two colours are used: those returned in R1 and R2.

The font colours, and number of anti-alias levels, can be altered using Font_SetFontColours, Font_SetPalette, Font_SetThresholds and Font_Paint.

Related SWIs

Font_SetFont (SWI &4008A), Font_SetFontColours (SWI &40092), Font_SetPalette (SWI &40093), Font_SetThresholds (SWI &40095), Font_Paint (SWI &40086)

Related vectors

None

Font_FutureFont (SWI &4008C)

Check font characteristics after Font_StringWidth

On entry

–

On exit

R0 = handle of font which would be selected
R1 = future background logical colour
R2 = future foreground logical colour
R3 = foreground colour offset

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call can be made after a Font_StringWidth to discover the font characteristics after a call to Font_Paint, without actually having to paint the characters.

Related SWIs

Font_StringWidth (SWI &40085), Font_Paint (SWI &40086)

Related vectors

None

Font_FindCaret (SWI &4008D)

Find where the caret is in the string

On entry

R1 = pointer to string
R2 = x offset in millipoints
R3 = y offset in millipoints

On exit

R1 = pointer to character where the search terminated
R2 = x offset after printing string (up to termination)
R3 = y offset after printing string (up to termination)
R4 = number of printable characters in string (up to termination)
R5 = index into string giving point at which it terminated

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

On exit, the registers give the nearest point in the string to the caret position specified on entry. This call effectively makes two calls to Font_StringWidth to discover which character is nearest the caret position. It is recommended that you use this call, rather than perform the calculations yourself using Font_StringWidth, though this is also possible.

You should use the SWI Font_ScanString (see page 5-79) in preference to this one.

Related SWIs

Font_StringWidth (SWI &40085), Font_FindCaretJ (SWI &40096), Font_ScanString (SWI &400A1)

Related vectors

None

Font_CharBBox (SWI &4008E)

Get the bounding box of a character

On entry

R0 = font handle
R1 = ASCII character code
R2 = flags (bit 4 set ⇒ return OS coordinates, else millipoints)

On exit

R1 = minimum x of bounding box (inclusive)
R2 = minimum y of bounding box (inclusive)
R3 = maximum x of bounding box (exclusive)
R4 = maximum y of bounding box (exclusive)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

You can use this call to discover the bounding box of any character from a given font. If OS coordinates are used and the font has been scaled, the box may be surrounded by an area of blank pixels, so the size returned will not be exactly accurate. For this reason, you should use millipoints for computing, for example, line spacing on paper. However, the millipoint bounding box is not guaranteed to cover the character when it is painted on the screen, so the OS unit bounding box should be used for this purpose.

Related SWIs

Font_ReadInfo (SWI &40084), Font_StringBBox (SWI &40097)

Related vectors

None

Font_ReadScaleFactor (SWI &4008F)

Read the internal to OS conversion factor

On entry

—

On exit

R1 = x scale factor
R2 = y scale factor

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The x and y scale factors are the numbers used by the font manager for converting between OS coordinates and millipoints. The default value is 400 millipoints per OS unit. This call allows the current values to be read.

Related SWIs

Font_ConverttoOS (SWI &40088), Font_SetScaleFactor (SWI &40090), Font_Converttopoints (SWI &40089)

Related vectors

None

Font_SetScaleFactor (SWI &40090)

Set the internal to OS conversion factor

On entry

R1 = x scale factor
R2 = y scale factor

On exit

R1 = preserved
R2 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

Applications that run under the Desktop should not use this call, as other applications may be relying on the current settings. If you must change the values, you should read the current values beforehand, and restore them afterwards. The default value is 400 millipoints per OS unit.

Related SWIs

Font_ConverttoOS (SWI &40088), Font_ReadScaleFactor (SWI &4008F), Font_Converttopoints (SWI &40089)

Related vectors

None

Font_ListFonts (SWI &40091)

Scan for fonts, returning their names one at a time; or build a menu of fonts

On entry

R1 = pointer to buffer for font identifier, or for menu definition (0 to return required size of buffer)
R2 = counter and flags:
- bits 0 - 15 = counter (0 on first call)
- bits 16 - 31 = 0 ⇒ RISC OS 2.0-compatible mode (see below)
- bit 16 set ⇒ return font name in buffer pointed to by R1 (or required size of buffer)
- bit 17 set ⇒ return local font name in buffer pointed to by R4 (or required size of buffer)
- bit 18 set ⇒ terminate strings with character 13, rather than character 0
- bit 19 set ⇒ return font menu definition in buffer pointed to by R1, and indirected menu data in buffer pointed to by R4 (or required sizes of buffers)
- bit 20 set ⇒ put 'System font' at head of menu
- bit 21 set ⇒ tick font indicated by R6, and its submenu parent
- bit 22 set ⇒ return list of encodings, rather than list of fonts
- bits 23 - 31 reserved (must be zero)

R3 = size of buffer pointed to by R1 (if R1 ≠ 0)
R4 = pointer to buffer for font name, or for indirected menu data (0 to return required size of buffer)
R5 = size of buffer pointed to by R4 (if R4 ≠ 0)
R6 = pointer to identifier of font to tick (0 ⇒ no tick, 1 ⇒ tick 'System font')

On exit

R1 preserved
R2 = updated counter and preserved flags if listing identifiers/names (-1 if no more to be listed); or preserved if building menu
R3 = required size of buffer pointed to by R1 (if R1 = 0 on entry); or 0 if building a font menu, and the menu is null; else preserved
R4 preserved
R5 = required size of buffer pointed to by R4 (if R4 = 0 on entry); else preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call has two possible uses:

1 Return a list of font/encoding names and/or local names known to the font manager, and cache the list. The names are returned in alphabetical order, regardless of the order in which they are found. ('Local names' are the names translated to the language of the current territory, if possible.)

  In this case you should first initialise R2. Only bits 16 - 18 and bit 22 may be set; all other bits must be clear. Then for each font/encoding you must call this SWI twice: the first time with R1 and R4 set to zero to find the required sizes of buffers, and the second time with the buffers set up to receive the name(s) of that font/encoding. Do not alter the value of R2 between calls. When R2 is –1 on exit, the last font/encoding has already been found, and any returned name(s) are invalid.

2 Build a menu definition of all fonts known to the font manager. The definition is suitable for passing to Wimp_CreateMenu (see page 4-222).

  In this case you may only set bits 19 - 21 of R2 on entry. You should make the call twice: the first time with R1 and R4 set to zero to find the required sizes of buffers, and the second time with the buffers set up to receive the menu definition.

Fonts are found by searching the path given by the system variable Font$Path, and its subdirectories, for files ending in '.IntMetrics'. Likewise, encodings are searched for by searching the path given by the system variable Font$Path, and its subdirectories, for files of the form '<font prefix>.Encodings.<encoding id>' (which are used to specify the encodings of the 'language' fonts, as opposed to the 'symbol' fonts, the encoding of which is fixed).

When such a file is found, the full name of the subdirectory is put in the buffer, terminated by a carriage return or null. If the same font/encoding name is found via different paths, only the first one will be reported. The local name is found from a Messages file, if present.

Possible errors are 'Buffer overflow' (R3 and/or R5 was too small), or 'Bad parameters' (the flags in R2 were invalid). If an error is returned, R2 = –1 on exit (ie listing fonts/encodings is terminated).

The font manager command *FontCat calls this SWI internally.

Notes on RISC OS 2.0

In the 'RISC OS 2.0-compatible mode' (used if bits 16 - 31 of R2 are clear), this call works as if bits 16 and 18 of R2 were set on entry, bits 17 and 19 - 31 were clear, and R3 was 40 (irrespective of its actual value).

Under RISC OS 2.0, this call works as if bits 16 and 18 of R2 were set on entry, and bits 17 and 19 - 31 were clear (hence R4, R5 and R6 are ignored). However, R3 is used to point to the path to search; a value of –1 means that Font$Path is used instead.

If your program does not RMEnsure the current version of the font manager, you should therefore always use Font$Path to specify the path to search.

Related SWIs

None

Related vectors

None

Font_SetFontColours (SWI &40092)

Change the current colours and (optionally) the current font

On entry

R0 = font handle (0 for current font)
R1 = background logical colour
R2 = foreground logical colour
R3 = foreground colour offset (–14 to +14)

On exit

R0 = preserved
R1 = preserved
R2 = preserved
R3 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is used to set the current font (or leave it as it is), and change the logical colours used. In up to 16 colour modes, the three registers are used as follows:

- R1 is the logical colour of the background
- R2 is the logical colour of the first foreground colour to use
- R3 specifies the offset from the first foreground colour to the last, which is used as the actual foreground colour.

The range specified must not exceed the number of logical colours available in the current screen mode, as follows:

Colours in mode	Possible values of R1,R2,R3 to use all colours
2	0,1,0
4	0,1,2
16 or 256	0,1,14

In a 16 colour mode, to use the top 8 colours, which are normally flashing colours, the values 8,9,6 could be used.

Note that 16 is the maximum number of anti-alias colours. In 256-colour modes, the background colour is ignored, and the foreground colour is taken as an index into a table of pseudo-palette entries – see Font_SetPalette.

Related SWIs

Font_SetFont (SWI &4008A), Font_CurrentFont (SWI &4008B), Font_SetPalette (SWI &40093)

Related vectors

None

Font_SetPalette (SWI &40093)

Define the anti-alias palette

On entry

R1 = background logical colour
R2 = foreground logical colour
R3 = foreground colour offset
R4 = physical colour of background
R5 = physical colour of last foreground
R6 = &65757254 ('True') to use 24 bit colours in R3 and R4

On exit

R1 - R6 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the anti-alias palette.

If the program you are writing is going to run under the Wimp environment then you must not use this call. It will damage the Wimp's colour information. You must instead choose from the range of colours already available by using Wimp_SetFontColours (SWI &400E6) or ColourTrans_SetFontColours (SWI &400E4) instead.

The values in R1, R2 and R3 have the same use as in Font_SetFontColours. See the description of that SWI on the previous page for the use of these parameters.

R4 and R5 contain physical colour setting information. R4 describes the background colour and R5 the foreground colour. The foreground colour is the dominant colour of the text and generally appears in the middle of each character.

The physical colours in R4 and R5 are of the form &BBGGRR00. That is, they consists of four bytes, with the palette entries for the blue, green and red guns in the upper three bytes. Bright white, for instance, would be &FFFFFF00, while half intensity cyan is &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.

Under RISC OS 2.0, this call sets the palette colour for the range described in R1, R2 and R3 using R4 and R5 to describe the colours at each end. It also sets the intermediate colours incrementally between those of R4 and R5. In non-256-colour modes, the palette is programmed so that there is a linear progression from the colour given in R4 to that in R5.

Under later versions of RISC OS, if R6 is set to the magic word 'True', this call treats the values in R3 and R4 as true 24-bit palette values (where white is &FFFFFF00, rather than &F0F0F000). Otherwise, for compatibility, palette values are processed as follows:

R3 = (R3 AND &F0F0F000) OR ((R3 AND &F0F0F000) >> 4)
R4 = (R4 AND &F0F0F000) OR ((R4 AND &F0F0F000) >> 4)

Thus the bottom nibbles of each gun are set to be copies of the top nibbles. Furthermore, this call now uses ColourTrans_WritePalette to set palette entries in non-256-colour modes, and ColourTrans_ReturnColourNumber to match RGB values with logical colours in modes with 256 or more colours. If ColourTrans is not loaded, it calls PaletteV to set the palette; if PaletteV is not intercepted, it finally calls OS_Word 12 to do so.

The VDU command: VDU 23,25,&80+<background logical colour>, <foreground logical colour>,<start R>,<start G>,<start B>,<end R>,<end G>,<end B> is an equivalent command to this SWI. As with all VDU font commands, it has been kept for compatibility with earlier versions of the operation system and must not be used.

Related SWIs

Font_SetFontColours (SWI &40092)

Related vectors

None

Font_ReadThresholds (SWI &40094)

Read the list of threshold values for painting

On entry

R1 = pointer to result buffer

On exit

R1 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call reads the list of threshold values that the font manager uses when painting characters. Fonts are defined using up to 16 anti-aliased levels. The threshold table gives a mapping from these levels to the logical colours actually used to paint the character.

The format of the data read is:

Offset	Value
0	Foreground colour offset
1	1st threshold value
2	2nd threshold value
3	:
n	&FF

The table is used in the following way. Suppose you want to use eight colours for anti-aliased colours, one background colour and seven foreground colours. Thus the foreground colour offset is 6 (there are 7 colours). The table would be set up as follows:

Offset	Value
0	6
1	2
2	4
3	6
4	8
5	10
6	12
7	14
8	&FF

When this has been set-up (using Font_SetThresholds), the mapping from the 16 colours to the eight available will look like this:

Input	Output	Threshold
0	0	
1	0	
2	1	2
3	1	
4	2	4
5	2	
6	3	6
7	3	
8	4	8
9	4	
10	5	10
11	5	
12	6	12
13	6	
14	7	14
15	7	

Where the output colour is 0, the font background colour is used. Where it is in the range 1 - 7, the colour f+o–1 is used, where 'f' is the font foreground colour, and 'o' is the output colour.

You can view the thresholds as the points at which the output colour 'steps up' to the next value.

Related SWIs

Font_SetThresholds (SWI &40095), Font_SetPalette (SWI &40093), Font_SetFontColours (SWI &40092)

Related vectors

None

Font_SetThresholds (SWI &40095)

Defines the list of threshold values for painting

On entry

R1 = pointer to threshold data

On exit

R1 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets up the threshold table for a given number of foreground colours. The format of the input data, and its interpretation, is explained in the previous section.

This command should rarely be needed, because the default set will work well in most cases.

The VDU command VDU 23,25,<bits per pixel>,<threshold 1>,...,<threshold 7> is an equivalent command to this SWI. As with all VDU font commands, it has been kept for compatibility with earlier versions of the operation system and must not be used.

Related SWIs

Font_ReadThresholds (SWI &40094), Font_SetPalette (SWI &40093), Font_SetFontColours (SWI &40092)

Related vectors

None

Font_FindCaretJ (SWI &40096)

Find where the caret is in a justified string

On entry

R1 = pointer to string
R2 = x offset in millipoints
R3 = y offset in millipoints
R4 = x justification offset
R5 = y justification offset

On exit

R1 = pointer to character where the search terminated
R2 = x offset after printing string (up to termination)
R3 = y offset after printing string (up to termination)
R4 = no of printable characters in string (up to termination)
R5 = index into string giving point at which it terminated

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The 'justification offsets', R4 and R5, are calculated by dividing the extra gap to be filled by the justification of the number of spaces (ie character 32) in the string. If R4 and R5 are both zero, then this call is exactly the same as Font_FindCaret.

You should use the SWI Font_ScanString (see page 5-79) in preference to this one.

Related SWIs

Font_FindCaret (SWI &4008D), Font_ScanString (SWI &400A1)

Related vectors

None

Font_StringBBox (SWI &40097)

Measure the size of a string

On entry

R1 = pointer to string

On exit

R1 = bounding box minimum x in millipoints (inclusive)
R2 = bounding box minimum y in millipoints (inclusive)
R3 = bounding box maximum x in millipoints (exclusive)
R4 = bounding box maximum y in millipoints (exclusive)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call measures the size of a string without actually printing it. The string can consist of printable characters and all the usual control sequences. The bounds are given relative to the start point of the string (they might be negative due to backward move control sequences, etc).

Note that this command cannot be used to measure the screen size of a string because of rounding errors. The string must be scanned 'manually', by stepping along in millipoints, and using Font_ConverttoOS and Font_CharBBox to measure the precise position of each character on the screen. Usually this can be avoided, since text is formatted in rows, which are assumed to be high enough for it.

You should use the SWI Font_ScanString (see page 5-79) in preference to this one.

Related SWIs

Font_ReadInfo (SWI &40084), Font_CharBBox (SWI &4008E), Font_ScanString (SWI &400A1)

Related vectors

None

Font_ReadColourTable (SWI &40098)

Read the anti-alias colour table

On entry

R1 = pointer to 16 byte area of memory

On exit

R1 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call returns the 16 entry colour table to the block pointed to by R1 on entry. This contains the 16 colours used by the anti-aliasing software when painting text – that is, the values that would be put into screen memory.

Related SWIs

Font_SetPalette (SWI &40093), Font_SetThresholds (SWI &40095), Font_SetFontColours (SWI &40092)

Related vectors

None

Font_MakeBitmap (SWI &40099)

Make a font bitmap file

On entry

R1 = font handle, or pointer to font name
R2 = x point size × 16
R3 = y point size × 16
R4 = x dots per inch
R5 = y dots per inch
R6 = flags

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call allows a particular size of a font to be pre-stored in the font's directory so that it can be cached more quickly. It is especially useful if subpixel positioning is to be performed, since this takes a long time if done directly from outlines.

The flags have the following meanings:

Bit	Meaning when set
0	construct f9999x9999 (else b9999x9999)
1	do horizontal subpixel positioning
2	do vertical subpixel positioning
3	just delete old file, without replacing it
4 - 31	reserved (must be 0)

Once a font file has been saved, its subpixel scaling will override the setting of FontMax4/5 currently in force (so, for example, if the font file had horizontal subpixel scaling, then when a font of that size is requested, horizontal subpixel scaling will be used even if FontMax4 is set to 0).

If the font has an encoding applied to it (ie if there was a '/E' qualifier in the Font_FindFont string, or if this is a 'language' font, which varies in encoding according to the territory), then the bitmaps are held inside a subdirectory of the font directory:

<prefix>.<fontname>.<encoding>.

Note that Font_Paint also looks inside this directory to find the bitmaps.

Related SWIs

Font_SetFontMax (SWI &4009B)

Related vectors

None

Font_UnCacheFile (SWI &4009A)

Delete cached font information, or recache it

On entry

R1 = pointer to full filename of file to be removed
R2 = recache flag (0 or 1 – see below)

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

If an application such as !FontEd wishes to overwrite font files without confusing the font manager, it should call this SWI to ensure that any cached information about the file is deleted.

The filename pointed to by R1 must be the full filename (ie in the format used by the Filer), and must also correspond to the relevant name as it would have been constructed from Font$Path and the font name. This means that each of the elements of Font$Path must be proper full pathnames, including filing system prefix and any required special fields (eg. net#fileserver:$.fonts.).

The SWI must be called twice: once to remove the old version of the data, and once to load in the new version. This is especially important in the case of IntMetrics files, since the font cache can get into an inconsistent state if the new data is not read in immediately.

The 'recache' flag in R2 determines whether the new data is to be loaded in or not, and might be used like this:


```
SYS "Font_UnCacheFile",,"filename",0
…
SYS "Font_UnCacheFile",,"filename",1
```

replace old file with new one

Related SWIs

None

Related vectors

None

Font_SetFontMax (SWI &4009B)

Set the FontMax values

On entry

R0 = new value of FontMax (bytes)
R1 - R5 = new values of FontMax1 - FontMax5 (pixels × 72 × 16)
R6, R7 reserved (must be zero)

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call can be used to set the values of FontMax and FontMax1 … FontMax5. Changing the configured settings will also change these internal settings, but Font_SetFontMax does not affect the configured values, which come into effect on Ctrl-Break or when the font manager is re-initialised.

This call also causes the font manager to search through the cache, checking to see if anything would have been cached differently if the new settings had been in force at the time. If so, the relevant data is discarded, and will be reloaded using the new settings when next required.

Related SWIs

Font_ReadFontMax (SWI &4009C)

Related vectors

None

Font_ReadFontMax (SWI &4009C)

Read the FontMax values

On entry

—

On exit

R0 = value of FontMax (bytes)
R1 - R5 = values of FontMax1 - FontMax5 (pixels × 72 × 16)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call can be used to read the values of FontMax and FontMax1 ... FontMax5. It reads the values that the font manager holds internally (which may have been altered from the configured values by Font_SetFontMax).

Related SWIs

Font_SetFontMax (SWI &4009B)

Related vectors

None

Font_ReadFontPrefix (SWI &4009D)

Find the directory prefix for a given font handle

On entry

R0 = font handle
R1 = pointer to buffer
R2 = length of buffer

On exit

R1 = pointer to terminating null
R2 = bytes remaining in buffer

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call finds the directory prefix relating to a given font handle, which indicates where the font's IntMetrics file is, and copies it into the buffer pointed to by R1; for example:

```
adfs::4.$.!Fonts.Trinity.Medium.
```

One use for this prefix would be to find out which sizes of a font were available pre-scaled in the font directory.

Related SWIs

None

Related vectors

None

Font_SwitchOutputToBuffer (SWI &4009E)

Switches output to a buffer, creating a Draw file structure

On entry

R0 = flags if R1 > 0, else reserved (must be zero)
R1 = pointer to word-aligned buffer, or:
8 initially to count the space required for a buffer
0 to switch back to normal
–1 to leave state unaltered (ie enquire about current status)

On exit

R0 = previous flag settings
R1 = previous buffer pointer, incremented by space required for Draw file structure

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

After this call, any calls to Font_Paint will be redirected into the buffer, as a Draw file structure.

Each letter painted will be treated as a separate filled object, with the colours specified in the paint command.

The flags in R0 have the following meaning:

Bit	Meaning when set
0	update R1, but don't store anything
1	apply 'hints' to the outlines
4	give error if bitmapped characters occur (this bit overrides bit 3)

All other bits are reserved, and must be zero.

This call is not available in RISC OS 2.0.

On entry, the buffer must contain the following if it is to receive output:

Size	Contents
4	0 (null terminator)
4	size remaining, in bytes

The Draw file structure is placed in the file before the null terminator, between (original R1) and (final R1 – 1). R1 still points to the null terminator; the terminator and free space count do not form part of the output data itself.

If bit 0 of R0 is set, output is not actually sent to the buffer, but the pointer is updated. This allows the size of the required buffer to be computed properly before allocating the space for it. Note that if bit 0 of R0 is set, R1 must initially be greater than 0 (a value of 8 is recommended, since the buffer must allow 8 bytes for the terminator and free space counter).

The rubout box(es) and any underlining are also sent to the buffer as a series of filled outlines. These will be in the correct order so as to be behind any characters which overlap them. The output will also take into account matrix transformations, font and colour changes, explicit movements, justification and kerning.

If bit 1 of R0 is set, the character outlines have hints applied to them at the current size. This means that they are not really suitable for scaling later on.

If bit 2 of R0 is set, the character objects consist of a group of two objects: the filled outline, and the stroked skeleton.

Any characters which are only available as bitmaps will either generate an error (if bit 4 of R0 is set), be ignored (if bit 3 of R0 is clear), or represented as bitmap objects in the output (either 1-bpp or 4-bpp, with a palette to match the output colours).

In this way drawing programs can turn on buffering, then proceed to draw text in the appropriate position and size, and end up with a series of Draw objects which represent the same thing. The set of objects that the Font Manager produces could easily be converted into a group by wrapping them suitably.

Related SWIs

None

Related vectors

None

Font_ReadFontMetrics (SWI &4009F)

On entry

R0 = font handle
R1 = pointer to buffer for bounding box information, or 0 to read size of data
R2 = pointer to buffer for x-width information, or 0 to read size of data
R3 = pointer to buffer for y-width information, or 0 to read size of data
R4 = pointer to buffer for miscellaneous information, or 0 to read size of data
R5 = pointer to buffer for kerning information, or 0 to read size of data
R6 = 0
R7 = 0

On exit

R0 = file flags
R1 - R5 = size of data (0 if not present in file)
R6, R7 undefined

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call reads the new metrics information held in a font's IntMetrics file.

The flags in R0 have the following meaning:

Bit	Meaning when set
1	kern pairs don't have x-offsets
2	kern pairs don't have y-offsets
3	there are more than 255 kern pairs

All other bits are reserved, and must be zero.

Currently this call is not permitted on fonts which have a transformation matrix applied to them. It is recommended that the call is made on the untransformed version of the font, and the results then transformed appropriately. Note that when transforming bounding boxes, the resulting box is that which bounds all 4 transformed bounding box corners. When transforming x and y offsets (ie character widths), the last 2 numbers in the matrix (the offsets) should be ignored, since the new origin is also moved by these amounts, and they therefore cancel out.

This call is not available in RISC OS 2.0.

The format of the data in the buffers is as follows. Except where otherwise stated:

- all units are millipoints (1/72000")
- all 2-byte and 4-byte numbers are little-endian, signed

Bounding box information

array[256] of groups of 4 words (x0, y0, x1, y1)

X-width information

array[256] of words

Y-width information

array[256] of words

Miscellaneous information

Size	Description
2	x0 maximum bounding box for font (16-bit signed)
2	y0 bottom-left (x0, y0) is inclusive
2	x1 top-right (x1, y1) is exclusive
2	y1 all coordinates are in millipoints
2	default x-offset per char (if *flags* bit 1 is set), in millipoints (16-bit signed)
2	default y-offset per char (if *flags* bit 2 is set), in millipoints (16-bit signed)
2	italic h-offset per em (−1000 × TAN(italic angle)) (16-bit signed)
1	underline position, in 1/256th em (signed)
1	underline thickness, in 1/256th em (unsigned)
2	CapHeight in millipoints (16-bit signed)
2	XHeight in millipoints (16-bit signed)
2	Descender in millipoints (16-bit signed)
2	Ascender in millipoints (16-bit signed)
4	reserved (must be zero)

Kerning information

The kerning information is indexed by a hash table. The hash function used is:

(first letter) EOR (second letter ROR 4)

where the rotate happens in 8 bits.

Size	Description
256 × 4	hash table giving offset from table start of first kern pair for each possible value (0 - 255) of hash function
4	offset of end of all kern pairs from table start
4	flag word: bit 0 set ⇒ no bounding boxes bit 1 set ⇒ no x-offsets bit 2 set ⇒ no y-offsets bits 3 - 30 reserved (ignore these) bit 31 set ⇒ 'short' kern pairs
?	kern pair data

Each kern pair consists of the code of the first letter of the kern pair, followed by the x-offset in millipoints (if flags bit 1 is clear) and the y-offset in millipoints (if flags bit 2 is clear).

If bit 31 of the flag word is clear, then the letter code, x-offset and y-offset are each held in a word. If bit 31 is set, then the kern pair data is shortened by combining the letter code with the first offset word as follows:

bits 0 - 7 = character code
bits 8 - 31 = x or y-offset

If necessary, the second letter can be deduced from the first letter and the hash index as follows:

2nd letter = (1st letter EOR hash table index) ROR 4

where the rotate happens in 8 bits.

The hash table indicates the point at which to start looking for a given kern pair in the list of kern pairs following the table. The entries are consecutive, so each list finishes as the next one starts. To search for a given kern pair:

1. Work out the value *n* of the hash function
2. Look up the *n*th and (*n*+1)th offsets in the hash table

3 Search for a kern pair having the correct 1st letter, looking from the *n*th offset up to – but not including – the (*n*+1)th offset.

Once the kern offsets are obtained, they can be inserted into a Font_Paint string as character 9 and 11 move sequences.

Note that if flag bits 1 and 2 are both set, then it is illegal for there to be any kern pairs.

Related SWIs

None

Related vectors

None

Font_DecodeMenu (SWI &400A0)

Decode a selection made from a font menu

On entry

R0 = flags:
- bit 0 set ⇒ encoding menu, else font menu
- all other bits reserved (must be zero)

R1 = pointer to menu definition (as returned by Font_ListFonts)
R2 = pointer to menu selections (as returned by Wimp_Poll with reason code = 9)
R3 = pointer to buffer to contain answer (0 ⇒ just return size)
R4 = size of buffer (if R3 ≠ 0)

On exit

R0, R1 preserved
R2 = pointer to rest of menu selections (if R3 ≠ 0 on entry)
R3 preserved
R4 = size of buffer required to hold output string (0 ⇒ no font selected)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call decodes a selection (as returned from Wimp_Poll) made from a font menu. The definition of the font menu is passed in the same format as returned from Font_ListFonts.

This call is not available in RISC OS 2.0.

Bit 0 of R0 determines whether it is the font menu or the encoding menu that is being decoded. In either case, the format of the returned string depends on whether the names of the fonts/encodings have been specified in a Messagesn file inside the font directory. The name field is not present if the Font Manager has worked out the list of fonts/encodings by scanning the directory instead.

File holds:	Format of returned string:
Font id, no name	\F<font id>
Font id, with name	\F<font id>\f<territory> <font name>
Encoding, no name	\E<encoding id>
Encoding, with name	\E<encoding id>\e <territory> <encoding name>

Since Font_DecodeMenu works by comparing the string in the menu against the Font Manager's known font names, in the case of 'System font' being selected from a menu that contained it, R4 would be returned as 0. To distinguish this from the 'no font selected' case, check for R2 pointing to 0 on entry, since 'System font' is always the first menu entry if present.

Related SWIs

None

Related vectors

None

Font_ScanString (SWI &400A1)

Return information on a string

On entry

R0 = initial font handle (1 - 255) or 0 for current handle – if bit 8 of R2 is set
R1 = pointer to string
R2 = plot type:
- bits 0 - 4 reserved (must be zero)
- bit 5 set ⇒ use R5 as indicated below
- bit 6 set ⇒ use R6 as indicated below
- bit 7 set ⇒ use R7 as indicated below
- bit 8 set ⇒ use R0 as indicated above
- bit 9 set ⇒ perform kerning on the string
- bit 10 set ⇒ writing direction is right to left; else left to right
- bits 11 - 16 reserved (must be zero)
- bit 17 set ⇒ return nearest caret position; else length of string
- bit 18 set ⇒ return bounding box of string in buffer pointed to by R5
- bit 19 set ⇒ return matrix applying at end of string in buffer pointed to by R6
- bit 20 ⇒ return number of split characters in R7
- bits 21 - 31 reserved (must be zero)

R3, R4 = offset of mouse click – if bit 17 of R2 is set; else maximum x, y-coordinate offset before split point
R5 = pointer to buffer used on entry for coordinate block and split character – if bit 5 of R2 is set – and on exit for returned bounding box– if bit 18 of R2 is set
R6 = pointer to buffer used on entry for transformation matrix – if bit 6 of R2 is set – and on exit for returned transformation matrix– if bit 19 of R2 is set
R7 = length of string – if bit 7 of R2 is set

On exit

R1 = pointer to point in string of caret position – if bit 17 of R2 is set; else to split point
R2 preserved
R3, R4 = x, y-coordinate offset to caret position – if bit 17 of R2 is set; else to split point
R5, R6 preserved

R7 = number of split characters encountered – if bit 20 of R2 was set; else preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call scans a string as if it were painted to the screen using Font_Paint, and returns various information about it. It is particularly useful for finding the correct position of the caret within a string, or for finding where to split a line.

For full details of the parameters passed, and of control sequences that may be included in the string, you should see the description of Font_Paint on page 5-24. Below we merely describe the changes and additions relative to that SWI.

This call is not available in RISC OS 2.0.

Coordinates

Unlike Font_Paint, this call uses millipoints for all coordinates; you may not specify OS units by setting bit 4 of R2.

R3 and R4 do not specify the start coordinates of the string. Instead they specify either the offset from the start of the string to the mouse click (used to work out where to insert the caret), or the maximum offset before the split point (ie the width and height remaining on the current line).

On exit R3 and R4 give the offset of the caret position or the split point. When scanning to determine the split point, the scan continues until the current offset is less than or greater than the limit supplied, depending on the sign of that limit. If R3 is negative on entry, the scan continues until the x-offset is less than R3, while if R3 is positive, the scan continues until the x-offset is greater than R3. Note that this is incompatible with the old Font_StringWidth call, which always continued until the x- and y-offsets were greater than R2 or R3. (Font_StringWidth still works in the old way, to ensure compatibility).

Graphics cursor coordinates

Font_ScanString does not use graphics cursor coordinates for justification, nor to specify a rubout box. Justification can still be performed using the coordinate block pointed to by R5, whereas rubout boxes are not supported at all.

The coordinate block and split character

The coordinate block pointed to by R5 differs from that used by Font_Paint in that no rubout box is given. Instead the word at offset 16 is used to specify the 'split character' on entry.

The four following words (ie starting at offset 20) are used to return the string's bounding box, if bit 18 of R2 is set on entry. This excludes the area occupied by underlining or rubout

Offset	Value
0	additional x, y offset on space
8	additional x, y offset between each letter
16	split character (-1 ⇒ none)
20	returned x, y coordinates for bottom left of string bounding box (inclusive) – if bit 18 of R2 is set
28	returned x, y coordinates for top right of string bounding box (exclusive) – if bit 18 of R2 is set

If there is no split character, but bit 20 of R2 is set ('return number of split characters in R7'), then R7 will instead be used to return the number of non-control characters encountered (ie those characters with codes of 32 or more which are not part of a control sequence).

Transformation matrices

If bit 19 of R2 is set on entry, the transformation matrix pointed to by R6 is updated on exit to return the matrix applying at the end of the string.

Text direction

Where bit 10 is set (ie the main writing direction is right to left), one would normally supply a negative value of R3.

String length

Note that the character at [R1,R7] may be accessed to determine whether it is a 'split character', as well as to determine the character offset due to kerning.

Related SWIs

This SWI replaces the following deprecated (still supported, but not recommended) SWIs:

Font_StringWidth (SWI &40085), Font_FindCaret (SWI &4008D), Font_FindCaretJ (SWI &40096), Font_StringBBox (SWI &40097)

Related vectors

None

Font_SetColourTable (SWI &400A2)

This call is for internal use by the ColourTrans module only. **You must not use it** in your own code.

This call is not available in RISC OS 2.0.

To set font colours you should either use ColourTrans_SetFontColours (see page 4-412) or Font_Paint control sequence 19 (see page 5-27).

Font_CurrentRGB (SWI &400A3)

Reads the settings of colours after calling Font_Paint

On entry

—

On exit

R0 = font handle
R1 = background font colour (&BBGGRR00)
R2 = foreground font colour (&BBGGRR00)
R3 = maximum colour offset (0 ⇒ mono, else anti-aliased)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call reads the settings of the RGB foreground and background colours after calling Font_Paint.

This call is not available in RISC OS 2.0.

The error 'Undefined RGB font colours' is generated if the colours were not set using RGB values.

Related SWIs

None

Related vectors

None

Font_FutureRGB (SWI &400A4)

Reads the settings of colours after calling various Font... SWIs

On entry

—

On exit

R0 = font handle
R1 = background font colour (&BBGGRR00)
R2 = foreground font colour (&BBGGRR00)
R3 = maximum colour offset (0 ⇒ mono, else anti-aliased)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call reads the settings of the RGB foreground and background colours after calling Font_ScanString, Font_StringWidth, Font_StringBBox, Font_FindCaret or Font_FindCaretJ.

This call is not available in RISC OS 2.0.

The error 'Undefined RGB font colours' is generated if the colours were not set using RGB values.

Related SWIs

None

Related vectors

None

Font_ReadEncodingFilename (SWI &400A5)

Returns the filename of the encoding file used for a given font handle

On entry

R0 = font handle
R1 = pointer to buffer to receive prefix
R2 = length of buffer

On exit

R0 = pointer to encoding filename (in buffer)
R1 = pointer to terminating 0 of filename
R2 = bytes remaining in buffer

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call returns the filename of the encoding file used for a given font handle. It is primarily useful for PDriverPS to gain access to the file of identifiers that defines an encoding, in order to send it to the printer output stream.

The filename depends on whether the font has a 'public' or 'private' encoding (public encodings apply to 'language' fonts, as described in Font_ListFonts, while private encodings are not used by the Font Manager, and simply describe the PostScript names for the characters in the font).

Encoding	Filename
public	*font_prefix*.Encodings.*encoding*
private	*font_prefix*.*font_name*.Encoding

The error 'Buffer overflow' is generated if the buffer is too small.

This call is not available in RISC OS 2.0.

Related SWIs

None

Related vectors

None

*Commands

*Configure FontMax

Sets the configured maximum size of the font cache

Syntax

`*Configure FontMax mK|n`

Parameters

*m*K	number of kilobytes of memory reserved
n	number of 4k chunks of memory reserved

Use

*Configure FontMax sets the configured maximum size of the font cache. The difference between FontSize and FontMax is the extra amount of memory that the font manager will attempt to use if it needs to. If other parts of the system have already claimed all the spare memory, then FontSize is what it is forced to work with.

If FontMax is bigger than FontSize, when the font manager cannot obtain enough cache memory it will attempt to expand the cache by throwing away unused blocks (ie ones that belong to fonts which have had Font_FindFont called on them more often than Font_LoseFont). Once the cache has expanded up to FontMax, the font manager will throw away the oldest block found, even if it is in use. This can result in the font manager heavily using the filing system, since during a window redraw it is possible that all fonts will have to be thrown away and recached in turn.

Example

`*Configure FontMax 256K`

Related commands

*Configure FontSize

Related SWIs

Font_CacheAddr (SWI &40080), Font_SetFontMax (SWI &4009B), Font_ReadFontMax (SWI &4009C)

Related vectors

None

*Configure FontMax1

Sets the maximum height at which to scale from a bitmap font

Syntax

`*Configure FontMax1 max_height`

Parameters

max_height	maximum font pixel height at which to scale from a bitmap font

Use

*Configure FontMax1 sets the maximum height at which to scale from a bitmap font rather than from an outline font – but only if 4 bit per pixel output is possible.

When the font manager can use 4 bits per pixel, it first looks for an f9999x9999 file of the correct size; then it looks for an x90y45 font of the correct size. Next it considers the values of FontMax2 and 3, and then of FontMax4 and 5. Only if the above fail to produce output does it then consider the value of FontMax1:

- If the font pixel height is less than or equal to the value specified in FontMax1, or if there is no Outlines file, the font manager looks for the x90y45 file to determine which bitmap font to scale. If the x90y45 file contains the name of an f9999x9999 file, then that file is scaled; else one of the fonts in the x90y45 file is scaled.
- Otherwise the font manager scales the Outlines file to give an anti-aliased (4 bits per pixel) bitmap.

The height is set in pixels rather than points because it is the pixel size that affects cache usage. This corresponds to different point sizes on different resolution output devices:

pixel height = height in points × pixels (or dots) per inch / 72

Example

`*Configure FontMax1 25`

Related commands

*Configure FontMax2

Related SWIs

Font_SetFontMax (SWI &4009B), Font_ReadFontMax (SWI &4009C)

Related vectors

None

*Configure FontMax2

Sets the maximum height at which to scale from outlines to anti-aliased bitmaps

Syntax

`*Configure FontMax2 max_height`

Parameters

max_height	maximum font pixel height at which to scale from outlines to anti-aliased bitmaps

Use

*Configure FontMax2 sets the maximum height at which to scale from outlines to anti-aliased bitmaps, rather than to 1 bit per pixel bitmaps.

When the font manager can use 4 bits per pixel, it first looks for an f9999x9999 file of the correct size; then it looks for an x90y45 font of the correct size. Only if the above fail to produce output does it then consider the value of FontMax2:

- If the font pixel height is less than or equal to the heights specified in both FontMax2 and 3, the font manager goes on to consider the values of FontMax4 and 5, and then of FontMax1. Any bitmaps it produces from outlines will be anti-aliased.
- Otherwise, the font manager uses 1 bit per pixel bitmaps. It first looks for a b9999x9999 file of the correct size.

  If it fails to find one it uses the Outlines file to paint a 1-bit-per-pixel bitmap. The value of FontMax3 determines whether the font manager caches the bitmap or the outline.

The height is set in pixels rather than points because it is the pixel size that affects cache usage. This corresponds to different point sizes on different resolution output devices:

pixel height = height in points x pixels (or dots) per inch / 72

Example

`*Configure FontMax2 20`

Related commands

*Configure FontMax1, *Configure FontMax3

Related SWIs

Font_SetFontMax (SWI &4009B), Font_ReadFontMax (SWI &4009C)

Related vectors

None

*Configure FontMax3

Sets the maximum height at which to retain bitmaps in the cache

Syntax

`*Configure FontMax3 max_height`

Parameters

max_height	maximum font pixel height at which to retain bitmaps in the cache

Use

*Configure FontMax3 sets the maximum height at which to retain bitmaps in the cache, rather than the outlines from which they were converted.

Unlike the other FontMax*n* values, FontMax3 affects the font manager both when it can use 4 bits per pixel, and when it can only use 1 bit per pixel.

4 bits per pixel

When the font manager can use 4 bits per pixel, it first looks for an f9999x9999 file of the correct size; then it looks for an x90y45 font of the correct size. Only if the above fail to produce output does it then consider the value of FontMax3:

- If the font pixel height is less than or equal to the heights specified in both FontMax2 and 3, the font manager goes on to consider the values of FontMax4 and 5, and then of FontMax1. Any bitmaps it produces will be cached.

  Otherwise, the font manager first looks for a b9999x9999 file of the correct size. If it fails to find one it uses the Outlines file to paint a 1-bit-per-pixel bitmap. The value of FontMax3 determines whether the font manager caches the bitmap or the outline:

  - If the font pixel height is less than or equal to the height specified in FontMax3, the font manager retains the resultant bitmap in the cache.
  - If the font pixel height is greater than the height specified in FontMax3, the font manager will not cache the bitmaps, but will instead cache the outlines themselves.

    It draws the outlines directly onto the destination using the Draw module; consequently they are not anti-aliased. The font manager sets up the appropriate GCOL and TINT settings for this, and resets them afterwards.

1 bit per pixel

If the font manager can only use 1 bit per pixel, it first looks for a b9999x9999 file of the correct size.

If it fails to find one it looks for the Outlines file, scaling it to give a 1-bit-per-pixel bitmap. The value of FontMax3 determines whether the font manager caches the bitmap or the outline:

- If the font pixel height is less than or equal to the height specified in FontMax3, the font manager retains the resultant bitmap in the cache.
- If the font pixel height is greater than the height specified in FontMax3, the font manager will not cache the bitmaps, but will instead cache the outlines themselves.

  It draws the outlines directly onto the destination using the Draw module; consequently they are not anti-aliased. The font manager sets up the appropriate GCOL and TINT settings for this, and resets them afterwards.

If there is no Outlines file, the font manager then looks for an f9999x9999 file of the correct size; then it looks for an x90y45 font of the correct size. Finally it uses the x90y45 file to determine which bitmap font to scale. If the x90y45 file contains the name of an f9999x9999 file, then that file is scaled; else one of the fonts in the x90y45 file is scaled.

The height is set in pixels rather than points because it is the pixel size that affects cache usage. This corresponds to different point sizes on different resolution output devices:

pixel height = height in points × pixels (or dots) per inch / 72

Example

```
*Configure FontMax3 35
```

Related commands

*Configure FontMax2

Related SWIs

Font_SetFontMax (SWI &4009B), Font_ReadFontMax (SWI &4009C)

Related vectors

None

*Configure FontMax4

Sets the maximum width at which to use horizontal subpixel anti-aliasing

Syntax

```
*Configure FontMax4 max_width
```

Parameters

max_width	maximum font pixel width at which to use horizontal subpixel anti-aliasing

Use

*Configure FontMax4 sets the maximum width at which to use horizontal subpixel anti-aliasing.

When the font manager can use 4 bits per pixel, it first looks for an f9999x9999 file of the correct size (note that this bitmap may have been constructed with subpixel anti-aliasing already performed – see Font_MakeBitmap); then it looks for an x90y45 font of the correct size. Next it considers the values of FontMax2 and 3. Only if the above fail to produce output does it then consider the value of FontMax4 and 5:

- If the font pixel width is less than or equal to the width specified in FontMax4, the font manager will look for the Outlines file, and will construct 4 anti-aliased bitmaps for each character, corresponding to 4 possible horizontal subpixel alignments on the screen.

  Likewise, if the font pixel height is less than or equal to the height specified in FontMax5, the font manager will perform vertical subpixel anti-aliasing. Thus if both horizontal and vertical subpixel anti-aliasing occurs, 16 bitmaps will be constructed.

  When painting the text, the font manager will use the bitmap which corresponds most closely to the required alignment
- Otherwise the font manager goes on to consider the value of FontMax1; it will not use subpixel anti-aliasing.

The width is set in pixels rather than points because it is the pixel size that affects cache usage. This corresponds to different point sizes on different resolution output devices:

pixel width = width in points × pixels (or dots) per inch / 72

Example

```
*Configure FontMax4 0
```

Related commands

*Configure FontMax5

Related SWIs

Font_SetFontMax (SWI &4009B), Font_ReadFontMax (SWI &4009C)

Related vectors

None

*Configure FontMax5

Sets the maximum height at which to use vertical subpixel anti-aliasing

Syntax

*Configure FontMax5 *max_height*

Parameters

max_height	maximum font pixel height at which to use vertical subpixel anti-aliasing

Use

*Configure FontMax5 sets the maximum height at which to use vertical subpixel anti-aliasing.

When the font manager can use 4 bits per pixel, it first looks for an f9999x9999 file of the correct size (note that this bitmap may have been constructed with subpixel anti-aliasing already performed – see Font_MakeBitmap); then it looks for an x90y45 font of the correct size. Next it considers the values of FontMax2 and 3. Only if the above fail to produce output does it then consider the value of FontMax4 and 5:

- If the font pixel height is less than or equal to the height specified in FontMax5, the font manager will look for the Outlines file, and will construct 4 anti-aliased bitmaps for each character, corresponding to 4 possible vertical subpixel alignments on the screen.

  Likewise, if the font pixel width is less than or equal to the width specified in FontMax4, the font manager will perform horizontal subpixel anti-aliasing. Thus if both vertical and horizontal subpixel anti-aliasing occurs, 16 bitmaps will be constructed.

  When painting the text, the font manager will use the bitmap which corresponds most closely to the required alignment
- Otherwise the font manager goes on to consider the value of FontMax1; it will not use subpixel anti-aliasing.

The width is set in pixels rather than points because it is the pixel size that affects cache usage. This corresponds to different point sizes on different resolution output devices:

pixel width = width in points × pixels (or dots) per inch / 72

Example

```
*Configure FontMax4 0
```

Related commands

*Configure FontMax5

Related SWIs

Font_SetFontMax (SWI &4009B), Font_ReadFontMax (SWI &4009C)

Related vectors

None

*Configure FontSize

Sets the configured amount of memory reserved for the font cache

Syntax

*Configure FontSize *size*K

Parameters

size	number of kilobytes to allocate

Use

*Configure FontSize sets the configured amount of memory reserved for the font cache. This is claimed when the font manager is first initialised. If insufficient memory is free, the font manager starts running using what is available.

The font manager will never shrink its cache below this configured size.

The minimum cache size can also be changed from the Task Manager, by dragging the font cache bar directly, although this is not remembered after a Control-reset.

Example

```
*Configure FontSize 32K
```

Related commands

*Configure FontMax

Related SWIs

Font_CacheAddr (SWI &40080)

Related vectors

None

*FontCat

Lists the fonts available in a directory

Syntax

`*FontCat [directory]`

Parameters

directory pathname of a directory to search for fonts

Use

*FontCat lists the fonts available in the given directory. If no directory is given, then the directory specified in the system variable Font$Path is used.

Font_FindFont uses the same variable when it searches for a font.

Example

```
*FontCat adfs:$.Fonts.
Corpus.Medium
Portrhouse.Standard
Trinity.Medium
```

The last '.' is essential

Related commands

None

Related SWIs

Font_FindFont (SWI &40081), Font_ListFonts (SWI &40091)

Related vectors

None

*FontInstall

Adds a directory to the list of those scanned for fonts

Syntax

`*FontInstall [directory]`

Parameters

directory pathname of a directory to add to Font$Path

Use

*FontInstall adds a directory to the list of those scanned for fonts. It does so by altering the system variable Font$Path so that the given pathname appears before any others, and is not repeated. It also rescans the directory, even if it was already known to the Font Manager.

If no pathname is given, all directories in Font$Path are rescanned.

Service_FontsChanged is issued whenever a directory is scanned.

This command is not available in RISC OS 2.0.

Example

```
*FontInstall RAM:$.Fonts.
```

The last '.' is essential

Related commands

*FontRemove

Related SWIs

None

Related vectors

None

*FontLibrary

Sets a directory as the font library, replacing the previous library

Syntax

`*FontLibrary directory`

Parameters

`directory`	a valid pathname specifying a directory

Use

*FontLibrary sets a directory as the font library, replacing the previous library in the list of those scanned for fonts. It does so by altering the system variable Font$Prefix to the given directory, and ensures that the string '<Font$Prefix>.' appears on the front of the system variable Font$Path.

Note however that if the previous font library had also been explicitly added to Font$Path (say by *FontInstall), it will still be scanned.

This command is not available in RISC OS 2.0.

Example

```
*FontLibrary scsifs::MyDisc.$.FontLib
```

Related commands

None

Related SWIs

None

Related vectors

None

*FontList

Displays the fonts in the font cache, its size, and its free space

Syntax

`*FontList`

Parameters

None

Use

*FontList displays the fonts currently in the font cache. For each font, details are given of its point size, its resolution, the number of times it is being used by various applications, and the amount of memory it is using.

The size of the font cache and the amount of free space (in Kbytes) is also given.

Example

```
*FontList
     Name             Size        Dots/inch   Use    Memory
     ----             ----        ---------   ---    ------
 1.Trinity.Medium     12 point    90x45        0     3 Kbytes
 2.Corpus.Standard    14 point    90x45        2     11 Kbytes

Cache size:  24 Kbytes
      free:   9 Kbytes
```

Related commands

None

Related SWIs

Font_ListFonts (SWI &40091)

Related vectors

None

*FontRemove

Removes a directory from the list of those scanned for fonts

Syntax

```
*FontRemove [directory]
```

Parameters

directory	pathname of a directory to remove from Font$Path

Use

*FontRemove removes a directory from the list of those scanned for fonts. It does so by removing the given pathname from the system variable Font$Path.

This command is not available in RISC OS 2.0.

Example

```
*FontRemove RAM:$.Fonts.
```

The last '.' is essential

Related commands

*FontInstall

Related SWIs

None

Related vectors

None

*LoadFontCache

Loads a file back into the font cache

Syntax

```
*LoadFontCache filename
```

Parameters

filename	a valid pathname specifying a file previously saved using *SaveFontCache

Use

*LoadFontCache loads a file back into the font cache that was previously saved using *SaveFontCache.

An error is generated if any fonts are currently claimed, or if the font cache format cannot be read by the current font manager (ie it was created by a version of the font manager that used an incompatible font cache format).

The size of the font cache slot will – if necessary – be increased to accommodate the new cache data; but it will not be decreased, even if the new cache data is smaller than the current cache slot size.

This command is useful for setting up the font cache to a predefined state, to save time scaling fonts later on.

This command is not available in RISC OS 2.0.

Example

```
*LoadFontCache scsifs::MyDisc.$.FontCache
```

Related commands

*SaveFontCache

Related SWIs

None

Related vectors

None

*SaveFontCache

Saves the font cache to a file

Syntax

`*SaveFontCache filename`

Parameters

filename	a valid pathname specifying a file

Use

*SaveFontCache saves the current contents of the font cache, with certain extra header information, to a file of type &FCF (FontCache). The Run alias for this filetype executes *LoadFontCache, which loads the file back into the font cache.

This command is not available in RISC OS 2.0.

Example

```
*SaveFontCache scsifs::MyDisc.$.FontCache
```

Related commands

*LoadFontCache

Related SWIs

None

Related vectors

None

Application Notes

BASIC example of justified text

```
100 SYS "Font_FindFont",,"Trinity.Medium",320,320,0,0 TO HAN%
110 REM sets font handle
120 SYS "Font_SetPalette",,8,9,6,&FFFFFF00,&00000000
130 REM Set the palette to use colours 8-15 as white to black
140 MOVE 800,500
150 REM Set the right hand side of justification
160 SYS "Font_Paint",,"This is a test",&11,0,500
170 SYS "Font_LoseFont",HAN%
```

On line 160, Font_Paint is being told to use OS coordinates and justify, starting at location 0,500. 800,500 has been declared as the right hand side of justification by line 140.

56 Draw module

Introduction

The Draw module is an implementation of PostScript type drawing. A collection of moves, lines, and curves in a user-defined coordinate system are grouped together and can be manipulated as one object, called a path.

A path can be manipulated in memory or upon writing to the VDU. There is full control over the following characteristics of it:

- rotation, scaling and translation of the path
- thickness of a line
- description of dots and dashes for a line
- joins between lines can be mitred, round or bevelled
- the leading or trailing end of a line, or dot (which are in fact just very short dashes), can be butt, round, a projecting square or triangular (used for arrows)
- filling of arbitrary shapes
- what the fill considers to be interior

A path can be displayed in many different ways. For example, if you write a path that draws a petal, and draw it several times rotating about a point, you will have a flower. This uses only one of the characteristics that you can control.

The Draw application was written using this module, and this is the kind of application that it is suited to. It is advisable to read the section on Draw in the User Guide to familiarise yourself with some of the properties of the Draw module.

Overview

There are many specialised terms used within the Draw module. Here are the most important ones. If you are familiar with PostScript, then many of these should be the same.

- A *path element* is a sequence of words. The first word in the sequence has a command number, called the *element type*, in the bottom byte. Following this are parameters for that element type.
- A *subpath* is a sequence of path elements that defines a single connected polygon or curve. The ends of the subpath may be connected, so it forms a loop (in which case it is said to be *closed*) or may be *loose ends* (in which case it is said to be *open*). A subpath can cross itself or other subpaths in the same path. See below for a more detailed explanation of when a subpath is open or closed.
- A *path* is a sequence of subpaths and path elements.
- A *Bezier curve* is a type of smooth curve connecting two *endpoints*, with its direction and curvature controlled by two *control points*.
- *Flattening* is the process of converting a Bezier curve into a series of small lines when outputting.
- *Flatness* is how closely the lines will approximate the original Bezier curve.
- A *transformation matrix* is the standard mathematical tool for two-dimensional transformations using a three by three array. It can rotate, scale and translate (move).
- To *stroke* means to draw a thickened line centred on a path.
- A *gap* is effectively a transparent line segment in a subpath. If the subpath is stroked, the piece around the gap will not be plotted. Gaps are used by Draw to implement dashed lines.
- *Line caps* are placed at the ends of an open subpath and at the ends of dashes in a dashed line when they are stroked. They can be *butt*, *round*, a *projecting square* or *triangular*.
- *Joins* occur between adjacent lines, and between the start and end of a closed subpath. They can be *mitred*, *round* or *bevelled*.
- To *Fill* means to draw everything inside a path.
- *Interior* pixels are ones that are filled. *Exterior* pixels are not filled.
- A *winding number rule* is the rule for deciding what is interior or exterior to a path when filling. The interior parts are those that are filled.
- *Boundary* pixels are those that would be drawn if the line were stroked with minimum thickness for the VDU.

- *Thickening* a path is converting it to the required thickness – that is generating a path which, if filled, would produce the same results as stroking the original path.

Scaling systems

This is an area where you must take great care when using the Draw module, because four different systems are used in different places.

OS units

OS units are notionally 1/180th of an inch, and are the standard units used by the VDU drivers for specifying output to the screen.

This coordinate system is (not surprisingly) what the Draw module uses when it strokes a path onto the screen.

Internal Draw units

Internally, Draw uses a coordinate system the units of which are 1/256th of an OS unit. We shall call these internal Draw units.

In a 32 bit internal Draw number, the top 24 bits are the number of OS units, and the bottom 8 bits are the fraction of an OS unit.8 fixed point system.

User units

The coordinates used in a path can be in any units that you wish to use. These are translated by the transformation matrix into internal Draw units when generating output.

Note that because it is a fixed point system, scaling problems can occur if the range is too far from the internal Draw units. Because of this problem, you are limited in the range of user units that you can use.

Transform units

Transform units are only used to specify some numbers in the transformation matrix. They divide a word into two parts: the top two bytes are the integer part, and the bottom two bytes are the fraction part.

Transformation matrix

This is a three by three matrix that can be used to rotate, scale or translate a path in a single operation. It is laid out like this:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

This matrix transforms a coordinate (x, y) into another coordinate (x', y') as follows:

$$x' = ax + cy + e$$
$$y' = bx + dy + f$$

The common transformations can all be easily done with this matrix. Translation by a given displacement is done by e for the x axis and f for the y axis. Scaling the x axis uses a, while the y axis uses d. Rotation can be performed by setting $a = \cos(\theta)$, $b = \sin(\theta)$, $c = -\sin(\theta)$ and $d = \cos(\theta)$, where θ is the angle of rotation.

a, b, c and d are given in transform units to allow accurate specification of the fractional part. e and f are specified in internal Draw units, so that the integer part can be large enough to adequately specify displacements on the screen. (Were transform units to be used for these coefficients, then the maximum displacement would only be 256 OS units, which is not very far on the screen.)

Winding rules

The winding rule determines what the Draw module considers to be interior, and hence filled.

Even-odd roughly means that an area is filled if it is enclosed by an even number of subpaths. The effect of this is that you will never have two adjacent areas of the same state, ie filled or unfilled.

Non-zero winding fills areas on the basis of the direction in which the subpaths which surround the area were constructed. If an equal number of subpaths in each direction surround the area, it is not filled, otherwise it is.

The positive winding rule will fill an area if it is surrounded by more anti-clockwise subpaths than clockwise. The negative winding rule works in reverse to this.

Even-odd and non-zero winding are printer driver compatible, whereas the other two are not. If you wish to use the path with a printer driver, then bear this in mind.

Stroking and filling

Flattening means bisecting any Bezier curves recursively until each of the resulting small lines lies within a specified distance of the curve. This distance is called flatness. The longer this distance, the more obvious will be the straight lines that approximate the curve.

All moving and drawing is relative to the VDU graphics origin (as set by VDU 29,x;y;).

None of the Draw SWIs will plot outside the boundaries of the VDU graphics window (as set by VDU 24,l;b;r;t;).

All calls use the colour (both pixel pattern and operation) set up for the VDU driver. Note that not all such colours are compatible with printer drivers.

Printing

If your program needs to generate printer output, then it is very important that you read the chapter entitled *Printer Drivers* on page 5-141. The Draw SWIs that are affected by printing have comments in them about the limitations and effects.

Floating point

SWI numbers and names have been allocated to support floating point Draw operations. In fact for every SWI described in this chapter, there is an equivalent one for floating point – just add FP to the end of each name.

The floating point numbers used in the specification are IEEE single precision floating point numbers.

They may be supported in some future version of RISC OS, but if you try to use them in current versions you'll get an error back.

Technical Details

Data structures

Many common structures are used by Draw module SWIs. Rather than duplicate the descriptions of these in each SWI, they are given here. Some SWIs have small variations which are described with the SWI.

Path

The path structure is a sequence of subpaths, each of which is a sequence of elements. Each element is from one to seven words in length. The lower byte of the first word is the element type. The remaining three bytes of it are free for client use. On output to the input path the Draw module will leave these bytes unchanged. However, on output to a standard output path the Draw module will store zeroes in these three bytes.

The element type is a number from 0 to 8 that is followed by the parameters for the element, each a word long. The path elements are as follows:

Element Type	Parameters	Description
0	n	End of path. n is ignored when reading the path, but is used to check space when reading and writing a path.
1	ptr	Pointer to continuation of path. ptr is the address of the first path element of the continuation.
2	x y	Move to (x, y) starting new subpath. The new subpath **does** affect winding numbers and so is filled normally. This is the normal way to start a new subpath.
3	x y	Move to (x, y) starting new subpath. The new subpath **does not** affect winding numbers when filling. This is mainly for internal use and rarely used by applications.
4		Close current subpath with a gap.
5		Close current subpath with a line. It is better to use one of these two to close a subpath than 2 or 3, because this guarantees a closed subpath.
6	x1 y1 x2 y2 x3 y3	Bezier curve to (x3, y3) with control points at (x1, y1) and (x2, y2).
7	x y	Gap to (x, y). Do not start a new subpath. Mainly for internal use in dot-dash sequences.
8	x y	Line to (x, y).

You will notice that there are some order constraints on these element types:

- path elements 2 and 3 start new subpaths
- path elements 6, 7 and 8 may only appear while there is a current subpath
- path elements 4 and 5 may only appear while there is a current subpath, and end it, leaving no current subpath
- path elements 2 and 3 can also be used to close the current subpath (which is a part of starting a new subpath).

Open and closed subpaths

When you are stroking (using Draw_Stroke), if a subpath ends with a 4 or 5 then it is closed, and the ends are joined – whereas a 2 or 3 leaves a subpath open, and the loose ends are capped. These four path elements explicitly leave a stroked subpath either open or closed.

Some other operations implicitly close open subpaths, and this will be stated in their descriptions.

Just because the ends of a subpath have the same coordinates, that doesn't mean the subpath is closed. There is no reason why the loose ends of an open subpath cannot be coincident.

Output path

After a SWI has written to an output path, it is identical to an input path. When it is first passed to the SWI as a parameter, the start of the block pointed to should contain an element type zero (end of path) followed by the number of available bytes. This is so that the Draw module will not accidentally overrun the buffer.

Fill style

The fill style is a word that is passed in a call to Draw_Fill, Draw_Stroke, Draw_StrokePath or Draw_ProcessPath. It is a bitfield, and all of the calls use at least the following common states. See the description of each call for differences from this:

Bit(s)	Value	Meaning
0, 1	0	non-zero winding number rule.
	1	negative winding number rule.
	2	even-odd winding number rule.

	3	positive winding number rule.
2	0	don't plot non-boundary exterior pixels.
	1	plot non-boundary exterior pixels.
3	0	don't plot boundary exterior pixels.
	1	plot boundary exterior pixels.
4	0	don't plot boundary interior pixels.
	1	plot boundary interior pixels.
5	0	don't plot non-boundary interior pixels.
	1	plot non-boundary interior pixels.
6 - 31		reserved – must be written as zero

Matrix

The matrix is passed as pointer to a six word block, in the order a, b, c, d, e, and f as described earlier. That is:

Offset	Value	Common use(s)
0	a	x scale factor, or cos(θ) to rotate
4	b	sin(θ) to rotate
8	c	–sin(θ) to rotate
12	d	y scale factor, or cos(θ) to rotate
16	e	x translation
20	f	y translation

If the pointer is zero, then the identity matrix is assumed – no transformation takes place.

Remember that a - d are in Transform units, while e and f are in internal Draw units.

Flatness

Flatness is the maximum distance that a line is allowed to be from a Bezier curve when flattening it. It is expressed in user units. So a smaller flatness will result in a more accurate rendering of the curve, but take more time and space. For very small values of flatness, it is possible to cause the 'No room in RMA' error.

A recommended range for flatness is between half and one pixel. Any less than this and you're wasting time; any more than this and the curve becomes noticeably jagged. A good starting point is:

flatness = number of user units in x axis / number of pixels in x axis

A value of zero will use the default flatness. This is set to a useful value that balances speed and accuracy when stroking to the VDU using the default scaling.

Note that if you are going to send a path to a high resolution printer, then you may have to set a smaller flatness to avoid jagged curves.

Line thickness

The line thickness is in user coordinates.

- If the thickness is zero then the line is drawn with the minimum width that can be used, given the limitations of the pixel size (so lines are a single pixel wide).
- If the thickness is 2, then the line will be drawn with a thickness of 1 user coordinate translated to pixels on either side of the theoretical line position.
- If the line thickness is non-zero, then the cap and join parameter must also be passed.

Cap and join

The cap and join styles are passed as pointer to a four word block. A pointer of zero can be passed if cap and join are ignored (as they are for zero thickness lines). The block is structured as follows:

Word	Byte	Description
0	0	join style 0 = mitred joins 1 = round joins 2 = bevelled joins
	1	leading cap style 0 = butt caps 1 = round caps 2 = projecting square caps 3 = triangular caps
	2	trailing cap style (as leading cap style)
	3	reserved – must be written as zero.
4		This value must be set if using mitred joins.
	0,1	fractional part of mitre limit for mitre joins
	2,3	integer part of mitre limit for mitre joins
8	0,1	setting for leading triangular cap width on each side (in 256ths of line widths, so &0100 is 1 linewidth)
	2,3	setting for leading triangular cap length away from the line, in the same measurements as above
12	all	This sets the trailing triangular cap size, using the same structure as the previous word.

The mitre limit is a little more complex than the others, so it is explained here rather than above. At any given corner, the mitre length is the distance from the point at which the inner edges of the stroke meet, to the point where the outer edges of the stroke meet. This distance increases as the angle between the lines decreases. If the ratio of the mitre length to the line width exceeds the mitre limit, stroke treats the corner with a bevel join instead of a mitre join. Also see the notes on scaling, later in this section.

Under RISC OS 2, the mitre limit is treated as unsigned. It is now treated as signed, but must be positive (ie ≤ &7FFFFFFF).

Note that words at offsets 4, 8, and 12 are only used if the appropriate style is selected by the earlier parts. The structure can therefore be made shorter if triangular caps and mitres are not used.

Dash pattern

The dash pattern is passed as a pointer to a block, the size of which is defined at the start, as follows:

Word	Description
0	distance into dash pattern to start in user coordinates
4	number of elements (*n*) in the dash pattern
8 - 4*n*+4	elements in the dash pattern, each of which is a distance in user coordinates.

Again the pointer can be zero, which implies that continuous lines are drawn.

Each element specifies a distance to draw in the present state. The pattern starts with the draw on, and alternates off and on for each successive element. If it reaches the end of the pattern while drawing the line, then it will restart at the beginning.

If *n* is odd, then the elements will alternate on or off with each pass through the pattern: so the first element will be on the first pass, off the second pass, on the third pass, and so on.

Scaling

The Draw module uses fixed point arithmetic for speed. The number representations used are chosen to keep rounding errors small enough not to be noticeable.

However, if you use the transformation matrix to scale a path up a great deal, you will also scale up the rounding errors and make them visible.

To avoid such problems, we recommend that you don't use scale factors of more than 8 when converting from User units to internal Draw units. (This maximum recommended scale factor of 8 is &80000 in the Transform units used in the transformation matrix.)

Draw SWIs

Though there are a number of SWIs, they all call Draw_ProcessPath. Because this takes so many parameters, the other SWIs are provided as an easy way of using its functionality.

There are two that output to the VDU. Draw_Stroke emulates the PostScript stroke function and will draw a path onto the VDU. Draw_Fill acts like the fill function and fills the inside of a path. It is likely that most applications will only use these two SWIs.

The others are shortcuts for processing a path in one way or other. Draw_StrokePath acts exactly like Draw_Stroke, except it puts its output into a path rather than onto the VDU. Filling its output path produces the same results as stroking its input path. Draw_FlattenPath will handle only the flattening of a path, writing its output to a path. Likewise, Draw_TransformPath will only use the matrix on a path. All these processing SWIs are useful when a path will be sent to the VDU many times. If the path is flattened or transformed before the stroking, then it will be done faster.

Printer drivers

If you are using a printer driver, you should note that it cannot deal with all calls to the Draw module. For full details of this, see the chapter entitled *Printer Drivers* on page 5-141. As a general rule, you should avoid the following features:

- AND, OR, etc operations on colours when writing to the screen.
- Choice of fill style: eg fill excluding/including boundary, fill exterior, etc.
- Positive and negative winding number rules.
- Line cap enhancements, particularly differing leading and trailing caps and triangular caps.

The printer driver will also intercept DrawV and modify how parts of the Draw module work. Here is a list of the effects that are common to all the SWIs that output to the VDU normally:

- cannot deal with positive or negative winding numbers
- cannot fill:
  1. non-boundary exterior pixels
  2. exterior boundary pixels only
  3. interior boundary pixels only
  4. exterior boundary and interior non-boundary pixels

- an application should not rely on any difference between the following fill states:
  1 interior non-boundary pixels only
  2 all interior pixels
  3 all interior pixels and exterior boundary pixels

SWI Calls

Draw_ProcessPath (SWI &40700)

Main Draw SWI

On entry

R0 = pointer to input path buffer (see below)
R1 = fill style
R2 = pointer to transformation matrix, or 0 for identity matrix
R3 = flatness, or 0 for default
R4 = line thickness, or 0 for default
R5 = pointer to line cap and join specification
R6 = pointer to dash pattern, or 0 for no dashes
R7 = pointer to output path buffer, or value (see below)

On exit

R0 depends on entry value of R7

if R7 = 0, 1 or 2	R0 is corrupted
if R7 = 3	R0 = size of output buffer
if R7 is a pointer	R0 = pointer to new end of path indicator

R1 - R7 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

All the other SWIs in the Draw module are translated into calls to this SWI. They are provided to ensure that suitable names exist for common operations and to reduce the number of registers to set up when calling.

The input path, matrix, flatness, line thickness, cap and join, and dash pattern are as specified in the section entitled *Data structures* on page 5-116.

The fill style is as on page 5-117, with the following additions:

Bit(s)	Meaning
6 - 26	reserved – must be written as zero
27	set if open subpaths are to be closed
28	set if the path is to be flattened
29	set if the path is to be thickened
30	set if the path is to be re-flattened after thickening
31	set for floating point output (not implemented)

Normally, the output path will act as described on page 5-117, but with the following changes if the following values are passed in R7:

Value	Meaning
0	Output to the input path buffer. Only valid if the input path's length does not change during the call.
1	Fill the path normally.
2	Fill the path, subpath by subpath. (Draw_Stroke will often use this to economise on RMA usage).
3	Count how large an output buffer is required for the given path and actions.
&80000000+pointer	Output the path's bounding box, in transformed coordinates. The buffer will contain the four words: low x, low y, high x, high y.
pointer	Output to a specified output buffer.

The length of the buffer must be indicated by putting a suitable path element 0 at the start of the buffer, and a pointer to the new path element 0 is returned in R0 to allow you to append to the output path.

You may do the following things with this call, in this order:

1 Open subpaths may be closed (if selected by bit 27 of R1).

2 The path may be flattened (if selected by bit 28 of R1). This uses R3.

3 The path may be dashed (if R6 ≠ 0).

4 The path may be thickened (if selected by bit 29 of R1). This uses R4 and R5.

5 The path may be re-flattened (if selected by bit 30 of R1). This uses R3.

6 The path may be transformed (if R2 ≠ 0).

7 Finally, the path is output in one of a number of ways, depending on R7.

Note that R3, R4 and R5 may be left unspecified if the options that use them are not specified.

If you try to dashing, thickening or filling on an unflattened Bezier curve, it will produce an error, as this is not allowed.

If you are using the printer driver, then it will intercept this SWI and affect its operation. In addition to the general comments in the section entitled *Printer drivers* on page 5-121, it is unable to handle R7 = 1 or 2.

Related SWIs

None

Related vectors

DrawV

Draw_Fill (SWI &40702)

Process a path and send to VDU, filling the interior portion

On entry

R0 = pointer to input path
R1 = fill style, or 0 for default
R2 = pointer to transformation matrix, or 0 for identity matrix
R3 = flatness, or 0 for default

On exit

R0 corrupted
R1 - 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 command emulates the PostScript 'fill' operator. It performs the following actions:

- closes open subpaths
- flattens the path
- transforms it to standard coordinates
- fills the resulting path and draws to the VDU.

The input path, matrix, and flatness are as specified in the section entitled *Data structures* on page 5-116.

The fill style is as specified on page 5-117 with the following addition. A fill style of zero is a special case. It specifies a useful default fill style, namely &30. This means fill to halfway through boundary, non-zero rule.

If you are using the printer driver, then it will intercept this SWI and affect its operation. See the general comments in the section entitled *Printer drivers* on page 5-121.

Related SWIs

None

Related vectors

DrawV

Draw_Stroke (SWI &40704)

Process a path and send to VDU

On entry

R0 = pointer to input path
R1 = fill style, or 0 for default
R2 = pointer to transformation matrix, or 0 for identity matrix
R3 = flatness, or 0 for default
R4 = line thickness, or 0 for default
R5 = pointer to line cap and join specification
R6 = pointer to dash pattern, or 0 for no dashes

On exit

R0 corrupted
R1 - R6 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 command emulates the PostScript 'stroke' operator. It performs the following actions:

- flattens the path
- applies a dash pattern to the path, if R6 ≠ 0
- thickens the path, using the specified joins and caps
- re-flattens the path, to flatten round caps and joins, so that they can be filled.
- transforms the path to standard coordinates
- fills the resulting path and draws to the VDU

The input path, matrix, flatness, cap and join, and dash pattern are as specified in the section entitled *Data structures* on page 5-116.

The fill style is as specified on page 5-117 with the following additions. A fill style of zero is a special case. If the line thickness in R4 is non-zero, then it means &30, as in Draw_Fill. If R4 is zero, then &18 is the default, as the flattened and thickened path will have no interior in this case.

If the top bit of the fill style is set, this makes the Draw module plot the stroke all at once rather than one subpath at a time. This means the code will never double plot a pixel, but uses up much more temporary work-space.

The line thickness is as on page 5-119, with the following added restrictions. If the specified thickness is zero, Draw cannot deal with filling non-boundary exterior pixels and not filling boundary exterior pixels at the same time, ie fill bits 3 - 2 being 01. If the specified thickness is non-zero, Draw cannot deal with filling just the boundary pixels, ie fill bits 5 - 2 being 0110.

If you are using the printer driver, then it will intercept this SWI and affect its operation. In addition to the general comments in the section entitled *Printer drivers* on page 5-121, you should also be aware that most printer drivers will not pay any attention to bit 31 of the fill style – ie plot subpath by subpath or all at once. Use Draw_ProcessPath to get around this problem by processing it before stroking.

Related SWIs

Draw_StrokePath (SWI &40706)

Related vectors

DrawV

Draw_StrokePath (SWI &40706)

Like Draw_Stroke, except writes its output to a path

On entry

R0 = pointer to input path
R1 = pointer to output path, or 0 to calculate output buffer size
R2 = pointer to transformation matrix, or 0 for identity matrix
R3 = flatness, or 0 for default
R4 = line thickness, or 0 for default
R5 = pointer to line cap and join specification
R6 = pointer to dash pattern, or 0 for no dashes

On exit

R0 depends on entry value of R1

if R1 = 0	R0 = calculated output buffer size
if R1 = pointer	R0 = pointer to end of path marker in output path

R1 - R6 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The input and output paths, matrix, flatness, line thickness, cap and join, and dash pattern are as specified in the section entitled *Data structures* on page 5-116.

This call acts exactly like a call to Draw_Stroke, except that it doesn't write its output to the VDU, but to an output path.

Related SWIs

Draw_Stroke (SWI &40704)

Related vectors

DrawV

Draw_FlattenPath (SWI &40708)

Converts an input path into a flattened output path

On entry

R0 = pointer to input path
R1 = pointer to output path, or 0 to calculate output buffer size
R2 = flatness, or 0 for default

On exit

R0 depends on entry value of R1
if R1 = 0 R0 = calculated output buffer size
if R1 = pointer R0 = pointer to end of path marker in output path
R1, 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

The input and output paths, and flatness are as specified in the section entitled *Data structures* on page 5-116.

This call acts like a subset of Draw_StrokePath. It will only flatten a path. This would be useful if you wanted to stroke a path multiple times and didn't want the speed penalty of flattening the path every time.

Related SWIs

Draw_StrokePath (SWI &40706)

Related vectors

DrawV

Draw_TransformPath (SWI &4070A)

Converts an input path into a transformed output path

On entry

R0 = pointer to input path
R1 = pointer to output path, or 0 to overwrite the input path
R2 = pointer to transformation matrix, or 0 for identity matrix
R3 = 0

On exit

R0 depends on entry value of R1

if R1 = 0	R0 is corrupted
if R1 = pointer	R0 = pointer to end of path marker in output path

R1 - 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

The input and output paths, and matrix are as specified in the section entitled *Data structures* on page 5-116.

This call acts like a subset of Draw_StrokePath. It will only transform a path. This would be useful if you wanted to stroke a path multiple times and didn't want the speed penalty of transforming the path every time. It is also useful if you want to transform a path before dashing, thickening and so on, to avoid having the rounding errors from the latter operations magnified by the transformation.

Related SWIs

Draw_StrokePath (SWI &40706)

Related vectors

DrawV

Application Notes

Example of simple drawing

The test program that is shown here was devised to represent millimetres internally and scale them to be the correct size when drawn on a particular monitor. Because monitors are different sizes, and even the same model can be adjusted differently in terms of vertical and horizontal picture size, this example would have to be adjusted to suit your particular setup.

This example also has a restriction on screen modes. It will only work on one where the screen is 1280 OS units by 1024 OS units – which most of the current modes are (but not, for example, 132 column modes). This corresponds to 327680 internal Draw units by 262144 internal Draw units.

The first thing to do is to fill the screen with a colour and measure the horizontal and vertical size in millimetres. For this test, the display area measured 210mm across by 160mm down.

Because of scaling limitations, we will work with a user scale of thousandths of millimetres. Thus, there are 210000 user units across and 160000 user units down.

The BASIC program described here is presented in a jumbled order so that the features are described and written one at a time. Once it is all typed in, then it will seem a lot more obvious.

Transformation matrix

The next step is to work out the scaling factors for the transformation matrix. Taking the horizontal size first, we start with 327680 internal Draw units = 210000 user units, giving 1.5604 internal Draw units per user unit. Vertically, 262144 internal Draw units = 160000 user units, giving 1.6384 internal Draw units per user unit.

These figures must now be converted to the Transform units used for scaling in the transformation matrix. The 32 bit Transform number is 2^{16} times the actual value, since its fractional part is 16 bits long. So horizontally we want $2^{16} \times 1.5604$, which is 102261 (&18F75), and vertically we want $2^{16} \times 1.6384$, which is 107374 (&1A36E).

The transformation matrix is initialised as follows:

$$\begin{bmatrix} \&00018F75 & 0 & 0 \\ 0 & \&0001A36E & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This could be calculated automatically, using the following BASIC code, which, whilst not the most efficient, is hopefully the clearest way of representing it:

```
30 xsize = 210000 : ysize = 160000
40 xscale% = (1280 * 256 / xsize) * &010000
50 yscale% = (1024 * 256 / ysize) * &010000
```

After this, `xscale%` would be &00018F75 and `yscale%` would be &0001A36E, the values to place in the matrix. The matrix would be programmed as follows:

```
20 DIM transform% 23
60 transform%!0 = xscale%                    :REM element a in the matrix
70 transform%!4 = 0                          :REM element b
80 transform%!8 = 0                          :REM element c
90 transform%!12 = yscale%                   :REM element d
100 transform%!16 = 0                        :REM element e
110 transform%!20 = 0                        :REM element f
```

Important

It is important to remember that, whilst this example is using thousandths of millimetres as its internal coordinate system, they could be anything within the valid limits. Draw is not affected by what they are. Using the technique described above, **any** valid units can be used. We used 210000 by 160000 user units for our scale; it could be 500000 by 350000 or 654363 by 314159 or whatever. This program will work with all valid scales, simply by changing the definitions of `xsize` and `ysize`.

Creating the path

In order to create the path, this simple program uses a procedure to put a single word into the path and advance the pointer. In a large application, it would be a good idea to write individual routines to generate each element type, because this technique would become tedious in a large program.

This preamble defines what needs to be at the start of the program. Notice that line 20 overwrites the earlier definition.

```
10 pathlength% = 256
20 DIM path% pathlength% - 1, transform% 23
160 pathptr% = 0                             :REM Initialise the pointer
```

Later on in the program would be the procedure to add a word to the path

```
320 END
330 DEF PROCadd(value%)
340 IF pathptr%+4 > pathlength% THEN ERROR 0,"Insufficient path buffer"
350 path%!pathptr% = value%
360 pathptr% += 4
370 ENDPROC
```

The simple path shown here generates a rectangle with no bottom line. It is 90mm by 40mm and offset by 80mm in the x and y axes from the origin.

```
170 PROCadd(2) : PROCadd(80000) : PROCadd(80000):REM Move to start
180 PROCadd(8) : PROCadd(80000) : PROCadd(120000):REM Draw
190 PROCadd(8) : PROCadd(170000) : PROCadd(120000)
200 PROCadd(8) : PROCadd(170000) : PROCadd(80000)
250 PROCadd(4) : REM Close the subpath. PROCadd(5) would close the rectangle
260 PROCadd(0) : PROCadd(pathlength%-pathptr%-4):REM End path
```

Simple stroke

Once the path and the transformation matrix have been completed, all that remains is to set the graphics origin and stroke the path onto the screen.

```
270 VDU 29,0;0;
280 SYS "Draw_Stroke",path%,0,transform%,0,0,0,0
```

Translation

Another matrix operation that can be performed is translation, or moving. Remember that the parameters in the matrix are in internal Draw coordinates, not the millimetres used in this example as user coordinates. If you want to translate in OS coordinates, then the translation must be multiplied by 256.

In this example, we are going to re-stroke the path, translated 60 OS units in x and –100 OS units in y.

```
290 transform%!16 = 60<<8
300 transform%!20 = -100<<8
310 SYS "Draw_Stroke",path%,0,transform%,0,0,0,0
```

You will now see two versions of the path, the new one 100 OS units lower and 60 OS units shifted to the right.

Similarly, the matrix may be modified to rotate the path. If you aren't sure how to do this, then see any mathematical text on matrix arithmetic.

Curves

In order to add a curve to the path, we will add a new subpath to the section that creates the path. This curve draws an alpha shape. Note that element type 2 implicitly closes the initial subpath:

```
210 PROCadd(2) : PROCadd(50000) : PROCadd(50000)   :REM x1, y1
220 PROCadd(6) : PROCadd(60000) : PROCadd(80000)   :REM x2, y2
230 PROCadd(85000) : PROCadd(30000)                :REM x3, y3
240 PROCadd(50000) : PROCadd(60000)                :REM x4, y4
```


Whilst the flatness can be left at its default value, this shows how the stroke commands can be changed to set the flatness to a sensible value. 640 is used because this program was run in a 640 pixel mode.

```
280 SYS "Draw_Stroke",path%,0,transform%,xsize/640,0,0,0
310 SYS "Draw_Stroke",path%,0,transform%,xsize/640,0,0,0
```

Line thickness

To make the lines shown thicker than the default, it is necessary to specify a thickness and also the joins and caps block. Notice that line 20 has been changed to allocate space for the joins and caps block. We will use round caps and bevelled joints.

```
20 DIM path% pathlength%-1, transform% 23, joinsandcaps% 15
120 joinsandcaps%!0 = &010102
130 joinsandcaps%!4 = 0
140 joinsandcaps%!8 = 0
150 joinsandcaps%!12 = 0
```

Now all that remains is to change the stroke commands to specify a thickness and point to the block just specified. For this example we will make the first stroke 5000 units (5mm) thick and the second one half that:

```
280 SYS "Draw_Stroke",path%,0,transform%,xsize/640, 5000,joinsandcaps%,0
310 SYS "Draw_Stroke",path%,0,transform%,xsize/640, 2500,joinsandcaps%,0
```

Plainly, there are many more features that could be added to this program. But you should have the idea now of how it fits together and be able to experiment for yourself.

57 Printer Drivers

Introduction

One of the major headaches on some operating systems is that all applications must write drivers for all the required types of printers. This duplicates a lot of work and makes each application correspondingly larger and more complex.

The solution to this problem that RISC OS has adopted is to supply a virtual printer interface, so that all printer devices can be used in the same way. Thus, your application can write to the printer, without being aware of the differences between, for example, a dot matrix or PostScript printer or an XY plotter.

To simplify printer driving further, the printer can be driven with a subset of the same calls that normally write to the screen. Calls to the VDU drivers and to the SpriteExtend, Draw, ColourTrans and Font modules are trapped by the printer driver. It interprets all these calls in the most appropriate way for the selected printer. Where possible, the greater resolution of most printers is used to its fullest advantage.

Of course, not all calls have meaning to the printer driver – flashing colours for example. These generate an error or are ignored as appropriate.

Printer drivers are written to support a general class of printers, such as PostScript printers. They each have a matching desktop application that allows users to control their unique attributes. Thus, applications need not know about printer specific operation, but this does not result in lack of fine control of the printer.

RISC OS 3 completely revamped the way in which printer drivers operate; in particular you can now have more than one printer driver installed at the same time and it is very easy to switch between them. Support for this is supplied by a sharer module and also by the new improved Printer manager application.

Overview

A printer driver is implemented in RISC OS as a relocatable module. It supplies SWIs concerned with starting, stopping and controlling a print job.

Rectangles

A key feature of all printer drivers is the rectangle. In normal use, it is a page. It is however possible to have many rectangles appear on the same physical sheet of paper. For example, an A3 sized plotter may be used to draw two A4 rectangles on it side by side; or it could be used to generate a pagination sheet for a DTP package, showing many rectangles on a sheet.

When reading this chapter, in most cases you can consider a rectangle and a page to be effectively equivalent, but bear in mind the above use of rectangles.

Measurement systems

Many of the printer driver SWIs deal with an internal measurement system, using millipoints. This is 1/1000th of a point, or 1/72000th of an inch. This system is an abstraction from the physical characteristics of the printer. Printed text and graphics can be manipulated by its size, rather than in terms of numbers of print pixels, which will vary from printer to printer.

OS units

OS units are the coordinate system normally used by the VDU drivers. In this context, an OS unit is defined as 1/180th of an inch, so each OS unit is 2/5ths of a point, or 400 millipoints.

It is in this coordinate system that all plotting commands are interpreted. When a rectangle is declared, it is given a size in OS units. This is treated like a graphics window, with output outside it being clipped, and so on.

Transform matrix

Like the Draw module, the printer driver uses a transform matrix to convert OS units to the scale, rotation and translation required on paper. With a matrix with no scaling transformation, a line of 180 OS units, or one inch, will appear as an approximation of an inch long line on all printers. Naturally, it depends on the resolution of the printer as to how close to this it gets. If the matrix scaled x and y up by two, then the line would be two inches long.

Using the printer driver

To send output to the printer, an application must engage in a dialogue with the printer driver. This is similar in part to the dialogue used with the Wimp when a window needs redrawing.

The application starts by opening a file to receive the printer driver's output. The file can be the printer, or a file on any filing system. It passes this to the printer driver to start a print job.

For each page, the application goes through the following steps:

1 Pass the printer driver a description of each rectangle to use for the page.
2 Tell the printer driver to start drawing the page. It will return with an ID for the first rectangle it needs.
3 Go through the printer output using calls to the VDU, Draw, Font, etc.
4 Ask the printer for the next rectangle and repeat stage 3
5 Repeat stages 3 and 4 as often as required. The printer driver will tell you when it no longer requires any output.

The printer driver will ask either for all of, or for a section of a rectangle you specified. It may ask for a given rectangle once, or many times. A dot matrix driver, for instance, may get the output a strip at a time to conserve workspace, whereas a PostScript driver can send the lot out to the printer in one go.

The point is that you should have no preconceptions about how many times the printer driver will ask for a rectangle, or the order in which it requests rectangles.

When all the required pages have been printed, you issue a SWI to finish the print job and then close the file.

See the example at the end of this chapter for a practical guide to this process.

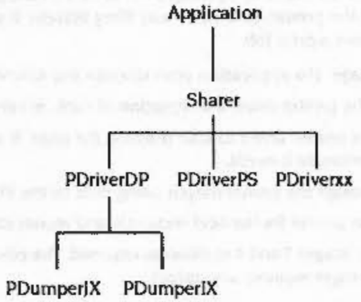
SWI interception

The printer driver works by trapping all calls to the VDU drivers and to the SpriteExtend, Draw, ColourTrans and Font modules. It will pass some on to the destination module unchanged. Some will generate an error because they cannot be interpreted by the printer driver. Some will be discarded. The ones that are of most interest are taken by the printer driver and interpreted in the most appropriate way for the printer. The section entitled *Technical Details* in this chapter describes how each module's calls are interpreted by the printer driver.

How the printer sharer works

Any SWI's directed directly at a specific driver will be decoded down to a normal call and then passed down to the decoder.

The system works as follows:



The printer sharer module allows many printer driver modules to be resident at once in the machine. The module is required so that devices can have their drivers present at the same time. For example, a dot-matrix printer, a PostScript printer and a Fax card.

Support has been added for a new type of device called 'bit map' id 7, this is designed to be a universal type of device to which PDumper modules are attached that provide the actual device driving and the rendering is handled at a higher level.

New features of the graphics system and font manager have been echoed onto the printer drivers as relevant, these include:

- Transformed sprites
- Transformed fonts, kerning
- Font downloading
- New ColourTrans SWIs
- Handling of true-256 colour sprites.

In all aspects we have retained backwards compatibility and provided a meaningful way for the applications author to decide which features are supported.

The PDriver module

All the standard printer driver SWI's pass through the printer driver sharer module. There are two SWI's PDriver_DeclareDriver and PDriver_SelectDriver. The first of these allows the new printer driver modules to declare themselves to the printer sharer, which they must do before they can be used. The second SWI allows the user to choose which printer driver is to be used for subsequent PDriver SWI's.

One new service call is introduced – Service_PDriverStarting. This is issued when the printer sharer module starts up, and it lets any printer drivers resident at that time declare themselves to the printer sharer module.

To provide for these new features the old printer drivers have to be recompiled with a new printer-independent source code, and with a single slight adjustment to one of the printer-dependent source code files (to provide a null service routine). ISVs who have written their own drivers using our printer-independent code will be able to recompile without difficulty.

The SWI handling

The new printer sharer module has completely taken over the printer driver SWI chunk. When a SWI such as PDriver_PageSize is issued the printer sharer module must pass it on to a particular declared printer driver module. To do this it has a 'current printer driver' concept.

To set the current printer driver use:

SWI PDriver_SelectDriver (&80156)

On entry:	R0 = printer number to select
On exit:	R0 = printer number previously selected, or error pointer

The printer number is as specified:

0 PDriverType_PS	Acorn PostScript
1 PDriverType_DM	Acorn Dot Matrix
2 PDriverType_LJ	Acorn LaserJet
3 PDriverType_IX	Acorn Integrex
4 PDriverType_FX	Computer Concepts Fax
5 PDriverType_LZ	Computer Concepts Laser board
6 PDriverType_LB	Computer Concepts Laser board
7 PDriverType_UF	Acorn Printer Dumper driver
99 PDriverType_JX	Ace Computing Epson JX/Star LC10 driver
99 PDriverType_PJ	Ace Computing PaintJet

Any calls to these SWIs will be passed to the current printer driver:

PDriver_Info
PDriver_SetInfo
PDriver_PageSize
PDriver_SetPageSize
PDriver_CheckFeatures
PDriver_SelectJob
PDriver_SelectIllustration
PDriver_ScreenDump
PDriver_SetPrinter

Any of the SWIs above may be postfixed by 'ForDriver', giving PDriver_InfoForDriver, PDriver_SelectJobForDriver and so on. These SWIs use a printer number in R7 instead of the 'current printer driver' number, thus allowing temporary selection of a printer driver just for a single SWI.

Job handling SWIs are treated differently. The printer sharer module keeps track of which printer driver owns which jobs, so that calls to PDriver_AbortJob can be passed on to the correct driver. There are three sorts of SWI which affect jobs – those which are handled completely by the sharer, those which are handled completely by the printer driver itself, and those which require both sides.

Those which are handled internally are PDriver_CurrentJob and PDriver_EnumerateJobs. These just require some inspection of the printer sharer's internal job management structures, and no interaction with the real drivers.

Those which are handled completely by the printer drivers are PDriver_CancelJob and PDriver_CancelJobWithError. These simply set flags inside the real driver to stop future printer actions on the specified job from working – they do not affect the job management in the printer sharer module itself.

The last set of job handling SWIs is PDriver_SelectJob, PDriver_SelectIllustration, PDriver_AbortJob, PDriver_EndJob, PDriver_Reset. The code for the select SWIs is quite complex, as it has to deselect the current job on one driver and then select the new job on a new driver. Any errors occurring in the selection process will lead to NO job being selected on exit – this is an incompatible change, but should really affect nobody. Ending and Aborting are easily handled, they clear the internal data for the specified job and then pass through to the real driver. PDriver_Reset has to reset ALL the printer drivers, which is again easily performed.

Two classes of SWI remain. Those like PDriver_DrawPage and PDriver_GetRectangle, which must be passed on to the driver which owns the currently selected job, and PDriver_DeclareDriver. The first class is easily handled.

The SWI PDriver_DeclareDriver declares a printer driver module to the printer sharer. R2 is the printer number for the printer driver module. R0 points to the SWI handler code in the printer driver. This is called with R11 being the SWI number (a kind of reason code) as specified by the operating system to a standard SWI handler. R1 contains the workspace pointer (R12) for calls to the printer driver SWI handler. This mechanism means that there is little real change required in the real printer driver modules.

PDumper modules

RISC OS 2.00 used four printer driver modules – one each for: Epson printers; Integrex printers; HP LaserJet printers; PostScript printers.

The upkeep on such a large amount of source code was tremendous, and a lot of the code was common between the printers. RISC OS 3 has introduced a "common" chunk of source code (printer independent), and printer dependent portions to handle those parts which operate differently on different printers. Hence the split between the large, well documented PDriver source file and the other code in the printer drivers individual directories. This split in source code is handled by having the large independent source file use the appropriate printer dependent source files at appropriate points in its body code. The files which are included contain code for: colour setting, job management, VDU 5 output, sprite output, font output, draw module output, and so on.

Most of the printer dependent parts of the Epson, Integrex and HP LaserJet drivers are similar or identical. So the PDumper module concept was invented. The idea of the PDumper was to provide a low volume of source code per new printer supported. The code would contain just those items of interest to the printer – colour selection, page handling, and bitmap outputting. The main printer driver module (PDriverDP) would just link into the appropriate PDumper module when it needed to select a colour or whatever. In this way the printer independent code is kept in a different module to the printer dependent code, as opposed to joining the two pieces of code at assembly time. The volume of code required to be understood by someone writing a PDumper modules is just that which makes up the PDumper module.

The PDumper modules were designed by examining the three bitmap printer drivers that Acorn had at the time, and selecting the similar parts of the source code together to form the PDriverDP module. In fact this turned out to be putting hooks into colour setting, bitmap line outputting, page handling, job handling and the service routine. Work was required on the service routine to allow PDumper modules to be resident all the time and declare themselves to PDriverDP when it started up – hence a new PDriverStarting service call was added, and the PDumper modules link in at that time. Also, when a PDumper module starts up it calls a SWI to declare itself to the PDumper module, if there is one present, thus ensuring the PDriverDP module always knows precisely which PDumper modules are present at any time.

Using fonts in the new printer system

The new PostScript printer drivers have enhanced support for utilising PostScript fonts resident in the printer, as well as the ability to download PostScript equivalents of RISC OS fonts.

As far as the application writer is concerned, the details of the process are transparent, but a brief summary is presented below.

New-style applications

When an application attempts to print a document containing fonts, it should call PDriver_DeclareFont once for each font to be printed. The font name passed to this call should be exactly the same as the one passed to Font_FindFont, including any encoding and matrix information. If the document does not use fonts, then it should call the SWI just once, with the end-of-list value of 0, to indicate this fact. (Otherwise the printer driver takes special action on the assumption that the application is unaware of the PDriver_DeclareFont call; see below).

When the printer driver is ready to output the PostScript prologue, it scans this list of fonts. Each name is passed to the MakePSFont module, which attempts to ensure that the font is available in the printer by one of the following methods:

- **Using an existing PostScript font directly**, (augmenting an existing PostScript font by applying a different encoding and/or transformation matrix).
- **Downloading a PostScript version of the font on the fly**.

The most efficient method possible is chosen – downloading is only done as a last resort, because the resulting fonts are very large.

To make this choice, the printer driver has to know which fonts are already available in the printer. This information is maintained by the printer driver system, and controlled by use of the !FontPrint application. FontPrint lets the user specify the mapping between RISC OS font names and PostScript font names, such as Trinity.Medium maps to Times-Roman.

Old-style applications

An old-style application does not make any calls to PDriver_DeclareFont, and hence the printer system cannot be certain about which fonts to provide. (The rules of PostScript prologue generation prevent us from simply sending the font the first time it is used in the print job – they must all be known in advance).

There are two mechanisms for coping with this situation. The simplest emulates the old printer driver and sends a prologue file that blindly provides a fixed set of fonts. This satisfies most old applications because they were written with this expectation. The advanced user can edit the prologue file by hand to adjust the list of fonts provided.

The second and more sophisticated method takes the intersection of the set of fonts known to the font manager and the set of fonts known to be resident in the printer. It passes each font in the resulting set to MakePSFont. Thus all of the fonts that can be provided by simple renaming of an existing PostScript font are sent, which is fairly comprehensive but still efficient.

The user chooses between these two mechanisms by the "Verbose prologue" switch in the !Printers configuration window.

Technical Details

Printer driver SWIs

Printer information

Though an application shouldn't need to look at all its information, PDriver_Info (SWI &80140) will provide information about the nature of the printer. This includes the:

- type of printer
- x and y resolution
- colour and shading capabilities
- name of the printer (applications usually need to look at this)
- ability to handle filled shapes, thick lines, screen dumps and transformations

PDriver_CheckFeatures (SWI &80142) allows an application to check the printer features described above. This means that an application could change the way it works depending on some general features of the printer.

Much as this system tries to avoid this sort of thing, it is inevitable in some cases. For example, an application that uses lots of sprites on screen will have to go about printing in a different way on an XY plotter. Many colour limitations, however, are solved using halftoning.

PDriver_PageSize (SWI &80143) returns the size of paper and printable area on it. This is used to calculate what size of rectangle to use on it.

Starting a print job

To open a print job, you should first open 'printer:' as a file. This device independent name is used because the printer driver application has control over the OS_Byte 5 settings of printer destination (see the chapter entitled *Character output* for details of OS_Byte 5).

You may open any other valid pathname as a file to use as a printer output. The file created may subsequently be dumped to the printer. This technique could be used for background printing, for instance.

The file handle is passed to PDriver_SelectJob (SWI &80145). It suspends the current print job, if there is one, and makes the handle you passed the current one. It is the application's responsibility to do this at the right time, because it has sole control over what gets printed at any time on the machine it is running on. Needless to say, a network printer spooler can cope with print commands coming from many machines.

A simple use of the printer driver is to call PDriver_ScreenDump (SWI &8014F) which will dump the screen to the printer, if it can handle it. See also the description of screen dumps in the chapter entitled *Sprites*.

Controlling a print job

PDriver_CurrentJob (SWI &80146) will tell you the file handle for the currently active print job.

PDriver_EnumerateJob (SWI &80150) allows you to scan through all the print jobs that the printer driver currently knows about.

PDriver_EndJob (SWI &80148) will end a job and remove the file handle from the printer driver's internal lists. It will issue all the closing commands to the printer to flush any pages in progress. The file should be closed after doing this, to formally finish the print job.

PDriver_AbortJob (SWI &80149) is a more forceful termination. It should be called after any errors while printing. It guarantees that no more commands will be sent to the printer after it.

PDriver_CancelJob (SWI &8014E) will cancel a job. It is normally followed by the job being aborted. It is not intended to be used by the printing application, but by another task that allows cancellations of print jobs. It would use PDriver_EnumerateJobs to find out which jobs exist and then cancel what it wishes to. The application that owns the cancelled job would subsequently find that it had been cancelled and would then abort the job.

PDriver_Reset (SWI &8014A) will abort all print jobs known to the printer driver. Normally, you should never have to use this command. It may be useful during development of an application as an emergency recovery measure.

Printing a page

There are two phases to printing a page. First you must specify all the rectangles to use on the page with PDriver_GiveRectangle (SWI &8014B). Each rectangle has a size, transformation matrix, position on the page and rectangle ID specified by you.

Then you call PDriver_DrawPage (SWI &8014C) to start the print phase. It returns the first rectangle to output. This may be only a strip of the rectangle you specified, if the printer driver cannot do it all at once. This call is followed by repeated calls to PDriver_GetRectangle (SWI &8014D) until it returns saying that there are no more rectangles to print.

The printer driver is free to request rectangles in any order it pleases and as many times as it pleases. For each rectangle request, you must redraw that part of the rectangle.

See the example at the end of this chapter for a practical guide to the sequence to use.

Private SWIs

Some SWIs are used in the interface between the printer driver desktop application, the printer driver, and the font manager. They are briefly described in this chapter, but you must not use them. If nothing else, the interface is not guaranteed because it is a private one. These are the private SWIs:

- PDriver_SetInfo (SWI &80141)
- PDriver_SetPageSize (SWI &80144)
- PDriver_FontSWI (SWI &80146)
- PDriver_SetPrinter (SWI &80151)

Trapping of screen SWIs

When a printer driver is running, it intercepts the following vectors:

- WrchV
- SpriteV
- DrawV
- ColourV
- ByteV

Many of the calls that pass through these vectors will be passed unchanged through the printer driver. However some calls are trapped. In some cases they are changed to something appropriate, and in others generate an error because they cannot be implemented. In addition, the font manager SWIs are trapped through an internal mechanism.

Character output operations passed on to the printer drivers

OS_WriteC
OS_WriteS
OS_WriteO
OS_NewLine
OS_Byte 3 (in the standard state OS_Byte 3,0)
OS_Byte242 (use strictly as documented)
OS_Byte245 (use strictly as documented)
OS_PrettyPrint
OS_Write1-1FF

VDU driver operations passed on to the printer drivers

VDU 5 (always behaves as though set – ignored if issued)
VDU 8
VDU 9
VDU 10
VDU 11
VDU 12
VDU 13
VDU 16 (= VDU 12)
VDU 18 (best not to use – use ColourTrans instead)
VDU 21 (disables print output)
VDU 23,16 (bit 6 ignored – behaves as though set)
VDU 23,17,0 - 3 (0-1 ignored, 2-3 only affect printer tints)
VDU 23,17,7
VDU 24 (works provided clipping box lies within rectangle being printed)
VDU 26
VDU 29
VDU 30
VDU 31
OS_Byte 19
OS_Byte 20
OS_Byte 25
OS_Byte 113
OS_Byte 114
OS_Byte 117
OS_Byte 135 (don't use except for screen mode)
OS_Byte 193
OS_Byte 194
OS_Byte 195
OS_Byte 211
OS_Byte 212
OS_Byte 213
OS_Byte 214
OS_Byte 217
OS_Byte 218
OS_Byte 250
OS_Byte 251
OS_Word 10
OS_Word 11
OS_Word 13
OS_Word 21, 0 to OS_Word 21, 6
OS_Word 22
OS_Mouse

OS_RemoveCursors
OS_RestoreCursors
OS_CheckModeValid
OS_ReadSysInfo
OS_ChangedBox
*Shadow
*TV

Sprite module operations passed on to the printer drivers

OS_SpriteOp 8
OS_SpriteOp 9
OS_SpriteOp 10
OS_SpriteOp 11
OS_SpriteOp 12
OS_SpriteOp 13
OS_SpriteOp 15
OS_SpriteOp 24 UserSprite passed on, else faulted
OS_SpriteOp 25 to 27
OS_SpriteOp 29 to 33
OS_SpriteOp 35 to 47
OS_SpriteOp 54 to 58
OS_SpriteOp 62
*SCopy
*SDelete
*SFlipX
*SFlipY
*SInfo
*SList
*SLoad
*SMerge
*SNew
*SRename
*SSave

ColourTrans module operations passed on to the printer drivers

ColourTrans_SelectTable (R2 = –1. The way to set colours/sprite trans tables)
ColourTrans_SetGCOL (The way to set colours – currently ignore bit 8)
ColourTrans_ReturnColourNumber
ColourTrans_ReturnColourNumberForMode
ColourTrans_ReturnOppColourNumber
ColourTrans_ReturnOppColourNumberForMode
ColourTrans_SetFontColours

Font Manager operations passed on to the printer drivers

Font_CacheAddr
Font_FindFont
Font_LoseFont
Font_ReadDefn
Font_ReadInfo
Font_StringWidth
Font_Paint (don't use with colour changing sequences)
Font_Caret
Font_ConverttoOS
Font_Converttopoints
Font_SetFont
Font_CurrentFont
Font_FutureFont
Font_FindCaret
Font_CharBBox
Font_ReadScaleFactor
Font_SetScaleFactor
Font_ListFonts
Font_ReadThresholds
Font_SetThresholds
Font_FindCaret
Font_StringBBox
Font_ReadColourTable
Font_MakeBitmap
Font_UnCacheFile
Font_SetFontMax
Font_ReadFontMax
Font_ReadFontPrefix
*Configure FontMax
*Configure FontMax1
*Configure FontMax2
*Configure FontMax3
*Configure FontMax4
*Configure FontMax5
*Configure FontSize
*FontCat
FontList

Draw module operations passed on to the printer drivers

Draw_ProcessPath (R7 = 1 or 2 faulted due to bounding box restrictions)
Draw_Fill (Bit 0 must be clear
Draw_Stroke (Some restrictions)
Draw_StrokePath
Draw_FlattenPath
Draw_TransformPath

Below, we pass section by section through the effects of the printer driver on the calls that pass through these vectors.

WrchV

Whenever a print job is active, the printer driver will intercept all characters sent through WrchV. It will then queue them in the same way as the VDU drivers do and process complete VDU sequences as they appear. Because the printer driver will not pick up any data currently in the VDU queue, and may send sequences of its own to the VDU drivers, a print job should not be selected with an incomplete sequence in the VDU queue.

OS_Byte 3

Also, because the printer driver may send sequences of its own to the VDU drivers, the output stream specification set by OS_Byte 3 should be in its standard state – as though set by OS_Byte 3,0.

Commands passed on to the VDU

The printer driver will pass the following VDU sequences through to the normal VDU drivers, either because they control the screen hardware or because they affect global resources such as the character and ECF definitions:

VDU 7	Produce bell sound
VDU 19,l,p,r,g,b	Change hardware palette
VDU 20	Set default hardware palette
VDU 23,0,n,m|	Program pseudo-6845 registers
VDU 23,1,n|	Change cursor appearance
VDU 23,2-5,a,b,c,d,e,f,g,h	Set ECF pattern
VDU 23,9-10,n|	Set flash durations
VDU 23,11|	Set default ECF patterns
VDU 23,12-15,a,b,c,d,e,f,g,h	Simple setting of ECF pattern
VDU 23,17,4,m|	Set ECF type
VDU 23,17,6,x,y;|	Set ECF origin
VDU 23,32-255,a,b,c,d,e,f,g,h	Define character

The printer driver will interpret or fault all other VDU sequences. If the printer driver currently wants a rectangle printed, these will result in it producing appropriate output or errors – that is, if there has been a call to PDriver_DrawPage or PDriver_GetRectangle and the last such call returned R0 ≠ 0. Otherwise, the printer driver will keep track of some state information – for example, the current foreground and background colours – but will not produce any printer output.

Error commands

The printer driver will always behave as though it is in VDU 5 state. No text coordinate system is defined, and no scrolling is possible. For these reasons, the following VDU sequences are faulted:

VDU 4	exit VDU 5 state
VDU 23,7,m,d,z|	scroll display
VDU 23,8,t1,t2,x1,y1,x2,y2|	clear text block

VDU printer

It is generally meaningless to try to send or echo characters directly to the printer while printing. Furthermore, attempts to do so are likely to disrupt the operation of printer drivers. For these reasons, the following VDU sequences are faulted:

VDU 1,c	send character to printer
VDU 2	start echoing characters to printer

Screen mode

It is not possible to change the 'mode' of a printed page, so the following VDU sequence is faulted:

VDU 22,m	change display mode

Reserved calls

A printer driver cannot be written to deal with undefined or reserved calls, so the following VDU sequences are faulted:

VDU 23,18-24,...	reserved for Acorn expansion
VDU 23,28-31,...	reserved for use by applications
VDU 25,216-231,...	reserved for Acorn expansion
VDU 25,240-255,...	reserved for use by applications

Ignored

The following VDU sequences are ignored, either because they normally do nothing (at least when stuck in VDU 5 mode and not echoing characters to the printer) or because they have no sensible interpretation when output to anything other than a screen.

VDU 0	do nothing
VDU 3	stop echoing characters to printer
VDU 5	enter VDU 5 state
VDU 14	start 'paged' display
VDU 15	end 'paged' display
VDU 17,c	define text colour
VDU 23,17,5l	exchange text foreground and background
VDU 27	do nothing
VDU 28,l,b,r,t	define text window

Colours

Colours are a rather complicated matter. It is strongly recommended that applications should use ColourTrans_SetGCOL, ColourTrans_SelectTable and ColourTrans_SetFontColours to set colours, as these will allow the printer to produce as accurate an approximation as it can to the desired colour, independently of the screen palette. The GCOL sequence (VDU 18,k,c) should only be used if absolutely necessary, and you should be aware of the fact that the printer driver has a simplified interpretation of the parameters, as follows:

- The fact that the background colour is affected if c 128 and the foreground colour if c < 128 is unchanged.
- If k MOD 8 ≠ 0, subsequent plots and sprite plots will not do anything.
- If k=0, subsequent plots will cause the colour c MOD 128 (possibly modified by the current tint) to be looked up in the screen palette at the time of plotting (rather than the time the VDU 18,k,c command was issued). Plotting is done by overwriting with the closest approximation the printer can produce to the RGB combination found. Subsequent sprite plotting will be done without use of the sprite's mask.
- If k=8, subsequent plots will be treated the same as k=0 above, except that sprite plots will be done using the sprite's mask, if any.
- If k > 8, an unspecified solid colour will be used.

VDU 18

The major problems with the use of VDU 18,k,c to set colours are:

- that it makes the printer driver output dependent on the current screen mode and palette.
- that it artificially limits the printer driver to the number of colours displayed on the screen, which can be very limiting in a two colour mode.

Other GCOLs

Other techniques that depend on GCOLs have the same problems and are similarly not recommended: for example Font_SetFontColours, colour-changing sequences in strings passed to Font_Paint, plotting sprites without a translation table, and so on.

No operations other than overwriting are permitted, mainly because they cannot be implemented on many common printers – such as PostScript printers.

Foreground and background colours

Note that the printer driver maintains its own foreground and background colour information. The screen foreground and background colours are not affected by VDU 18,k,c sequences encountered while a print job is active.

VDU 23,17

Similarly, VDU 23,17,2-3,tl sequences encountered while a print job is active do not affect the screen tints, just the printer driver's own tints. VDU 23,17,0-1,tl sequences would only affect the colours of the text tints, so the printer driver ignores them.

Other graphics state operations

The VDU 6 and VDU 21 sequences have their normal effects of enabling and disabling execution, but not parsing, of subsequent VDU sequences. As usual, the printer driver keeps track of this independently of the VDU drivers.

Cursor movement

The cursor movement VDU sequences (ie. VDU 8-11, VDU 13, VDU 30 and VDU 31,x,y) all update the current graphics position without updating the previous graphics positions, precisely as they do in VDU 5 mode on the screen.

VDU 24

VDU 24,l;b;r;t; will set the printer driver's graphics clipping box. The rectangle specified should lie completely within the box that was reported on return from the last call to PDriver_DrawPage or PDriver_GetRectangle. If this is not the case, it is not defined what will happen, and different printer drivers may treat it in different ways. This is analogous to the situation with the window manager. Attempts to set a graphics clipping box outside the rectangle currently being redrawn may be ignored completely if they go outside the screen, or may get obeyed with consequences that are almost certainly wrong.

VDU 29

VDU 29,x;y; sets the printer driver's graphics origin.

VDU 26

VDU 26 will reset the printer driver's graphics clipping box to its maximum size. This is essentially the box reported on return from the last call to PDriver_DrawPage or PDriver_GetRectangle, but may be slightly different due to rounding problems when converting from a box expressed in printer pixels to one expressed in OS units. It also resets its versions of the graphics origin, the current graphics position and all the previous graphics positions to (0,0).

VDU 23,6

VDU 23,6 will fault because dot-dash lines are not implemented in current printer drivers. Use the dashed line facility of Draw_Stroke instead.

VDU 23,16

VDU 23,16,x,yl changes the printer driver's version of the cursor control flags, and thus how the cursor movement control sequences and BBC-style character plotting affect the current graphics position. As usual, this is completely independent of the corresponding flags in the VDU drivers. However, printer drivers pay no attention to the setting of bit 6, which controls whether movements beyond the edge of the graphics window cause carriage return/line feeds and other cursor movements to be generated automatically. They always behave as though it is set. Note that the Wimp normally sets this bit, and that it is not sensible to have it clear at any time during a Wimp redraw.

VDU 23,17,7

VDU 23,17,7,flags,x;y;l changes the printer driver's version of the size that BBC-style characters are to be plotted and the spacing that is required between them. Setting the VDU 4 character size cannot possibly affect the printer driver's output and so will be ignored completely. As noted below under 'Plotting operations', a pixel is regarded as the size of a screen pixel for the screen mode that was in effect when the print job was started.

Plotting operations

The printer driver regards a pixel as having size 2 OS units square (1/90 inch square). The main effect of this is that all PLOT line, PLOT point and PLOT outline calls will produce lines that are approximately 2 OS units wide.

Use Draw module calls if you wish to produce different lines.

VDU 23,17,7

However, when translating the character size and spacing information provided by VDU 23,17,7,... (see above) from pixels to OS units, the screen pixel size for the screen mode that was in effect when the print job was started is used. This is done in the expectation that the application is basing its requested sizes on that screen mode.

VDU plot operations

The following VDU sequences perform straightforward plotting operations; printer drivers will produce the corresponding printed output:

VDU 12	clear graphics window in VDU 5 state
VDU 16	clear graphics window
VDU 25,0-63,x;y;	draw line; however, the lines are always plotted solid, so only VDU 25,0-15,... and VDU 25,32-47,... will look the same as in VDU output. Use Draw_Stroke to generate dashed lines that will come out well in printed output.
VDU 25,64-71,x;y;	draw point
VDU 25,80-87,x;y;	fill triangle
VDU 25,96-103,x;y;	fill axis-aligned rectangle
VDU 25,112-119,x;y;	fill parallelogram
VDU 25,144-151,x;y;	draw circle
VDU 25,152-159,x;y;	fill circle
VDU 25,160-167,x;y;	draw circular arc
VDU 25,168-175,x;y;	fill circular segment
VDU 25,176-183,x;y;	fill circular sector
VDU 25,192-199,x;y;	draw ellipse
VDU 25,200-207,x;y;	fill ellipse
VDU 32-126	print characters in BBC-style font
VDU 127	backspace & delete
VDU 128-255	print characters in BBC-style font

Rounding

One difference to note is that most printer drivers will either not do the rounding to pixel centres normally done by the VDU drivers, or will round to different pixel centres – probably the centres of their device pixels.

Faulted

The following VDU sequences are faulted because they cannot be split up easily across rectangles, and also because they depend on the current picture contents and so cannot be implemented, for example, on PostScript printers:

VDU 25,72-79,x;y;	horizontal line fill (flood fill primitive)
VDU 25,88-95,x;y;	horizontal line fill (flood fill primitive)
VDU 25,104-111,x;y;	horizontal line fill (flood fill primitive)
VDU 25,120-127,x;y;	horizontal line fill (flood fill primitive)
VDU 25,128-143,x;y;	flood fills
VDU 25,184-191,x;y;	copy/move rectangle

VDU 25,184,x;y; and VDU 25,188,x;y; are exceptions to this; they are correctly interpreted by printer drivers as being equivalent to VDU 25,0,x;y; and VDU 25,4,x;y; respectively.

Sprite VDUs

The sprite plotting VDU sequences (VDU 23,27,m,n| and VDU 25,232-239,x;y;) and the font manager VDU sequences (VDU 23,25,a,b,c,d,e,f,g,h, VDU 23,26,a,b,c,d,e,f,g,h,text and VDU 25,208-215,x;y;text) cannot be handled by the printer drivers and generate errors. You should use OS_SpriteOp and the font manager SWIs instead.

SpriteV

Printer drivers intercept OS_SpriteOp via the SpriteV vector. Most calls are simply passed through to the operating system or the SpriteExtend module. The ones that normally plot to the screen are generally intercepted and used to generate printer output by the printer driver.

Faulted

The following reason codes normally involve reading or writing the screen contents and are not straightforward sprite plotting operations. Because some printer drivers redirect output to a sprite internally, it is unknown what the 'screen' is during these operations. They are therefore faulted.

2	screen save
3	screen load
14	get sprite from current point on screen
16	get sprite from specified point on screen

Passed on

Reason codes that are passed through to the operating system or the SpriteExtend module are:

8	read sprite area control block
9	initialise sprite area
10	load sprite file
11	merge sprite file
12	save sprite file
13	return name of numbered sprite
15	create sprite
25	delete sprite
26	rename sprite
27	copy sprite
29	create mask
30	remove mask
31	insert row
32	delete row
33	flip about X axis
35	append sprite
36	set pointer shape
40	read sprite size
41	read pixel colour
42	write pixel colour
43	read pixel mask
44	write pixel mask
45	insert column
46	delete column
47	flip about Y axis
62	read save area size

Select error

The following reason code is passed through to the operating system when it is called for a user sprite (ie with &100 or &200 added to it), as this call is simply asking the operating system for the address of the sprite concerned. If the system version is called (ie without anything added to it), it is asking for a sprite to be selected for use with the VDU sprite plotting sequences. As these sequences are not handled by the printer driver, this version of the call generates an error.

24	select sprite

Sprite plotting

The following reason codes plot a sprite or its mask, and are converted into appropriate printer output:

28	plot sprite at current point on screen
34	plot sprite at specified point on screen
48	plot mask at current point on screen
49	plot mask at specified point on screen
50	plot mask at specified point on screen, scaled
52	plot sprite at specified point on screen, scaled
53	plot sprite at specified point on screen, grey scaled

Scaled characters

The following reason code is mainly used by the VDU drivers to implement sizes other than 8x8 and 8x16 for VDU 5 characters. It is not handled by the printer drivers, which deal with scaled VDU 5 text by another mechanism, and causes an error if encountered during a print job.

51	plot character, scaled

GCOLs

As usual for a printer driver, only some GCOL actions are understood. If the GCOL action is not divisible by 8, nothing is plotted. If it is divisible by 8, the overwrite action is used. If it is divisible by 16, the sprite is plotted without using its mask; otherwise the mask is used.

The colours used to plot sprite pixels are determined as follows:

- If the call does not allow a pixel translation table, or if no translation table is supplied, the current screen palette is consulted to find out what RGB combination the sprite pixel's value corresponds to. The printer driver then does its best to produce that RGB combination. Use of this option is not recommended.

- If a translation table is supplied with the call, the printer driver assumes that the table contains code values allocated by one of the following SWIs:
  ColourTrans_SelectTable with R2 = –1
  ColourTrans_ReturnColourNumber
  ColourTrans_ReturnColourNumberForMode with R1 = –1
  ColourTrans_ReturnOppColourNumber
  ColourTrans_ReturnOppColourNumberForMode with R1 = –1

It can therefore look up precisely which RGB combination is supposed to correspond to each sprite pixel value. Because of the variety of ways in which printer drivers can allocate these values, the translation table should always have been set up in the current print job and using these calls.

Scale

If a sprite is printed unscaled, its size on the printed output is the same as its size would be if it were plotted to the screen in the screen mode that was in effect at the time that the print job concerned was started. If it is printed scaled, the scaling factors are applied to this size. This is one of the few ways in which the printed output does depend on this screen mode. The main other ones are in interpreting GCOL and Tint values, and in interpreting VDU 5 character sizes. It is done this way in the expectation that the application is scaling the sprite for what it believes is the current screen mode.

VDU output

Finally, the following two reason codes are intercepted to keep track of whether plotting output is currently supposed to go to a sprite or to the screen. If it is supposed to go to a sprite, it really will go to that sprite.

60	switch output to sprite
61	switch output to mask

This allows applications to create sprites normally while printing. When output is supposed to go to the screen, it will be processed by the printer driver. Note that printer drivers that redirect output to a sprite internally will treat this case specially, regarding output to that sprite as still being destined for the screen.

DrawV

Printer drivers intercept the DrawV vector and re-interpret those calls whose purpose is to plot something on the screen, producing appropriate printer output instead. There are a number of restrictions on the calls that can be dealt with, mainly due to the limitations of PostScript. Most of the operations that are disallowed are not particularly useful, fortunately.

Colour

Note that the Draw module calls normally use the graphics foreground colour to plot with and the graphics origin. The printer driver uses its versions of these values. In particular, this means that the fill colour is subject to all the restrictions noted elsewhere in this document.

Floating point

The floating point Draw module calls are not intercepted at present. If and when the Draw module is upgraded to deal with them, printer drivers will be similarly upgraded.

Draw_Fill

Printer drivers can deal with most common calls to this SWI. The restrictions are:

- They cannot deal with fill styles that invoke the positive or negative winding number rules – ie those with bit 0 set.
- They cannot deal with a fill style which asks for non-boundary exterior pixels to be plotted (ie bit 2 is set), except for the trivial case in which all of bits 2 - 5 are set (ie all pixels in the plane are to be plotted).
- They cannot deal with the following values for bits 5 - 2:
  0010 – plot exterior boundary pixels only.
  0100 – plot interior boundary pixels only.
  1010 – plot exterior boundary and interior non-boundary pixels only.
- An application should not rely on there being any difference between what is printed for the following three values of bits 5 - 2:
  1000 – plot interior non-boundary pixels only.
  1100 – plot all interior pixels.
  1110 – plot all interior pixels and exterior boundary pixels.

A printer driver will generally try its best to distinguish these, but it may not be possible.

Draw_Stroke

Again, most common calls to this SWI can be dealt with. The restrictions on the parameters depend on whether the specified thickness is zero or not.

If the specified thickness is zero, the restrictions are:

- Printer drivers cannot deal with a fill style with bits 3 - 2 equal to 01 – one that asks for pixels lying off the stroke to be plotted and those that lie on the stroke not to be.

- Most printer drivers will not pay any attention to bit 31 of the fill style, which distinguishes plotting the stroke subpath by subpath from plotting it all at once.

If the specified thickness is non-zero, the restrictions are:

- All the restrictions mentioned under Draw_Fill above.
- They cannot deal with bits 5 - 2 being 0110 – a call asking for just the boundary pixels of the resulting filled path to be plotted.
- Most printer drivers will not pay any attention to bit 31 of the fill style, which distinguishes plotting the stroke subpath by subpath from plotting it all at once.

Draw_StrokePath, Draw_FlattenPath, Draw_TransformPath

None of these do any plotting; they are all dealt with in the normal way by the Draw module.

Draw_ProcessPath

This SWI is faulted if R7=1 (fill path normally) or R7=2 (fill path subpath by subpath) on entry. Use the appropriate one of Draw_Fill or Draw_Stroke if you want to produce printed output. If the operation you're trying to do is too complicated for them, it almost certainly cannot be handled by the PostScript printer driver for example.

If you are using this call to calculate bounding boxes, using the R7=&80000000 +address option, then the matrix, flatness, line thickness, etc., must exactly correspond with those in the intended call. Because of rounding errors, flattening errors, etc., clipping may result if these parameters are different.

All other values of R7 correspond to calls that don't do any plotting and are dealt with in the normal way by the Draw module. If you're trying to do something complicated and you've got enough workspace and RMA, a possible useful trick is to use Draw_ProcessPath with R7 pointing to an output buffer, followed by Draw_Fill on the result.

ColourV

The printer driver intercepts calls to the ColourTrans module, via the ColourV vector. Most of them are passed straight on to the ColourTrans module. The exceptions are:

ColourTrans_SelectTable with R2 = -1

Each RGB combination in the source palette, or implied by it in the case of 256 colour modes, is converted into a colour number as though by ColourTrans_ReturnColourNumber. The resulting values are placed in the table.

ColourTrans_SetGCOL

The printer driver's version of the foreground or background colour is set as appropriate. The GCOL actions are interpreted precisely as for the VDU 18,k,c call. However, rather than looking up a GCOL in the screen palette at plot time, the exact RGB combination specified in this call is remembered and used, as accurately as the printer will render it at plot time.

After this has been done, the call is effectively converted into ColourTrans_ReturnGCOL and passed down to the ColourTrans module in order to set the information returned correctly. Note that this implies that subsequently using the GCOL returned in a VDU 18,k,c sequence will not produce the same effect on the colour as this call. It will merely produce the best approximation the printer can manage to the best approximation the current screen palette can manage to the specified RGB combination. It is therefore probably a bad idea to use the values returned.

This call allows the application to make full use of a printer's colour resolution without having to switch to another screen mode or mess around with the screen's palette, and without worrying about the effects of a change in the screen's palette. It is therefore the recommended way to set the foreground and background colours.

ColourTrans_ReturnColourNumber

This will return a code value, in the range 0 - 255, that identifies the specified RGB combination as accurately as possible to the printer driver. How this code value is determined may vary from printer driver to printer driver, and indeed even from print job to print job for the same printer driver. An application should therefore not make any assumptions about what these code values mean. Most printer drivers implement this by pre-allocating some range of code values to evenly spaced RGB combinations, then adopting the following approach:

- If the RGB combination is already known about, return the corresponding code value.
- If the RGB combination is not already known about and some code values are still free, allocate one of the unused code values to the new RGB combination and return that code value.

- If the RGB combination is not already known about and all code values have been allocated, return the code number whose RGB combination is as close as possible to the desired RGB combination.

The pre-allocation of evenly spaced RGB combinations will ensure that even the third case does not have really terrible results.

ColourTrans_ReturnColourNumberForMode with R1 = -1

This is treated exactly the same as ColourTrans_ReturnColourNumber above.

'Opposite' colours

The printer driver handles 'opposite' colours in a subtly different way to the ColourTrans module. It returns the colour closest to the RGB value most different to that given, whereas ColourTrans returns the colour furthest from the given RGB. This difference will only be obvious if your printer cannot print a very wide range of colours.

ColourTrans_SetOppGCOL

This behaves like ColourTrans_SetGCOL above, except that the RGB combination it remembers is the furthest possible RGB combination from the one actually specified in R0, and it ends by being converted into a call to ColourTrans_ReturnOppGCOL. Note that there is no guarantee that the GCOL returned is anywhere near the RGB combination remembered.

ColourTrans_ReturnOppColourNumber

This behaves exactly as though ColourTrans_ReturnColourNumber had been called with R0 containing the furthest possible RGB combination from the one actually specified.

ColourTrans_ReturnOppColourNumberForMode with R1 = -1

This behaves exactly as though ColourTrans_ReturnColourNumberForMode (see above) had been called with R1 = -1 and R0 containing the furthest possible RGB combination from the one actually specified.

ColourTrans_SetFontColours

The printer driver's version of the font colours is set, to as accurate a representation of the desired RGB combinations as the printer can manage.

Before this is done, the call is passed down to the ColourTrans module to determine the information to be returned. Note that this implies that subsequently using the values returned in a Font_SetFontColours call will not produce the same effect on the font colours as this call. It will merely produce the best approximations the printer can manage to the best approximations the current screen palette can manage to the specified RGB combinations. It is therefore a bad idea to use the values returned.

This call therefore allows the application to make full use of a printer's colour resolution without having to switch to another screen mode or mess around with the screen's palette, and without worrying about the effects of a change in the screen's palette. It is the recommended way to set the font colours.

Font manager SWIs

The printer driver interacts with the font manager via a service call and PDriver_FontSWI in such a way that when it is active, calls to the following SWIs are processed by the printer driver:

- Font_Paint
- Font_LoseFont
- Font_SetFontColours
- Font_SetPalette

This enables the printer driver to make Font_Paint produce printer output rather than affecting the screen.

Font_SetFontColours

The use of Font_SetFontColours is not recommended, as it results in the setting of colours that depend on the current screen palette. Instead, use ColourTrans_SetFontColours to set font colours to absolute RGB values. Similarly, the use of colour-changing control sequences in strings passed to Font_Paint is not recommended.

Font_Paint

How exactly this call operates varies quite markedly between printer drivers. For instance, most dot matrix printer drivers will probably use the font manager to write into the sprite they are using to hold the current strip of printed output, while the PostScript printer driver uses the PostScript prologue to define a translation from font manager font names to printer fonts.

Miscellaneous SWIs

OS_Byte 163

OS_Byte 163,242,0-64 are intercepted to set the printer driver version of the dot pattern repeat length instead of the VDU drivers' version.

OS_Byte 218

OS_Byte 218 is intercepted to act on the printer driver's VDU queue instead of the VDU drivers' version.

OS_ReadVduVariables

It should be noted that most of the informational calls associated with the VDU drivers, and OS_ReadVduVariables in particular, will produce undefined results when a printer driver is active. These results are likely to differ between printer drivers. In particular, they will vary according to whether the printer driver plots to a sprite internally and if so, how large the sprite concerned is.

The only informational calls that the application may rely upon are:

OS_Word 10	used to read character and ECF definitions.
OS_Word 11	used to read palette definitions.
OS_ReadPalette	used to read palette definitions.
OS_Byte 218	when used to read number of bytes in VDU queue.

Error handling

There are a couple of somewhat unusual features about the printer drivers' error handling that an application author should be aware of:

Escape handling

First, Escape condition generation and side effects are turned on within various calls to the printer driver and restored to their original state afterwards. If the application has Escape generation turned off, it is guaranteed that any Escape generated within the print job will be acknowledged and turned into an 'Escape' error. If the application has Escape generation turned on, most Escapes generated within the print job will be acknowledged and turned into 'Escape' errors, but there is a small period at the end of the call during which an Escape will not be acknowledged. If the application makes a subsequent call of one of the relevant types to the printer driver, that subsequent call will catch the Escape. If no such subsequent call is made, the application will need to trap the Escape itself.

The SWIs during which Escape generation is turned on are:

- PDriver_SelectJob for a new job
- PDriver_EndJob
- OS_WriteC
- All ColourTrans SWIs – except ColourTrans_GCOLToColourNumber, ColourTrans_ColourNumberToGCAL, and ColourTrans_InvalidateCache.
- Draw_Fill
- Draw_Stroke
- Font_SetFontColours
- Font_SetPalette
- Font_Paint
- OS_SpriteOp with reason codes:
  PutSprite
  PutSpriteUserCoords
  PutSpriteScaled
  PutSpriteGreyScaled
  PlotMask
  PlotMaskUserCoords
  PlotMaskScaled

All but the first two only apply at times when the printer driver is intercepting plotting calls; that is, at times when all of the following conditions hold:

- There is an active print job.
- Plotting output is directed either to the screen or to a sprite internal to the printer driver.
- The Wimp is not reporting an error. This is as defined by the service call with reason WimpReportError.

Persistent errors

Secondly, inside a number of calls, any error that occurs is converted into a 'persistent error'. The net effect of this is that:

- The error number is left unchanged.
- The error message has the string ' (print cancelled)' appended to it. If it is so long that this would cause it to exceed 255 characters, it is truncated to a suitable length and '… (print cancelled)' is appended to it.
- Any subsequent call to any of the routines concerned will immediately return the same error.

The reason for this behaviour is to prevent errors the application is not expecting from being ignored; for example, quite a lot of code assumes incorrectly that OS_WriteC cannot produce an error. This ensures that an error during OS_WriteC cannot reasonably get ignored forever.

The SWIs during which persistent errors are created are:

- PDriver_EndJob
- PDriver_GiveRectangle
- PDriver_DrawPage
- PDriver_GetRectangle
- OS_WriteC
- All ColourTrans SWIs – except ColourTrans_GCOLToColourNumber, ColourTrans_ColourNumberToGCAL, and ColourTrans_InvalidateCache.
- Draw_Fill
- Draw_Stroke
- Draw_ProcessPath with R7=1
- Font_SetFontColours
- Font_SetPalette
- Font_Paint
- OS_SpriteOp with reason codes:
  PutSprite
  PutSpriteUserCoords
  PutSpriteScaled
  PutSpriteGreyScaled
  PlotMask
  PlotMaskUserCoords
  PlotMaskScaled
  ScreenSave
  ScreenLoad
  PutSprite
  GetSpriteUserCoords
  PaintCharScaled
  SelectSprite in the system sprite area only
  Reason codes unknown to the printer driver

All but the first four only apply at times that the printer driver is intercepting plotting calls. See above for details of this.

PDriver_CancelJob

PDriver_CancelJob puts a print job into a similar state, with the error message being simply 'Print cancelled'. However, this error is only returned by subsequent calls from the list above, not by PDriver_CancelJob itself.

PDriver_AbortJob

Note that an application must respond to any error during a print job that could have come from one of the above sources by calling PDriver_AbortJob. In particular, take care to respond to errors from PDriver_EndJob by calling PDriver_AbortJob, not PDriver_EndJob, otherwise an infinite succession of errors will occur or an unfinished print job will be left around.

Service Calls

Service_Print (Service Call &41)

This service call is for internal use only. You must not use it in your own code.

Service_PDriverStarting (Service Call &65)

Territory manager started

On entry

R1 = &65 (reason code)

On exit

All registers preserved

Use

This is issued when the printer sharer module starts up, and it lets any printer drivers resident at that time declare themselves to the printer sharer module.

Service_PDumperStarting (Service Call &66)

PDumper module starting up

On entry

R1 = &66 (reason code)

On exit

All registers preserved

Use

Issued when the PDumper modules should declare themselves with PDriverDP. The service call is issued when the module is linked into the module chain, so the receiver can use PDriver_MiscOp to register.

Service_PDumperDying (Service Call &67)

PDumper module dying

On entry

R1 = &66 (reason code)

On exit

All registers preserved

Use

Issued as a broadcast to inform PDumpers that they have been deregistered and that the PDriverDP module is about to die.

Service_PDriverGetMessage (Service Call &78)

Get common messages file

On entry

R1 = &78 (reason code)
R2 = 0

On exit

Not claimed

R0 to R8 must be preserved

Call claimed

R1 = 0 (implies service claimed)
R3 = 16 byte MessageTrans block for open messages file

Use

Issued when a PDriver module is looking for the messages file prior to opening it.

If this call is not claimed, then you should attempt to open "Resources:$.Resources.PDrivers.Messages".

Service_PDriverChanged (Service Call &7F)

PDriver has changed

On entry

R1 = &7F (reason code)
R2 = number of new driver

On exit

All registers are preserved

Use

Issued when the PDriver has changed and only then, you will not get repeatedly called when the same PDriver is selected. R2 contains the ID of the printer driver being selected (ie 0 for PostScript, 7 for dot matrix, etc)

SWI Calls

PDriver_Info (SWI &80140)

Get information on the printer driver

On entry

—

On exit

R0 = general type of printer chosen and version number of driver (see below)
R1 = x resolution of printer driven in dots per inch
R2 = y resolution of printer driven in dots per inch
R3 = features word (see below)
R4 = printer device name (a maximum of 20 characters)
R5 = x halftone resolution in repeats/inch. Same as R1 if no halftoning
R6 = y halftone resolution in repeats/inch. Same as R2 if no halftoning
R7 = printer driver specific number identifying the configured printer. This is zero unless it has been changed via the SWI PDriver_SetInfo.

Some of these values can be changed by the SWI PDriver_SetInfo. If PDriver_Info is called while a print job is selected, the values returned are those that were in effect when that print job was started (when it was first selected using PDriver_SelectJob). If PDriver_Info is called when no print job is active, the values returned are the current ones.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI tells an application what the capabilities of the attached printer are. This allows the application to change the way it outputs its data to suit the printer.

Some of the values returned can be changed by the configuration application attached to the printer driver by PDriver_SetInfo.

If this is called while a print job is selected, the values returned are those that were in effect when that print job was started with PDriver_SelectJob. If it is called when no print job is active, the values returned are those that would be used for a new print job.

The value returned in R0 is split in half. The top 16 bits contains the description of which printer driver type is running. The current values it can have are:

0 = PostScript
1 = FX80 or similar
2 = HP LaserJet or compatible
3 = Integrex ColourJet
4 = FAX modem
5 = Dumb laser device
6 = Caspel graphics language
7 = PDumper interface

See the PDumper documentation to find out about the PDumper interface and the declared PDumper numbers. The application author need only talk to PDriver type 7, but the output is redirected to a PDumper.

The bottom 16 bits of R0 have the version number of the printer driver × 100: eg Version 3.21 would be 321 (&0141).

R3 returns a bitfield that describes the available features of the current printer. Most applications shouldn't need to look at this word, unless they wish to alter their output depending on the facilities available.

It is split into several fields:

Bits	Subject
0 - 7	printer driver's colour capabilities
8 - 15	printer driver's plotting capabilities
16 - 23	reserved – must be set to zero
24 - 31	printer driver's optional features

In more detail, each individual bit has the following meaning:

Bit(s)	Value	Meaning
0	0	it can only print in monochrome.
	1	it can print in colour.
1	0	it supports the full colour range – ie it can manage each of the eight primary colours. Ignored if bit 0 = 0.
	1	it supports only a limited set of colours.
2	0	it supports a semi-continuous range of colours at the software level. Also, if bit 0 = 0 and bit 2 = 0, then an application can expect to plot in any level of grey.
	1	it only supports a discrete set of colours at the software level; it does not support mixing, dithering, toning or any similar technique.
3 - 7		reserved and set to zero.
8	0	it can handle filled shapes.
	1	it cannot handle filled shapes other than by outlining them; an unsophisticated XY plotter would have this bit set, for example.
9	0	it can handle thick lines.
	1	it cannot handle thick lines other than by plotting a thin line. An unsophisticated XY plotter would also come into this category. The difference is that the problem can be solved, at least partially, if the plotter has a range of pens of differing thicknesses available.
10	0	it handles overwriting of one colour by another on the paper properly. This is generally true of any printer that buffers its output, either in the printer or the driver.
	1	it does not handle overwriting of one colour by another properly, but only overwriting of the background colour by another. This is a standard property of XY plotters.
11	0	does not support transformed sprite plotting.
	1	supports transformed sprite plotting.
12	0	cannot handle new Font manager features.
12	1	Can handle new Font manager features such as transforms and encodings.
13 - 23		reserved and set to zero.
24	0	it does not support screen dumps.
	1	it does support screen dumps.

25	0	it does not support transformations other than scalings, translations, rotations by multiples of 90 degrees and combinations thereof. These are the transformations supplied to PDriver_DrawPage.
	1	it does support arbitrary transformations supplied to PDriver_DrawPage.
26	0	it does not support the PDriver_InsertIllustration call
	1	it does support the PDriver_InsertIllustration call
27	0	it does not support the SWI PDriver_MiscOp call.
	1	it does support the SWI PDriver_MiscOp call.
28	0	it does not support the SWI PDriver_SetDriver call.
	1	it does support the SWI PDriver_SetDriver call.
29	0	it does not support the SWI PDriver_DeclareFont
	1	it does support the SWI PDriver_DeclareFont

The table below shows the effect of bits 0 - 2 in more detail:

Bit 0	Bit 1	Bit 2	Colours available
0	0	0	Arbitrary greys
0	0	1	A limited set of greys (probably only black and white)
0	1	0	Arbitrary greys
0	1	1	A limited set of greys (probably only black and white)
1	0	0	Arbitrary colours
1	0	1	A limited set of colours, including all the eight primary colours
1	1	0	Arbitrary colours within a limited range (for example, it might be able to represent arbitrary greys, red, pinks and so on, but no blues or greens). This is not a very likely option
1	1	1	A finite set of colours – as for instance an XY plotter might have

The printer name returned in R4 is always terminated by a zero (0) character regardless of what the terminating character was when the name was passed to PDriver_SetInfo. If PDriver_SetInfo has not been called, then R4 will point to a zero length string on return from PDriver_Info.

A copy should be taken of the name at R4 if you intend to use this. With the introduction of multiple printer drivers this name can change.

Related SWIs

None

Related vectors

None

PDriver_SetInfo (SWI &80141)

Configure the printer driver

On entry

R1 = x resolution of printer driven in dots per inch
R2 = y resolution of printer driven in dots per inch
R3 = bit 0 only is used– all other bits are ignored
R4 = a pointer to the new name for the device
R5 = x halftone resolution in repeats/inch (same as R1 if no halftoning)
R6 = y halftone resolution in repeats/inch (same as R2 if no halftoning)
R7 = printer driver specific number identifying the configured printer

On exit

R1 - R3 preserved
R4 preserved
R5 - R7 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call is used by the Printer Manager application on the desktop to set the user requested settings for a specific printer within a general class of printers. The printer name can also be modified, a copy is taken and any future calls to PDriver_Info will return this modified string.

This call only affects print jobs started after it is called. Existing print jobs use whatever values were in effect when they were started.

When monochrome printing, R3 bit zero is set to zero. When colour printing, R3 bit zero is set to one.

This SWI must never be called by any other application.

Related SWIs

PDriver_Info (SWI &80140)

Related vectors

None

PDriver_CheckFeatures (SWI &80142)

Check the features of a printer

On entry

R0 = features word mask
R1 = features word value

On exit

R0, R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

If the features word that PDriver_Info would return in R3 satisfies ((features word) AND R0) = (R1 AND R0), then the return is normal with all registers preserved. Otherwise a suitable error is generated if appropriate. For example, no error will be generated if the printer driver has the ability to support arbitrary rotations and your features word value merely requests axis preserving ones.

Related SWIs

PDriver_Info (SWI &80140)

Related vectors

None

PDriver_PageSize (SWI &80143)

Find how large paper and print area is

On entry

—

On exit

R1 = x size of paper, including margins
R2 = y size of paper, including margins
R3 = left edge of printable area of paper
R4 = bottom edge of printable area of paper
R5 = right edge of printable area of paper
R6 = top edge of printable area of paper

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

An application can use this call to find out how big the paper in use is and how large the printable area on the paper is. This information can then be used to decide how to place the data to be printed on the page.

These values can be changed by the configuration application associated with the printer driver (using PDriver_SetPageSize). If PDriver_PageSize is called while a print job is selected, the values returned are those that were in effect when that print job was started (ie when it was first selected using PDriver_SelectJob). If PDriver_PageSize is called when no print job is active, the values returned are those that would be used for a new print job.

All units are in millipoints, and R3 - R6 are relative to the bottom left corner of the page.

Related SWIs

PDriver_SetPageSize (SWI &80144)

Related vectors

None

PDriver_SetPageSize (SWI &80144)

Set how large paper and print area is

On entry

R1 = x size of paper, including margins
R2 = y size of paper, including margins
R3 = left edge of printable area of paper
R4 = bottom edge of printable area of paper
R5 = right edge of printable area of paper
R6 = top edge of printable area of paper

On exit

R1 - R6 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The configuration application associated with a particular printer driver uses this SWI to change the page size values associated with subsequent print jobs.

It must never be called by any other application.

All units are in millipoints, and R3 - R6 are relative to the bottom left corner of the page.

Related SWIs

PDriver_PageSize (SWI &80143)

Related vectors

None

PDriver_SelectJob (SWI &80145)

Make a given print job the current one

On entry

R0 = file handle for print job to be selected, or zero to cease having any print job selected.
R1 = zero or points to a title string for the job

On exit

R0 = file handle for print job that was previously active, or zero if no print job was active.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

A print job is identified by a file handle, which must be that of a file that is open for output. The printer output for the job concerned is sent to this file.

Calling PDriver_SelectJob with R0=0 will cause the current print job (if any) to be suspended, and the printer driver will cease intercepting plotting calls.

Calling PDriver_SelectJob with R0 containing a file handle will cause the current print job (if any) to be suspended, and a print job with the given file handle to be selected. If a print job with this file handle already exists, it is resumed; otherwise a new print job is started. The printer driver will start to intercept plotting calls if it is not already doing so.

Note that this call never ends a print job. To do so, use one of the SWIs PDriver_EndJob or PDriver_AbortJob.

The title string pointed to by R1 is treated by different printer drivers in different ways. It is terminated by any character outside the range ASCII 32 - 126. It is only ever used if a new print job is being started, not when an old one is being resumed. Typical uses are:

- A simple printer driver might ignore it.
- The PostScript printer driver adds a line '%%Title:' followed by the given title string to the PostScript header it generates.
- Printer drivers whose output is destined for an expensive central printer in a large organisation might use it when generating a cover sheet for the document.

An application is always entitled not to supply a title (by setting R1=0), and a printer driver is entitled to ignore any title supplied.

Printer drivers may also use the following OS variables when creating cover sheets, etc:

PDriver$For	indicates who the output is intended to go to
PDriver$Address	indicates where to send the output.

These variables must not contain characters outside the range ASCII 32 - 126.

If an error occurs during PDriver_SelectJob, the previous job will still be selected afterwards, though it may have been deselected and reselected during the call. No new job will exist. One may have been created during the call, but the error will cause it to be destroyed again.

Related SWIs

PDriver_CurrentJob (SWI &80146), PDriver_EndJob (SWI &80148), PDriver_AbortJob (SWI &80149), PDriver_Reset (SWI &8014A)

Related vectors

None

PDriver_CurrentJob (SWI &80146)

Get the file handle of the current job

On entry

—

On exit

R0 = file handle

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

R0 returns the file handle for the current active print job, or zero if no print job is active.

Related SWIs

PDriver_SelectJob (SWI &80145), PDriver_EndJob (SWI &80148), PDriver_AbortJob (SWI &80149), PDriver_Reset (SWI &8014A)

Related vectors

None

PDriver_FontSWI (SWI &80147)

Internal call

On entry

—

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI is part of the internal interface between the font system and printer drivers. Applications must not call it.

Related SWIs

None

Related vectors

None

PDriver_EndJob (SWI &80148)

End a print job normally

On entry

R0 = file handle for print job to be ended

On exit

R0 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI should be used to end a print job normally. This may result in further printer output – for example, the PostScript printer driver will produce the standard trailer comments.

If the print job being ended is the currently active one, there will be no current print job after this call, so plotting calls will no longer be intercepted.

If the print job being ended is not currently active, it will be ended without altering which print job is currently active or whether plotting calls are being intercepted.

Related SWIs

PDriver_SelectJob (SWI &80145), PDriver_CurrentJob (SWI &80146), PDriver_AbortJob (SWI &80149), PDriver_Reset (SWI &8014A)

Related vectors

None

PDriver_AbortJob (SWI &80149)

End a print job without any further output

On entry

R0 = file handle for print job to be aborted

On exit

R0 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI should be used to end a print job abnormally. It should generally be called after errors while printing. It will not try to produce any further printer output. This is important if an error occurs while sending output to the print job's output file.

If the print job being aborted is the currently active one, there will be no current print job after this call, so plotting calls will no longer be intercepted.

If the print job being aborted is not currently active, it will be aborted without altering which print job is currently active or whether plotting calls are being intercepted.

Related SWIs

PDriver_SelectJob (SWI &80145), PDriver_CurrentJob (SWI &80146), PDriver_EndJob (SWI &80148), PDriver_Reset (SWI &8014A)

Related vectors

None

PDriver_Reset (SWI &8014A)

Abort all print jobs

On entry

—

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI aborts all print jobs known to the printer driver, leaving the printer driver with no active or suspended print jobs and ensuring that plotting calls are not being intercepted.

Normal applications shouldn't use this SWI, but it can be useful as an emergency recovery measure when developing an application.

A call to this SWI is generated automatically if the machine is reset or the printer driver module is killed or RMTidy'd.

Related SWIs

PDriver_SelectJob (SWI &80145), PDriver_CurrentJob (SWI &80146), PDriver_EndJob (SWI &80148), PDriver_AbortJob (SWI &80149)

Related vectors

None

PDriver_GiveRectangle (SWI &8014B)

Specify a rectangle to be printed

On entry

R0 = rectangle identification word
R1 = pointer to 4 word block, containing rectangle to be plotted in OS units.
R2 = pointer to 4 word block, containing transformation table
R3 = pointer to 2 word block, containing the plot position.
R4 = background colour for this rectangle, in the form &BBGGRRXX.

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI allows an application to specify a rectangle from its workspace to be printed, how it is to be transformed and where it is to appear on the printed page.

The word in R0 is reported back to the application when it is requested to plot all or part of this rectangle.

The value passed in R2 is the dimensionless transformation to be applied to the specified rectangle before printing it. The entries are given as fixed point numbers with 16 binary places, so the transformation is:

$$x' = (x * R2!0 + y * R2!8)/2^{16}$$
$$y' = (x * R2!4 + y * R2!12)/2^{16}$$

The value passed in R3 is the position where the bottom left corner of the rectangle is to be plotted on the printed page in millipoints.

An application should make one or more calls to PDriver_GiveRectangle before a call to PDriver_DrawPage and the subsequent calls to PDriver_GetRectangle. Multiple calls allow the application to print multiple rectangles from its workspace to one printed page – for example, for 'two up' printing.

The printer driver may subsequently ask the application to plot the specified rectangles or parts thereof in any order it chooses. An application should not make any assumptions about this order or whether the rectangles it specifies will be split. A common reason why a printer driver might split a rectangle is that it prints the page in strips to avoid using excessively large page buffers.

Assuming that a non-zero number of copies is requested and that none of the requested rectangles go outside the area available for printing, it is certain to ask the application to plot everything requested at least once. It may ask for some areas to be plotted more than once, even if only one copy is being printed, and it may ask for areas marginally outside the requested rectangles to be plotted. This can typically happen if the boundaries of the requested rectangles are not on exact device pixel boundaries.

If PDriver_GiveRectangle is used to specify a set of rectangles that overlap on the output page, the rectangles will be printed in the order of the PDriver_GiveRectangle calls. For appropriate printers (ie most printers, but not XY plotters for example), this means that rectangles supplied via later PDriver_GiveRectangle calls will overwrite rectangles supplied via earlier calls.

The rectangle specified should be a few OS units larger than the 'real' rectangle, especially if important things lie close to its edge. This is because rounding errors are liable to appear when calculating bounding boxes, resulting in clipping of the image. Such errors tend to be very noticeable, even when the amounts involved are small.

However, you shouldn't make the rectangle a lot larger than the real rectangle. This will result in slowing the process down and use of unnecessarily large amounts of memory. Also, some subsequent users may scale the image according to this rectangle size (say to use some PostScript as an illustration in another document), resulting in it being too small.

Related SWIs

PDriver_GetRectangle (SWI &8014D)

Related vectors

None

PDriver_DrawPage (SWI &8014C)

Called after all rectangles plotted to draw the page

On entry

R0 = number of copies to print
R1 = pointer to 4 word block, to receive the rectangle to print
R2 = page sequence number within the document, or 0
R3 = zero or points to a page number string

On exit

R0 = non-zero if rectangle required, zero if finished
R1 = preserved
R2 = rectangle identification word if R0 is non-zero
R3 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI should be called after all rectangles to be plotted on the current page have been specified using PDriver_GiveRectangle. It returns the first rectangle that the printer driver wants plotted in the area. If nothing requires plotting it will indicate the end of the list.

R2 on entry is zero or contains the page's sequence number within the document being printed (ie. 1–n for an n-page document).

R3 on entry is zero or points to a string, terminated by a character in the ASCII range 33 - 126, which gives the text page number: for example '23', 'viii', 'A–1'. Note that spaces are not allowed in this string.

If R0 on exit is non-zero, the area pointed to by R1 has been filled in with the rectangle that needs to be plotted, and R2 contains the rectangle identification word for the user-specified rectangle that this is a part of. If R0 is zero, the contents of R2 and the area pointed to by R1 are undefined. The rectangle in R1 is in user coordinates before transformation.

The application should stop trying to plot the current page if R0=0, and continue otherwise. If R0<>0, the fact that R0 is the number of copies still to be printed is only intended to be used for information purposes – for example, putting a 'Printing page m of n' message on the screen. Note that on some printer drivers you cannot rely on this number changing incrementally, ie it may suddenly go from 'n' to zero. As long as it is non-zero, R0's value does not affect what the application should try to plot.

The information passed in R2 and R3 is not particularly important, though it helps to make output produced by the PostScript printer driver conform better to Adobe's structuring conventions. If non-zero values are supplied, they should be correct. Note that R2 is NOT the sequence number of the page in the print job, but in the document.

An example: if a document consists of 11 pages, numbered '' (the title page), 'i'–'iii' and '1'–'7', and the application is requested to print the entire preface part, it should use R2 = 2, 3, 4 and R3 → 'i', 'ii', 'iii' for the three pages.

When plotting starts in a rectangle supplied by a printer driver, the printer driver behaves as though the VDU system is in the following state:

- VDU drivers enabled.
- VDU 5 state has been set up.
- all graphics cursor positions and the graphics origin have been set to (0,0) in the user's rectangle coordinate system.
- a VDU 5 character size and spacing of 16 OS units by 32 OS units have been set in the user's rectangle coordinate system.
- the graphics clipping region has been set to bound the actual area that is to be plotted. But note that an application cannot read what this area is: the printer drivers do not and cannot intercept OS_ReadVduVariables or OS_ReadModeVariable.
- the area in which plotting will actually take place has been cleared to the background colour supplied in the corresponding PDriver_GiveRectangle call, as though the background had been cleared.
- the cursor movement control bits (ie the ones that would be set by VDU 23,16,...) are set to &40 – so that cursor movement is normal, except that movements beyond the edge of the graphics window in VDU 5 mode do not generate special actions.

- one OS unit on the paper has a nominal size of 1/180 inch, depending on the transformation supplied with this rectangle.

This is designed to be as similar as possible to the state set up by the window manager when redrawing.

Related SWIs

None

Related vectors

None

PDriver_GetRectangle (SWI &8014D)

Get the next print rectangle

On entry

R1 = pointer to 4 word block, to receive the print rectangle

On exit

R0 = number of copies still requiring printing, or zero if no more plotting
R1 = preserved
R2 = rectangle identification word if R0 is non-zero

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI should be used after plotting a rectangle returned by a previous call to PDriver_DrawPage or PDriver_GetRectangle, to get the next rectangle the printer driver wants plotted. It returns precisely the same information as PDriver_DrawPage.

If R0 is non-zero, the area pointed to by R1 has been filled in with the rectangle that needs to be plotted, and R2 contains the rectangle identification word for the user-specified rectangle that this is a part of. If R0 is zero, the contents of R2 and the area pointed to by R1 are undefined.

Related SWIs

None

Related vectors

None

PDriver_CancelJob (SWI &8014E)

Stops the print job associated with a file handle from printing

On entry

R0 = file handle for job to be cancelled

On exit

R0 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re–entrancy

Not defined

Use

This SWI causes subsequent attempts to output to the print job associated with the given file handle to do nothing other than generate the error 'Print cancelled'. An application is expected to respond to this error by aborting the print job. Generally, this call is used by applications other than the one that started the job.

Related SWIs

PDriver_AbortJob (SWI &80149)

Related vectors

None

PDriver_ScreenDump (SWI &8014F)

Output a screen dump to the printer

On entry

R0 = file handle of file to receive the dump

On exit

R0 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

If this SWI is supported (ie if bit 24 is set in the value PDriver_Info returns in R3), this SWI causes the printer driver to output a screen dump to the file handle supplied in R0. The file concerned should be open for output.

If the SWI is not supported, an error is returned.

Related SWIs

None

Related vectors

None

PDriver_EnumerateJobs (SWI &80150)

List existing print jobs

On entry

R0 = zero to get first, or previous handle to get next print job handle

On exit

R0 = next print job handle, or zero if there are no more in the list

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This allows the print jobs that exist to be enumerated. The order in which they appear is undefined.

Related SWIs

None

Related vectors

None

PDriver_SetPrinter (SWI &80151)

Set printer driver specific options

On entry

Printer driver specific

On exit

Printer driver specific

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This allows the setting of options specific to a particular printer driver. In general, this SWI is used by the configuration application associated with the printer driver module and no other application should use it.

This SWI has now been replaced by the RISC OS 3 SWI PDriver_SetDriver.

Related SWIs

None

Related vectors

None

PDriver_CancelJobWithError (SWI &80152)

Cancels a print job – future attempts to output to it generate an error

On entry

R0 = file handle for job to be cancelled
R1 = pointer to error block

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI causes subsequent attempts to output to the print job associated with the given file handle to do nothing other than generate the specified error. An application is expected to respond to this error by aborting the print job.

This SWI only exists in versions 2.00 and above of the printer driver module (which is present in versions 1.00 and above of the printer driver application).

Related SWIs

None

Related vectors

None

PDriver_SelectIllustration (SWI &80153)

Makes the given print job the current one, and treats it as an illustration

On entry

R0 = file handle for print job to be selected, or 0 to deselect all jobs
R1 = pointer to title string for job, or 0

On exit

R0 = file handle for previously active print job, or 0 if none was active

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call does exactly the same thing as PDriver_SelectJob, except when it used to start a new print job. In this case, the differences are:

- The print job started must contain exactly one page; if it doesn't, an error will be generated.
- Depending on the printer driver involved, the output generated may differ. (For instance, the PostScript printer driver will generate Encapsulated PostScript output for a job started this way.)

The intention of this SWI is that it should be used instead of PDriver_SelectJob when an application is printing a single page that is potentially useful as an illustration in another document.

This SWI only exists in versions 2.00 and above of the printer driver module (which is present in versions 1.00 and above of the printer driver application).

Related SWIs

None

Related vectors

None

PDriver_InsertIllustration (SWI &80154)

Inserts a file containing an illustration into the current job's output

On entry

R0 = file handle for file containing illustration.
R1 = pointer to Draw module path to be used as a clipping path, or 0 if no clipping is required.
R2 = x coordinate of where the bottom left corner of the illustration is to go.
R3 = y coordinate of where the bottom left corner of the illustration is to go.
R4 = x coordinate of where the bottom right corner of the illustration is to go.
R5 = y coordinate of where the bottom right corner of the illustration is to go.
R6 = x coordinate of where the top left corner of the illustration is to go.
R7 = y coordinate of where the top left corner of the illustration is to go.

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

If this SWI is supported (bit 26 is set in the value SWI PDriver_Info returns in R3), it allows an external file containing an illustration, such as an Encapsulated PostScript file, to be inserted into the current job's output. The format of such an illustration file depends on the printer driver concerned, and many printer drivers won't support any sort of illustration file inclusion at all.

All coordinates in the clipping path and in R2 - R7 are in 256ths of an OS unit, relative to the PDriver_GiveRectangle rectangle currently being processed.

This SWI only exists in versions 2.00 and above of the printer driver module (which is present in versions 1.00 and above of the printer driver application).

Related SWIs

None

Related vectors

None

PDriver_DeclareFont (SWI &80155)

Allows fonts used to be declared

On entry

R0 = handle of font to be declared (or zero)
R1 = font name to be declared
R2 = flags word
bit 0 set = don't download font if not present within device
bit 1 set = when font is used kerning is applied

On exit

V set
R0 = error block, else all preserved

Interrupts

?

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call allows the fonts that will be used within a document to be declared, this is important for both the downloading and structuring of PostScript files.

The SWI needs to be called after the PDriver_SelectJob, but prior to any PDriver_DrawPage request. It should be called once for each distinct font name and encoding to be used, but not for each font size (or colour, etc.), the printer driver capabilities word has a bit which indicates whether this SWI is supported, bit 29 will be set if this is the case, applications should check this prior to calling.

If this SWI is not called at all, the results are printer driver dependant. PDriverDP does not care in the least whether you call this SWI or not. PDriverPS on the other hand, will care, and will perform default actions configured by the user, including which fonts are already in the printer and which fonts to download to the printer.

The font is declared by either its RISC OS font handle or its font name, if the handle specified in R0 is zero then R1 is assumed to be a pointer to the font name. This should specify any encoding to be used and any find matrix (/E or /M when calling Font_FindFont).

When an application attempts to print a document containing fonts, it should call PDriver_DeclareFont once for each font to be printed. The font name passed to this call should be exactly the same as the one passed to Font_FindFont, including any encoding and matrix information. If the document does not use fonts, then it should call the SWI just once, with the end-of-list value of 0, to indicate this fact. (Otherwise the printer driver takes special action on the assumption that the application is unaware of the PDriver_DeclareFont call).

The flags word describes other information about the font, bit 0 is used to stop the downloading of this font, if it is not downloaded then it will be substituted with a resident font, usually Courier (although this is driver specific). Bit 1 is used to indicate if kerning is applied, this is very important for the PostScript printer driver which needs to download kerning information about the font.

Related SWIs

None

Related vectors

None

PDriver_DeclareDriver (SWI &80156)

Declares a driver to the sharing system.

On entry

R0 = reason code handle for device
R1 = private word for device (passed in R12)
R2 = general printer driver type

On exit

V set
R0 = error block, else all preserved

Interrupts

?

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call is handled by the PDriver Sharer module, it is used to register another printer driver within the system. This driver can be selected using the PDriver_SelectDriver SWI. Duplicate printer devices are not allowed and an error will be generate if one device is not already registered.

The caller passes in a pointer to a routine to be called to handle the decoding of reason codes and a pointer to a private word. When the printer driver is called the following are setup:

On entry

R11 = reason code (0 - 31)
R12 = pointer to private word
R13 = return address

On exit

V clear as documented for call
V set R0 = pointer to error block

The reason code passed in will be in the range of 0 - 31, and maps directly onto the SWI number, the following calls will never be seen by the registered device:

PDriver_DeclareDriver
PDriver_RemoveDriver
PDriver_SelectDriver
PDriver_EnumerateDrivers

Any SWIs directed directly at a specific driver will be decoded down to a normal call and then passed down to the decoder.

The service call Service_PDriverStarting is issued when the sharer module is installed so that the PDriver modules installed can call this SWI to register themselves.

Related SWIs

None

Related vectors

None

PDriver_RemoveDriver (SWI &80157)

Removes a printer driver from the sharing system.

On entry

R0 = global printer type (i.e. zero for PostScript)

On exit

V set
R0 = error block, else all preserved

Interrupts

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call will deregister a printer driver, doing this calls all jobs associated with this device. It is strongly suggested that a PDriver module checks that it has no jobs pending prior to calling this SWI as doing so in such a situation will result in confusing and possibly crashing applications which currently think that a printer driver is present.

Related SWIs

None

Related vectors

None

PDriver_SelectDriver (SWI &80158)

Selects the specified driver

On entry

R0 = global printer type or -1 to read current device

On exit

R0 = previous global printer type or -1 if none

Interrupts

?

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

Selects the specified driver, returning an error if the driver has not been registered.

This is not designed for use by applications authors and should only be used by the Printer Manager application.

If for any reason your printer driver needs to use this call it should attempt always restore the printer driver as requied, or use the PDriver<foo>ForDriver feature.

Related SWIs

None

Related vectors

None

PDriver_EnumerateDrivers (SWI &80159)

Enumerate all drivers within the system.

On entry

R0 = next printer (zero gives first printer in list)

On exit

If V = 1 then R0 = Error block

else

R0 = returns the next record (zero is no record)
R1 = driver type

Interrupts

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call allows you to enumerate all the drivers within the system, scanning from the first (with R0 specified as 0 on entry), until R0 is returned to zero.

Related SWIs

None

Related vectors

None

PDriver_MiscOp (SWI &8015A)

PDriver_MiscOpForDriver (SWI &8015B)

Processing of miscellaneous PDriver operations

On entry

R0 = reason code
= 0 add fonts
= 1 remove fonts
= 2 enumerate fonts

If PDriver_MiscOpForDriver
R8 = Identifier for the driver

On exit

V set
R0 = Error block, otherwise defined by call.

V clear
Depends on reason code.

Interrupts

?

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call allows the processing of miscellaneous printer driver operations.

Reason codes marked with bit 31 clear are processed by the device, any with bit 31 set are passed through to the device dependant code to be handled.

The first three reason codes are to do with the handling of fonts within the printer driver environment. When a no job is selected then they will set the state for the next job created, if a job is selected then these values get written to the current job.

Add font (0)

On entry

R0 = 0 (reason code)
R1 = RISC OS font name (terminated with control code)
R2 = Name to associate with it (terminated with control code) / =0 for none
R3 = word to associate with it
bit 0 set = font is resident within device
bit 1 set = font to be downloaded at job start
bit 2 set = font has been downloaded
bits 3 to 31 reserved and should be zero

R4 = word for adding font
If bit 0 set, overwrite existing entries

On exit

If V set, then R0 = Error block

If V clear, then all registers are preserved

This code adds a font to either the global list or the local list associated with a job.

The global list is the list of fonts known by the printer and the local list is the one associated with each job and describes the fonts and their mappings within the job. Each record is stored as a separate block within the RMA. Blocks within the global list are copied to each job when PDriver_SelectJob is called.

On entry R1 contains the pointer to the RISC OS name, ideally this will contain the encoding vector used, ie. /F<font name> /E<encoding>, you can also include matrix information for derived fonts. This name is case insensitive. Duplicate names are also filtered out.

R2 contains a pointer to the Alien name to be associated with the RISC OS name, this is used by the printer dependant code as required.

R3 is a flags word to be used by the printer dependent code, see specific printer documentation for further details.

R4 contains a flags word to associate with the addition of the record, currently only bit 0 is used and all others should be zero.

The two strings specified are control code terminated, the RISC OS name is case insensitive and the name associated with this is case sensitive. All reserved flags should be set to zero.

If an RISC OS name is already registered then its data will be overwritten with the new data specified.

Remove fonts (1)

On entry

R0 = 1 (reason code)
R1 -> RISC OS name to be removed / =0 for all

On exit

If V set, then R0 = Error block

If V clear, then all registers are preserved

If a print job is currently selected, this call removes the fonts associated with the print job. If no print job is selected then all fonts are removed. R1 should be the pointer to the name to be removed or zero if all fonts are to be removed.

No error is generated if all names are to be removed but none are registered, but an error will be generated if a specific name is being removed but is not present.

Enumerate fonts (2)

On entry

R0 = 2 (reason code)
R1 -> return buffer (zero for maximum size of buffer needed)
R2 = size of return buffer (zero for maximum size of buffer needed)
R3 = handle (for first call set to zero)
R4 = Flags
all bits reserved and should be zero

On exit

V set, r0 -> error block.

If V clear:

if R1 <> 0 on entry then;

[R1] +0 -> RISC OS name
+4 -> Alien name
+8 = Flags word

...until buffer is full

R1 -> free byte in buffer
R2 = number of free bytes in buffer (<12)
R3 = handle to be passed to read rest of data, =0 if none

else;

R1 preserved.
R2 = maximum size of buffer to return data.
R3 preserved.
R4 preserved.

Fill the buffer with three word records listing the fonts that have been added to either the global list or the local list stored with the job. The routine accepts a pointer to the buffer which if zero returns the size of buffer required; in fact it is R3 +size to allow you to pre-allocate room for a header if needed.

The buffer is filled until the size is <12, all pointers point to blocks stored within the RMA. Ideally a copy should be made of these strings as someone could perform a remove list call to zap them.

R3 on entry for the first call should contain zero and then passed in on subsequent calls to read the remaining data. When the last object has been read it will be returned as zero.

Related SWIs

None

Related vectors

None

PDriver_SetDriver SWI 8015C

Sets the specific printer driver.

On entry

More information needed for this SWI!!!

On exit

Interrupts

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

Related SWIs

None

Related vectors

None

Application Notes

This is an example BASIC procedure that does a standard 'two up' printing job:

```
DEFPROCprintout(firstpage%, lastpage%, title$, filename$)
REM Get SWI numbers used in this procedure.
LOCAL select%, abort%, pagesize%, giverect%, drawpage%, getrect%, end%
SYS "OS_SWINumberFromString",,"PDriver_SelectJob" TO select%
SYS "OS_SWINumberFromString",,"PDriver_AbortJob" TO abort%
SYS "OS_SWINumberFromString",,"PDriver_PageSize" TO pagesize%
SYS "OS_SWINumberFromString",,"PDriver_GiveRectangle" TO giverect%
SYS "OS_SWINumberFromString",,"PDriver_DrawPage" TO drawpage%
SYS "OS_SWINumberFromString",,"PDriver_GetRectangle" TO getrect%
SYS "OS_SWINumberFromString",,"PDriver_EndJob" TO end%
:
REM Open destination file and set up a local error handler that
REM will close it again on an error.
LOCAL H%, O%
H% = OPENOUT(filename$)
LOCAL ERROR
ON ERROR LOCAL:RESTORE ERROR:CLOSE#H%:PROCpasserror
:
REM Start up a print job associated with this file, remembering the
REM handle associated with the previous print job (if any), then
REM set up a local error handler for it.
SYS select%,H%,title$ TO O%
LOCAL ERROR
ON ERROR LOCAL:RESTORE ERROR:SYSabort%,H%:SYSselect%,O%:PROCpasserro
:
REM Now we decide how two pages are to fit on a piece of paper.
LOCAL left%, bottom%, right%, top%
PROCgetdocumentsize(box%)
SYS pagesize% TO ,,,left%,bottom%,right%,top%
PROCfittwopages(left%,bottom%,right%,top%,box%,matrix%,origin1%,origin2%)
:
REM Start the double page loop
LOCAL page%, copiesleft%, pagetoprint%, white%
white%=&FFFFFF00
:
FOR page%=firstpage% TO lastpage% STEP 2
:
  REM Set up to print two pages, or possibly just one last timearound.
  SYS giverect%, page%, box%, matrix%, origin1%, white%
  IF page%<lastpage% THEN
    SYS giverect%, page%+1, box%, matrix%, origin2%, white%
  ENDIF
:
  REM Start printing. As each printed page corresponds to two document pages,
  REM we cannot easily assign any sensible page numbers to printed pages.
  REM So we simply pass zeroes to PDriver_DrawPage.
  SYS drawpage%,1,box2%,0,0 TO copiesleft%,,pagetoprint%
  WHILE copiesleft%<>0
    PROCprintpage(pagetoprint%, box2%)
    SYS getrect%,,box% TO copiesleft%,,pagetoprint%
```

```
  ENDWHILE
:
REM End of page loop
NEXT
:
REM All pages have now been printed. Terminate this print job.
SYS end%,H%
:
REM Go back to the first of our local error handlers.
RESTORE ERROR
:
REM And go back to whatever print job was active on entry to this procedure
REM (or to no print job in no print job was active).
SYS select%,O%
:
REM Go back to the caller's error handler.
RESTORE ERROR
REM Close the destination file.
CLOSE#H%
ENDPROC
:
DEFPROCpasserror
ERROR ERR,REPORT$+" (from line "+STR$(ERL)+")"
ENDPROC
```

Notes

This uses the following global areas of memory:

`box%`	4 words
`box2%`	4 words
`matrix%`	4 words
`origin1%`	2 words
`origin2%`	2 words

And the following external procedures:

`DEFPROCgetdocumentsize(box%)`

- fills the area pointed to by `box%` with the size of a document page in OS units.

`DEFPROCfittwopages(l%, b%, r%, t%, box%, transform%, org1%, org2%)`

- given left, bottom, right and top bounds of a piece of paper, and a bounding box of a document page in OS units, sets up a transformation and two origins in the areas pointed to by `tr%`, `org1%` and `org2%` to print two of those pages on a piece of paper.

`DEFPROCdrawpage(page%, box%)`

- draw the parts of document page number 'page%' that lie with the box held in the 4 word area pointed to by 'box%'.

If printing is likely to take a long time and the application does not want to hold other applications up while it prints, it should regularly use a sequence like the following during printing:

```
SYS select%,O%
SYS "Wimp_Poll",mask%,area% TO reason%
...
```

process reason% as appropriate

```
...
SYS select%,H% TO O%
```

58 MessageTrans

Introduction and Overview

The MessageTrans module provides facilities for you to separate text messages from the main body of an application. The messages are held in a text file, and the application refers to them using tokens.

Using this module makes it much easier to prepare versions of your program to supply to different international markets. Changing your application's textual output becomes a simple matter of editing its messages file using your favourite text editor.

Summary of MessageTrans facilities

The module provides SWIs to

- get information about a message file
- open a message file
- look up a text message in the file by its token
- look up a text message in the file by its token, and then GSTrans it
- look up a text message in the file by its token, and convert it to an error block
- look up text messages in the file by their tokens, and convert them to a menu structure
- close a message file.

It also provides a service call to ease the handling of message files over (for example) a module reinitialisation.

Technical Details

Message file descriptors

MessageTrans uses a *file descriptor* to refer to message files. A file descriptor consists of a 4-word data structure. A file descriptor is always passed to MessageTrans as a pointer to this data structure.

We recommend that when your application stores a file descriptor, it uses a fifth word to keep a record of the file's status (ie whether or not it is open).

Standard RISC OS messages

If MessageTrans is passed a null pointer to a file descriptor, it uses a file of standard RISC OS messages, held in Resources:$.Resources.Global.Messages. Obviously, if any of these messages are suitable for your application, you should use them; this will save on RAM usage, and on any future effort in translating these messages.

Message file format

Message files contain a series of one-line token / value pairs, terminated by character 10 (an ASCII linefeed).

```
<file>        ::=   { <line> }*
<line>        ::=   <tokline> | '#' <comment><nl> | <nl>
<tokline>     ::=   <token> { '/'<token> | <nl><token> }* : <value><nl>
<token>       ::=   <tokchar> { <tokchar> }*
<tokchar>     ::=   <char> | <wildcard>
<char>        ::=   any character > ' ' except ',', ')', ':', '?' or '/'
<wildcard>    ::=   '?' (matches any character)
<comment>     ::=   { <anychar> }*
<anychar>     ::=   any character except <nl>
<nl>          ::=   character code 10
<value>       ::=   { <anychar> | '%0' | '%1' | '%2' | '%3' | '%%' }*
```

Note that the spaces in the above description are purely to improve readability – in fact spaces are significant inside tokens, so should only really appear in <comment> and <value>.

Alternative tokens

Alternative tokens are separated by '/' or <nl>. If any of the alternative tokens before the next ':' in the file match the token supplied in a call, the value after the next ':' up to the following <nl> is returned.

Wildcards

The '?' character in a token in the file matches any character in the token supplied to be matched.

Case significance

Case is significant.

Parameter substitution

Most MessageTrans SWIs support parameter substitution. If R2 is not 0 on entry, '%0', '%1', '%2' and '%3' are substituted with the parameters supplied in R4...R7, except where the relevant register is 0, in which case the text is left alone. '%%' is converted to '%'; otherwise if no parameter substitution occurs the text is left alone. No other substitution is performed on the string.

Example file

```
# This is an example message file
TOK1:This value is obtained only for "TOK1".
TOK2
TOK3/TOK4:This value is obtained for "TOK2","TOK3" or "TOK4"
TOK?:This value is obtained for "TOK<not 1,2,3 or 4>"
ANOTHER:Parameter in R4 = %0, parameter in R5 = %1.
MENUTITLE:Title of menu
MENUITEM1:First item in menu
MENUITEM2:Second item in menu
MENUITEM3:Third item in menu
```

Unmatchable tokens

There are a number of actions MessageTrans may take if it fails to find a match in the specified file. In order they are:

1. Search for the token in the file of standard RISC OS messages.
   It only does so for certain calls, as stated in their documentation.
2. Use a default string (see below).
3. Generate an error (see below).

Supplying default strings

Whenever you have to supply MessageTrans with a token to be matched, you can also supply a default string to be used if MessageTrans is unable to match the token. The syntax is:

token:default

That is, the token and its default value are separated by a ':'. The default value must be null terminated.

Errors

MessageTrans generates the error 'Message token not found' if it is totally unable to supply any string equivalent to a token. This error is also given if the string to be returned is on the last line of the file, and does not have a terminating ASCII linefeed.

Service_Reset

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 a soft reset.

Service Call

Service_MessageFileClosed (Service Call &5E)

From MessageTrans module

On entry

R0 = 4-word data structure passed to MessageTrans_OpenFile
R1 = &5E (reason code)

On exit

All registers are preserved

Use

If the application recognises the value of R0 passed in, and it has any direct pointers into the message data that it relates to, it should re-initialise itself by calling MessageTrans_OpenFile again to re-open the file, and recache its pointers. If it has used MessageTrans_MakeMenus, it should call Wimp_GetMenuState to see if its menu tree it open, and delete it using Wimp_CreateMenu(-1) if so.

This service call is only ever issued if the file is not held in the user's own buffer. It tells the application that its file data has been thrown away, for example if the file is held inside a module which is then reloaded.

It is only necessary to trap this service call if direct pointers into the file data are being used. Otherwise, the MessageTrans module will make a note in the file descriptor that the file has been closed, and simply re-open it when MessageTrans_Lookup or MessageTrans_MakeMenus is next called on that file.

It is recommended that applications that cannot trap service calls do not use direct pointers into the file data (eg indirected icons with MessageTrans_MakeMenus). They can still use such indirected icons, if they provide a buffer pointer in R2 on entry to MessageTrans_OpenFile (so that the message file data is copied into the buffer)

SWI Calls

MessageTrans_FileInfo (SWI &41500)

Gives information about a message file

On entry

R1 = pointer to filename

On exit

R0 = flag word:
- bit 0 set ⇒ file is held in memory (can be accessed directly)
- bits 1-31 reserved (ignore them)

R2 = size of buffer required to hold file

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call gives information about a message file, telling you if it is held in memory, and the size of buffer that is required to hold the file. If the file is held in memory, and you require read-only access, you need not use a buffer to access it.

Related SWIs

MessageTrans_OpenFile (SWI &41501)

Related vectors

None

MessageTrans_OpenFile (SWI &41501)

Opens a message file

On entry

R0 = pointer to file descriptor, held in the RMA if R2=0 on entry
R1 = pointer to filename, held in the RMA if R2=0 on entry
R2 = pointer to buffer to hold file data
0 ⇒ allocate some space in the RMA, or use the file directly if possible

On exit

—

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call opens a message file for subsequent use by the MessageTrans module.

The error 'Message file already open' is generated if R0 points to a structure already known to MessageTrans (ie already open).

An application may decide that it would like to buffer the file in its own workspace (rather than the RMA) if it needs to be loaded, or use the file directly if it is already in memory. To do this:

```
SYS 'MessageTrans_FileInfo',,filename$ TO flags%,,size%
IF flags% AND 1 THEN buffer%=0 ELSE buffer%=FNalloc(size%)
SYS 'OS_Module',6,,,17+LENfilename$ TO ,,filedesc%
$(filedesc%+16)=filename$
SYS 'MessageTrans_OpenFile',filedesc%,filedesc%+16,buffer%
```

where FNalloc() allocates a buffer of a given size, by using the Wimp_SlotSize or 'END=' command. Note that in fact the filename and file descriptor only need to be in the RMA if R2=0 on entry to MessageTrans_OpenFile.

Furthermore, if R2=0 on entry to this SWI, and the application uses direct pointers into the file (rather than copying the messages out) or uses MessageTrans_MakeMenus, it should also trap Service_MessageFileClosed, in case the file is unloaded.

Related SWIs

MessageTrans_FileInfo (SWI &41500), MessageTrans_CloseFile (SWI &41504)

Related vectors

None

MessageTrans_Lookup (SWI &41502)

Translates a message token into a string

On entry

R0 = pointer to file descriptor passed to MessageTrans_OpenFile
R1 = pointer to token, terminated by character 0, 10 or 13
R2 = pointer to buffer to hold result (0 ⇒ don't copy it)
R3 = size of buffer (if R2 non-zero)
R4 = pointer to parameter 0 (0 ⇒ don't substitute for '%0')
R5 = pointer to parameter 1 (0 ⇒ don't substitute for '%1')
R6 = pointer to parameter 2 (0 ⇒ don't substitute for '%2')
R7 = pointer to parameter 3 (0 ⇒ don't substitute for '%3')

On exit

R0, R1 preserved
R2 = pointer to result string (read-only with no substitution if R2=0 on entry)
R3 = size of result before terminator

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call translates a message token into a string, with optional parameter substitution. If the token is not found in the given message file, it is then looked up in the standard RISC OS messages file; see the section entitled *Standard RISC OS messages* on page 5-234.

Your application must have previously called MessageTrans_OpenFile, although you can still call this SWI if the file has been automatically closed by the system, because the system will also automatically re-open the file.

See the section entitled *Message file format* on page 5-234 for further details of the file format used to hold message tokens and their corresponding strings.

Related SWIs

MessageTrans_ErrorLookup (SWI &41506),
MessageTrans_GSLookup (SWI &41507)

Related vectors

None

MessageTrans_MakeMenus (SWI &41503)

Sets up a menu structure from a definition containing references to tokens

On entry

R0 = pointer to file descriptor passed to MessageTrans_OpenFile
R1 = pointer to menu definition (see below)
R2 = pointer to buffer to hold menu structure
R3 = size of buffer

On exit

R0 - R2 preserved
R3 = bytes remaining in buffer (0 if call was successful)

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call sets up a menu structure from a definition containing references to tokens, and also sets up appropriate widths for the menu and any submenus. Parameter substitution is not allowed.

The menu structure created can then be passed directly to Wimp_CreateMenu (see page 4-222).

Your application must have previously called MessageTrans_OpenFile, although you can still call this SWI if the file has been automatically closed by the system, because the system will also automatically re-open the file.

A 'Buffer overflow' error is generated if the buffer provided for the menu structure is too small.

The menu definition consists of one or more submenu definitions, terminated by a null byte. Each submenu definition consists of a title definition followed by one or more menu item definitions. A title definition has the following structure:

Bytes	Meaning
n	Token for menu title, terminated by character 0, 10 or 13
1	menu title foreground and frame colour
1	menu title background colour
1	menu work area foreground colour
1	menu work area background colour
1	height of following menu items
1	vertical gap between items

and a menu item definition has this structure:

Bytes	Meaning
m	token for menu item, terminated by character 0, 10 or 13
	word-align to here (*addr* := (*addr*+3) AND (NOT 3))
4	menu flags (bit 7 set ⇒ last item)
4	offset from RAM menu start to RAM submenu start (0 ⇒ no submenu)
4	icon flags

If the icon flags have bit 8 clear (ie they are not indirected), the message text for the icon will be read into the 12-byte block that forms the icon data; otherwise the icon data will be set up to point to the message text inside the file data. In the latter case they are read-only.

If the menu item flags bit 2 is set (writable) and the icon is indirected, the 3 words of the icondata in the RAM buffer are assumed to have already been set up by the calling program. The result of looking up the message token is copied into the buffer indicated by the first word of the icon data (truncated if it gets bigger than the buffer size indicated in [icondata,#8]).

See the section entitled *Message file format* on page 5-234 for further details of the file format used to hold message tokens and their corresponding strings.

For a more complete definition of the flags etc used in the menu definition, see the definition of Wimp_CreateMenu on page 4-222.

Related SWIs

None

Related vectors

None

MessageTrans_CloseFile (SWI &41504)

Closes a message file

On entry

R0 = pointer to file descriptor passed to MessageTrans_OpenFile

On exit

—

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call closes a message file.

Related SWIs

MessageTrans_OpenFile (SWI &41501)

Related vectors

None

MessageTrans_EnumerateTokens (SWI &41505)

Enumerates tokens that match a wildcarded token

On entry

R0 = pointer to file descriptor passed to MessageTrans_OpenFile
R1 = pointer to (wildcarded) token, terminated by character 0, 10, 13 or ':'
R2 = pointer to buffer to hold result
R3 = size of buffer
R4 = index (zero for first call)

On exit

R0, R1 preserved
R2 preserved, or zero if no further matching tokens found
R3 = length of result excluding terminator (if R2 ≠ 0)
R4 = index for next call (non-zero)

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call successively enumerates tokens that match a wildcarded token. Each successive call places a token in the buffer pointed to by R2, with the same terminator as that used for the wildcarded token that it matches. To enumerate all matching tokens, you should set R4 to zero, and repeatedly call this SWI until R2 is zero on exit.

Valid wildcards in the supplied token are:

Wildcard	Meaning
?	match 1 character
*	match 0 or more characters

See the section entitled *Message file format* on page 5-234 for further details of the file format used to hold message tokens and their corresponding strings.

Related SWIs

None

Related vectors

None

MessageTrans_ErrorLookup (SWI &41506)

Translates a message token within an error block

On entry

R0 = pointer to error block (word aligned)
R1 = pointer to file descriptor passed to MessageTrans_OpenFile
R2 = pointer to buffer to hold result (0 ⇒ use internal buffer)
R3 = buffer size (if R2 non-zero)
R4 = pointer to parameter 0 (0 ⇒ don't substitute for '%0')
R5 = pointer to parameter 1 (0 ⇒ don't substitute for '%1')
R6 = pointer to parameter 2 (0 ⇒ don't substitute for '%2')
R7 = pointer to parameter 3 (0 ⇒ don't substitute for '%3')

On exit

R0 = pointer to error buffer used
V flag set

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call translates a message token within an error block, with optional parameter substitution. If the token is not found in the given message file, it is then looked up in the standard RISC OS messages file; see the section entitled *Standard RISC OS messages* on page 5-234.

On entry the error block must contain:

Offset	Contents
0	error number
4	null terminated token

On exit the token is translated to the corresponding string.

If R2 is 0 on entry, MessageTrans will use one of its internal buffers for the result. There are 3 buffers for foreground processes and 3 for calls made from within IRQ processes. MessageTrans will cycle between these buffers.

Your application must have previously called MessageTrans_OpenFile, although you can still call this SWI if the file has been automatically closed by the system, because the system will also automatically re-open the file.

See the section entitled *Message file format* on page 5-234 for further details of the file format used to hold message tokens and their corresponding strings.

Related SWIs

MessageTrans_Lookup (SWI &41502), MessageTrans_GSLookup (SWI &41507)

Related vectors

None

MessageTrans_GSLookup (SWI &41507)

Translates a message token into a string, GSTrans'ing it

On entry

R0 = pointer to file descriptor passed to MessageTrans_OpenFile
R1 = pointer to token, terminated by character 0, 10 or 13
R2 = pointer to buffer to hold result (0 ⇒ don't copy it)
R3 = size of buffer (if R2 non-zero)
R4 = pointer to parameter 0 (0 ⇒ don't substitute for '%0')
R5 = pointer to parameter 1 (0 ⇒ don't substitute for '%1')
R6 = pointer to parameter 2 (0 ⇒ don't substitute for '%2')
R7 = pointer to parameter 3 (0 ⇒ don't substitute for '%3')

On exit

R0, R1 preserved
R2 = pointer to result string (read-only with no substitution if R2=0 on entry)
R3 = size of result before terminator

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call translates a message token into a string, with optional parameter substitution. If the token is not found in the given message file, it is then looked up in the standard RISC OS messages file; see the section entitled *Standard RISC OS messages* on page 5-234. The string is GSTrans'd after parameter substitution; this needs an intermediate buffer, and so will fail if one cannot be allocated from the RMA.

Your application must have previously called MessageTrans_OpenFile, although you can still call this SWI if the file has been automatically closed by the system, because the system will also automatically re-open the file.

See the section entitled *Message file format* on page 5-234 for further details of the file format used to hold message tokens and their corresponding strings.

Calling this SWI with R2=0 is exactly equivalent to calling MessageTrans_Lookup with R2=0

Related SWIs

OS_GSTrans (SWI &27), MessageTrans_Lookup (SWI &41502), MessageTrans_ErrorLookup (SWI &41506)

Related vectors

None

59 International module

Introduction

The International module allows the user to tailor the machine for use in different countries by setting:

- the keyboard – the mapping of keys to character codes
- the alphabet – the mapping from character codes to characters
- the country – both of the above mappings.

This module, in conjunction with the RISC OS kernel, controls the selection of these mappings, but it allows the actual mappings to be implemented in other modules, via the service mechanism. Thus, you could write your own international handlers.

Each country is represented by a name and number. The keyboard shares this list, and is normally on the same setting. However, there are cases for the country and the keyboard to be different. For example, the Greek keyboard would not allow you to type * Commands, because only Greek characters could be entered. In this case, the country could remain Greek, while the keyboard setting is changed temporarily for * Commands.

Each alphabet is also represented by a name and number. A country can only have one alphabet associated with it, but an alphabet can be used by many countries. For example, the Latin1 alphabet contains a general enough set of characters to be used by most Western European countries.

Overview and Technical Details

Names and numbers

Country numbers range from 0 to 99, and alphabet numbers are from 100 to 126. Here are lists of the currently available countries and alphabets.

Countries and keyboards

Here is a list of the currently-defined country and keyboard codes (provided by the international module), and the alphabets they use:

Code	Country and Keyboard	Alphabet
0	Default	Selects the configured country. If the configured country is 'Default', the keyboard ID byte is read from the keyboard
1	UK	Latin1
2	Master	BFont
3	Compact	BFont
4	Italy	Latin1
5	Spain	Latin1
6	France	Latin1
7	Germany	Latin1
8	Portugal	Latin1
9	Esperanto	Latin3
10	Greece	Greek
11	Sweden	Latin1
12	Finland	Latin1
13	(not used)	
14	Denmark	Latin1
15	Norway	Latin1
16	Iceland	Latin1
17	Canada1	Latin1
18	Canada2	Latin1
19	Canada	Latin1
20	Turkey	Latin3
21	Arabic	Special – ISO 8859/6
22	Ireland	Latin1
23	Hong Kong	*Not defined at time of going to press*
80	ISO1	Latin1
81	ISO2	Latin2
82	ISO3	Latin3
83	ISO4	Latin4

Alphabets

Here is a list of the alphabet codes currently defined, provided by the international module:

Code	Alphabet
100	BFont
101	Latin1
102	Latin2
103	Latin3
104	Latin4
107	Greek

Alphabet

OS_Byte 71 (SWI &06) reads or sets the alphabet by number. *Alphabet can also set the alphabet by name. *Alphabets lists all the available alphabets on the system. Remember that you should normally only need to change the country setting as this will also change the alphabet.

Use OS_ServiceCall &43,1 (SWI &30) to convert between alphabet name and number forms and OS_ServiceCall &43,3 to convert from alphabet number to name forms.

OS_ServiceCall &43,5 causes a module which recognises the alphabet number to define the characters in an alphabet in the range specified, by issuing VDU 23 commands itself. The call is issued by the OS when OS_Byte 71 is called to set the alphabet and also by OS_Byte 20 and 25.

Keyboard

OS_Byte 71 can also be used to read or set the keyboard number. *Keyboard can set it as well. Remember that you should normally only need to change the country setting as this will also change the keyboard.

When the keyboard setting is changed, by either of the above ways, an OS_ServiceCall &43,6 will be generated automatically. This is a broadcast to all keyboard handler modules that the keyboard selection has changed.

Country

Setting the country will set values for the alphabet and the keyboard. You should not usually have to override these settings. The country number can be read or set with OS_Byte 70. OS_Byte 240 can also read it. *Country can set the country by name. *Countries will list all the available country names. *Configure Country will set the default country by name and store it in CMOS RAM.

Use OS_ServiceCall &43,0 to convert between country name and number forms and OS_ServiceCall &43,2 to convert from country number to name forms.

To get the default alphabet for a country, OS_ServiceCall &43,4 can be called. Remember that the default keyboard number is the same as the country number.

Service calls

RISC OS provides service calls for the use of any module that adds to the set of international character sets and countries.

Service Calls

Service_International (Service Call &43)

International service

On entry

R1 = &43 (reason code)
R2 = sub-reason code
R3 - R5 depend on R2

On exit

R1 = 0 to claim, else preserved to pass on
R4, R5 depend on R2 on entry

Use

This call should be supported by any modules which add to the set of international character sets and countries. It is used by the international system module * Command interface, and may be called by applications too. See the chapter entitled *International module* on page 5-253 for details.

R2 contains a sub reason code which indicates which service is required:

R2	Service required
0	Convert country name to country number
1	Convert alphabet name to alphabet number
2	Convert country number to country name
3	Convert alphabet number to alphabet name
4	Convert country number to alphabet number
5	Define range of characters
6	Informative: New keyboard selected for use by keyboard handlers

Sub-reason codes

The following pages give details of each of these sub-reason codes. Most users will not need to issue these service calls directly, but the OS_Byte calls and * Commands use these. The information is provided mainly for writers of modules which provide additional alphabets etc.

Service_International 0 (Service Call &43)

Convert country name to country number

On entry

R1 = &43 (reason code)
R2 = 0 (sub-reason code)
R3 = pointer to country name terminated by a null

On exit

R1 = 0 if claimed, otherwise preserved
R2, R3 preserved
R4 = country number if recognised, preserved if not recognised

Use

Any module providing additional countries should compare the given country name with each country name provided by the module, ignoring case differences between letters and allowing for abbreviations using '.'. If the given country name matches a known country name, then it should claim the service (by setting R1 to 0), and set R4 to the corresponding country number.

If the given country name is not recognised, all registers should be preserved.

Service_International 1 (Service Call &43)

Convert alphabet name to alphabet number

On entry

R1 = &43 (reason code)
R2 = 1 (sub-reason code)
R3 = pointer to alphabet name terminated by a null

On exit

R1 = 0 if claimed, otherwise preserved
R2, R3 preserved
R4 = alphabet number if recognised, preserved if not recognised

Use

Any module providing additional alphabets should compare the given alphabet name with each alphabet name provided by the module, ignoring case differences between letters and allowing for abbreviations using '.'. If the given alphabet name matches a known alphabet name, then it should claim the service (by setting R1 to 0), and set R4 to the corresponding alphabet number.

If the given alphabet name is not recognised, all registers should be preserved.

Service_International 2 (Service Call &43)

Convert country number to country name

On entry

R1 = &43 (reason code)
R2 = 2 (sub-reason code)
R3 = country number
R4 = pointer to buffer for name
R5 = length of buffer in bytes

On exit

R1 = 0 if claimed, otherwise preserved
R2 - R4 preserved
R5 = number of characters put into buffer if recognised, otherwise preserved

Use

Any module providing additional countries should compare the given country number with each country number provided by the module. If the given country number matches a known country number, then it should claim the service (by setting R1 to 0), put the country name in the buffer, and set R5 to the number of characters put in the buffer.

If the given country number is not recognised, all registers should be preserved.

Service_International 3 (Service Call &43)

Convert alphabet number to alphabet name

On entry

R1 = &43 (reason code)
R2 = 3 (sub-reason code)
R3 = alphabet number
R4 = pointer to buffer for name
R5 = length of buffer in bytes

On exit

R1 = 0 if claimed, otherwise preserved
R2 - R4 preserved
R5 = number of characters put into buffer if recognised, otherwise preserved

Use

Any module providing additional alphabets should compare the given alphabet number with each alphabet number provided by the module. If the given alphabet number matches a known alphabet number, then it should claim the service (by setting R1 to 0), put the alphabet name in the buffer, and set R5 to the number of characters put in the buffer.

If the given alphabet number is not recognised, all registers should be preserved.

Service_International 4 (Service Call &43)

Convert country number to alphabet number

On entry

R1 = &43 (reason code)
R2 = 4 (sub-reason code)
R3 = country number

On exit

R1 = 0 if claimed, otherwise preserved
R2, R3 preserved
R4 = alphabet number if recognised, otherwise preserved

Use

Any module providing additional countries should compare the given country number with each country number provided by the module. If the given country number matches a known country number, then it should claim the service (by setting R1 to 0), and set R4 to the corresponding alphabet number.

If the given country number is not recognised, all registers should be preserved.

Service_International 5 (Service Call &43)

Define a range of characters from a given alphabet number

On entry

R1 = &43 (reason code)
R2 = 5 (sub-reason code)
R3 = alphabet number
R4 = ASCII code of first character in range
R5 = ASCII code of last character in range

On exit

R1 = 0 if claimed, otherwise preserved
R2 - R5 preserved

Use

Any module providing additional alphabets should compare the given alphabet number with each alphabet number provided by the module. If the given alphabet number matches a known alphabet number, then that service should be claimed (by setting R1 to 0) and all the characters should be defined in the range R4 to R5 inclusive, using calls to VDU 23. Any characters not defined in the specified alphabet are missed out: for example, characters &80–&9F in Latin1.

If the given alphabet number is not recognised, all registers should be preserved.

Service_International 6 (Service Call &43)

Notification of a new keyboard selection

On entry

R1 = &43 (reason code)
R2 = 6 (sub-reason code)
R3 = new keyboard number
R4 = alphabet number associated with this keyboard

On exit

R1 preserved (call must not be claimed)
R2 - R4 preserved

Use

This service call is for internal use by keyboard handlers. It is sent automatically after the keyboard selection is changed. You must not claim it.

SWI Calls

OS_Byte 70 (SWI &06)

Read/write country number

On entry

R0 = 70 (&46) (reason code)
R1 = 127 to read or country number to write

On exit

R0 is preserved
R1 = country number read or before being overwritten, or 0 if invalid country number passed
R2 is corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns or sets the country number used by the International module.

Related SWIs

OS_Byte 240 (SWI &06)

Related vectors

ByteV

OS_Byte 71 (SWI &06)

Read/write alphabet or keyboard

On entry

R0 = 71 (&47) (reason code)
R1 = 0–126 for setting the alphabet number
127 for reading the current alphabet number
128–254 for setting the keyboard number (R1–128)
255 for reading the current keyboard number

On exit

R0 is preserved
R1 = alphabet or keyboard number read or before being overwritten, or 0 if invalid value passed
R2 is corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns or sets the alphabet or keyboard number used by the international module. Their settings can be read without altering them, or you can set a new value for either. This SWI will return a zero if the value passed to set the new value is not one of the known alphabets or keyboards.

Note that the keyboard setting is offset by 128; eg to set keyboard 3, you must pass 131 in R1.

Related SWIs

OS_Byte 70 (SWI &06)

Related vectors

ByteV

OS_Byte 240 (SWI &06)

Read country number

On entry

R0 = 240 (&F0) (reason code)
R1 = 0
R2 = 255

On exit

R0 is preserved
R1 = country number
R2 = user flag (see OS_Byte 241)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the country number used by the international module.

Related SWIs

OS_Byte 70 (SWI &06)

Related vectors

ByteV

*Commands

*Alphabet

Selects an alphabet

Syntax

`*Alphabet [country_name|alphabet_name]`

Parameters

country_name	Valid countries are currently Canada, Canada1, Canada2, Compact, Default, Denmark, Esperanto, Finland, France, Germany, Greece, Iceland, ISO1, Italy, Master, Norway, Portugal, Spain, Sweden, and UK. A list of parameters can be obtained with the *Countries command.
alphabet_name	Valid alphabets are currently BFont, Latin1, Latin2, Latin3, Latin4 and Greek. A list of parameters can be obtained with the *Alphabets command.

Use

*Alphabet selects an alphabet, setting the alphabetical set of characters according to the country name or alphabet name.

If a country name of Default is given, then the keyboard ID byte (read from the keyboard) is used as the country number, providing it is in the range 1 – 31. Current UK keyboards return keyboard ID 1, which corresponds to country UK.

With no parameter, this command displays the currently selected alphabet.

Example

`*Alphabet Latin3`

Related commands

*Alphabets

Related SWIs

OS_Byte 71 (SWI &06)

Related vectors

None

*Alphabets

Lists all the alphabets currently supported

Syntax

```
*Alphabets
```

Parameters

None

Use

*Alphabets lists all the alphabets currently supported by your Acorn computer.

Use the *Alphabet command to change the alphabetical set of characters you are using.

Example

```
*Alphabets
Alphabets:
BFont   Latin1  Latin2  Latin3  Latin4  Greek
```

Related commands

*Alphabet

Related SWIs

OS_Byte 71 (SWI &06)

Related vectors

None

*Configure Country

Sets the configured alphabet and keyboard layout

Syntax

`*Configure Country country_name`

Parameters

country_name	Valid countries are currently Canada, Canada1, Canada2, Compact, Default, Denmark, Esperanto, Finland, France, Germany, Greece, Iceland, ISO1, Italy, Master, Norway, Portugal, Spain, Sweden, and UK. A list of parameters can be obtained with the *Countries command.

Use

*Configure Country sets the configured alphabet and keyboard layout to the appropriate ones for the given country. For some countries you will also need to load a relocatable module that defines the keyboard layout.

If the configured country is Default, then the keyboard ID byte (read from the keyboard) is used as the country number, providing it is in the range 1 – 31. Current UK keyboards return keyboard ID 1, which corresponds to country UK.

Example

`*Configure Country Italy`

Related commands

*Country, *Countries

Related SWIs

OS_Bytes 70 and 240 (SWI &06)

Related vectors

None

*Countries

Lists all the countries currently supported

Syntax

`*Countries`

Parameters

None

Use

*Countries lists all the countries currently supported by modules in the system.

Example

```
*Countries
Countries:
Default UK       Master  Compact Italy   Spain   France
Germany Portugal         Esperanto       Greece  Sweden
Norway  Iceland Canada1 Canada2 Canada   ISO1
```

Related commands

*Configure Country, *Country, *Alphabet, *Alphabets, *Keyboard

Related SWIs

OS_Bytes 70 and 240 (SWI &06)

Related vectors

None

*Country

Selects the appropriate alphabet and keyboard layout for a given country

Syntax

`*Country [country_name]`

Parameters

`country_name`	Valid countries are currently Canada, Canada1, Canada2, Compact, Default, Denmark, Esperanto, Finland, France, Germany, Greece, Iceland, ISO1, Italy, Master, Norway, Portugal, Spain, Sweden, and UK. A list of parameters can be obtained with the *Countries command.

Use

*Country selects the appropriate alphabet and keyboard layout for a given country. For some countries you will also need to load a relocatable module that defines the keyboard layout. If you prefer, you can use *Alphabet and *Keyboard to set independently the alphabet and keyboard layout, leaving the country setting unchanged.

If the given country is Default, then the keyboard ID byte (read from the keyboard) is used as the country number, providing it is in the range 1 – 31. Current UK keyboards return keyboard ID 1, which corresponds to country UK.

With no parameter, this command displays the currently selected country.

Example

`*Country Italy`

Related commands

*Configure Country, *Countries, *Alphabet, *Alphabets, *Keyboard

Related SWIs

OS_Bytes 70 and 240 (SWI &06)

Related vectors

None

*Keyboard

Selects the keyboard layout for a given country

Syntax

`*Keyboard [country_name]`

Parameters

`country_name`	Valid countries are currently Canada, Canada1, Canada2, Compact, Default, Denmark, Esperanto, Finland, France, Germany, Greece, Iceland, ISO1, Italy, Master, Norway, Portugal, Spain, Sweden, and UK. A list of parameters can be obtained with the *Countries command.

Use

*Keyboard selects the appropriate keyboard layout for a given country. For some countries you will also need to load a relocatable module that defines the keyboard layout.

If the given country is Default, then the keyboard ID byte (read from the keyboard) is used as the country number, providing it is in the range 1 – 31. Current UK keyboards return keyboard ID 1, which corresponds to country UK.

With no parameter, this command displays the currently selected keyboard layout.

Example

`*Keyboard Denmark`

Related commands

*Country

Related SWIs

OS_Byte 71 (SWI &06)

Related vectors

None

60 The Territory Manager

Introduction

The *territory manager* provides SWIs and * Commands for applications to access *territory modules*. Each territory module provides the services and information necessary for both RISC OS 3 and its applications to be easily adapted for use in different *territories* (ie regions of the world).

Purpose of the territory manager

There are three main purposes in providing the territory manager:

1. To enable Acorn to produce a version of RISC OS 3 targeted at a foreign market. This requires not only the ability to translate all system text to a foreign language, but also the ability to support different time zones, different alphabets and different keyboard layouts.
2. To help you write application software so you can easily adapt it for a foreign market.
3. To enable you to write application software that can cope with using more than one language at the same time.

RISC OS 3 addresses all of these aspects.

Use of the territory manager

The territory manager provides a wide range of services and information to help you.

Use the territory manager wherever possible. Don't make assumptions about any of the features it supports and can provide information on.

If you do use the territory manager, you will find it much easier to modify your programs for supply to international markets, and have a much wider potential user base.

Overview

The territory manager

The territory manager is a new RISC OS 3 module providing control over the localised aspects of the computer. RISC OS itself only uses one territory for all its functions, but the territory manager can have more than one territory module loaded at any one time, and applications can use these additional territories.

Territory modules

A territory module (such as the UK Territory module present in the RISC OS 3 ROM) is a module providing the territory manager with services and information for a specific territory (such as the UK), amongst which are:

- a keyboard handler for the territory's keyboard layout
- the correct alphabet for the territory
- information on the use of that alphabet, including the direction of writing to use, the properties of each character, and variant forms of each charter (such as upper/lower case, control characters, and unaccented characters)
- a sort order for strings using the territory's alphabet
- the characters that are used for numbers, and how those numbers are formatted, both as numeric and monetary quantities
- the time zones and the formats of time and date used in the territory; together with facilities for reading and setting the local time using these formats
- information on the calendar used in the territory.

Obviously this is only a summary of what is provided; for full information you should see the section entitled *Territory manager* SWIs on page 5-284 and the section entitled *Territory module* SWIs on page 5-295, especially the latter.

Technical details

Loading and setting the current territory

Each computer running RISC OS has a configured value for the current territory, set using *Configure Territory (see page 5-332), and stored in its CMOS RAM. On a reset or a power-on, RISC OS will try to load this territory as follows:

1 It will load any territory modules in ROM. (Typically there is only one, for the territory into which the computer has been sold.) If one of these is the configured territory, no further action is taken.

2 Otherwise, it will look on the *configured device* (ie the configured filesystem and drive) for the file $.!Territory.Territory.

   - If the configured filesystem is Econet, it will instead look for the file &.!Territory.Territory

3 If it finds that file, it will load it, and also any files in the directory ...!Territory.Territory.Messages.

4 If it doesn't find that file, it will use a pictorial request to ask the user to insert a floppy disc containing the territory. It will keep doing so until it finds the file adfs::0.$.!Territory.Territory, which it loads along with any files in the directory adfs::0.$.!Territory.Territory.Modules.

At the end of this process:

- If the configured territory is in ROM, only those territory modules in ROM will be loaded
- If the configured territory is not in ROM, both those territory modules in ROM and another territory module (hopefully the configured one) will be loaded.

RISC OS then selects as the current territory either the configured territory, or – if it is not present – a default territory from ROM.

The current territory

The *current territory* is used by RISC OS for all operating system functions that may change from territory to territory. This includes such things as the language used to display menus, and the default time offset from UTC. As we saw above, the current territory will normally be the configured territory; but if that can't be found, a default ROM territory is used instead.

There can only be one current territory for any one computer. This is because the current territory controls such things as the language used for menus. It would be very confusing to have, for example, some of the menus appear in one language and some in another language. In the UK, even if you are editing a German document, you would normally still want the menus to appear in English.

Once the current territory has been set, you can't change it in mid-session. To change the current territory, you should change the configured territory, and ensure that the new current territory you wish to use is available (either in ROM, or in $.!Territory on the default device). You then need to reboot your computer.

Multiple territories

Whilst RISC OS itself only makes use of the computer's one current territory, the territory manager can have more than one territory module loaded. Applications can then make use of these extra territory modules. For example, you may wish to provide an application that can include text in two different languages in the same document. It is useful for such an application to be able to read the information relating to both languages at the same time.

Initialising territory modules

When the territory manager starts, it issues a service call (Service_TerritoryManagerLoaded) to announce its presence to territory modules. Whenever a territory module receives this service call, it must issue the SWI Territory_Register to add itself to the territory manager's list of active territories. A territory module must also issue this SWI whenever its initialisation entry point is called, thus ensuring that if it is initialised after the territory manager, it still registers itself.

Territory_Register

This SWI also registers with the territory manager the entry points to the routines that the territory module uses to provide its information and services. These entry points are called by issuing SWIs to the territory manager, which specify the territory module that is to be used to service the SWI. The territory manager then calls the appropriate entry point in the specified territory module.

Setting up for the current territory

Once the territory manager has started, and any loaded territory modules have registered themselves, it then sets up the current territory. To do so, it:

- calls Territory_SelectKeyboardHandler to select the keyboard handler

- calls Territory_Alphabet to find the alphabet number that should be used in the territory
- issues Service_International 5 to define that alphabet.

Scope of a territory

A territory need not correspond to a country. Rather, a territory is a region for which a single territory module correctly provides all the services and information. As soon as one or more of the services or information differ, you should provide a different territory (but see below). Sometimes you may need to provide more than one territory for a single country. For example, to properly support the whole of Switzerland you would need a separate territory for each of the languages used.

Supporting minor differences

Sometimes it might appear that a region needs to be split into several territories because of a single minor difference. For example, to support the whole of the USA you would need five territories identical in every respect except for their support of time zones. In such cases you may consider supplying a single generic territory with an extra configuration option; in this case, it would set which time zone to use.

Remember that if you wish to store this configuration option in CMOS RAM, you must apply for an allocation from Acorn. See the section entitled *CMOS RAM bytes* on page 6-475.

Service Calls

Service_TerritoryManagerLoaded (Service Call &64)

Tell territory modules to register themselves.

On entry

R1 = &64 (reason code)

On exit

All registers preserved

Use

For more information see the TerritoryManager chapter.

Service_TerritoryStarted (Service Call &75)

New Territory starting

On entry

R0 = &75 (reason code)

On exit

This service call should not be claimed.

All registers preserved

Use

This is issued by the territory manager when a new territory has been selected as the machine territory.

This can only happen on machine startup, and is used by the ROM modules to re-open their messages files. RAM resident modules do not need to take notice of this service call.

Territory manager SWIs

Territory_Number (SWI &43040)

Returns the territory number of the current territory

On entry

—

On exit

R0 = current territory's number

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the territory number of the current territory (see the section entitled *Loading and setting the current territory* on page 5-279, and **Configure Territory* on page 5-332).

Related SWIs

None

Related vectors

None

Territory_Register (SWI &43041)

Adds the given territory to the list of active territories

On entry

R0 = territory number
R1 = pointer to buffer containing list of entry points for SWIs
R2 = value of R12 on entry to territory

On exit

R0 - R2 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call adds the given territory to the list of active territories, making it available for application programs. A territory module must issue this call from its initialisation entry point when it is initialised, and whenever it receives the service call Service_TerritoryManagerLoaded.

The list of entry points is in the same order as the SWIs detailed below in the section entitled *Territory module SWIs* on page 5-295.

Related SWIs

Territory_Deregister

Related vectors

None

Territory_Deregister (SWI &43042)

Removes the given territory from the list of active territories

On entry

R0 = territory number

On exit

R0 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call removes the given territory from the list of active territories. A territory module must issue this call from its finalisation entry point when it is killed.

Related SWIs

None

Related vectors

None

Territory_NumberToName (SWI &43043)

Returns the name of the given territory

On entry

R0 = territory number
R1 = pointer to buffer to contain name of territory in current territory
R2 = length of buffer

On exit

R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the name of the given territory in the current territory's language and alphabet.

Related SWIs

None

Related vectors

None

Territory_Exists (SWI &43044)

Checks if the given territory is currently present in the machine

On entry

R0 = territory number

On exit

R0 preserved
Z flag set if territory is currently loaded

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call checks if the given territory is currently present in the machine, and can be used by applications.

Related SWIs

None

Related vectors

None

Territory_AlphabetNumberToName (SWI &43045)

Returns the name of the given alphabet

On entry

R0 = alphabet number
R1 = pointer to buffer to hold name of alphabet in current territory
R2 = length of buffer

On exit

R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the name of the given alphabet in the current territory's language and alphabet.

Related SWIs

None

Related vectors

None

Territory_SelectAlphabet (SWI &43046)

Selects the correct alphabet for the given territory

On entry

R0 = territory number, or -1 to use current territory

On exit

R0 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call selects the correct alphabet for the given territory, and defines the system font appropriately.

Related SWIs

None

Related vectors

None

Territory_SetTime (SWI &43047)

Sets the clock to a given 5 byte UTC time

On entry

R0 = pointer to 5 byte UTC time

On exit

R0 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets the clock to a given 5 byte UTC time.

Related SWIs

None

Related vectors

None

Territory_ReadCurrentTimeZone (SWI &43048)

Returns information on the current time zone

On entry

—

On exit

R0 = pointer to name of current time zone
R1 = offset from UTC to current time zone, in centiseconds (signed 32-bit)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns information on the current time zone, giving its name in the current territory's language and alphabet, and its offset in centiseconds from UTC time.

Related SWIs

None

Related vectors

None

Territory_ConvertTimeToUTCOrdinals (SWI &43049)

Converts a 5 byte UTC time to UTC time ordinals

On entry

R1 = pointer to 5 byte UTC time
R2 = pointer to word aligned buffer to hold ordinals

On exit

R1, R2 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call converts a 5 byte UTC time to UTC time ordinals. The word-aligned buffer pointed to by R2 holds the following:

Offset	Value
0	centiseconds
4	seconds
8	minutes
12	hours (out of 24)
16	day number in month
20	month number in year
24	year number
28	day of week

Related SWIs

None

Related vectors

None

Territory module SWIs

The following SWIs are provided by individual territory modules. The territory manager calls these SWIs using the entry points that territory modules pass by calling Territory_Register when they start (or when the territory manager restarts).

For all of the following SWIs, on entry R0 is used to specify to the territory manager the number of the territory module which will handle the call. A value of −1 means that the **current** territory (see the section entitled *Loading and setting the current territory* on page 5-279, and **Configure Territory* on page 5-332) will handle the call.

Territory_ReadTimeZones (SWI &4304A)

Returns information on the time zones for the given territory

On entry

R0 = territory number, or –1 to use current territory

On exit

R0 = pointer to name of standard time zone for given territory
R1 = pointer to name of daylight saving (or summer) time for given territory
R2 = offset from UTC to standard time, in centiseconds (signed 32-bit)
R3 = offset from UTC to daylight saving time, in centiseconds (signed 32-bit)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns information on the time zones for the given territory, giving the names of the territory's standard time zone and daylight saving time, and their offsets from UTC time.

Related SWIs

None

Related vectors

None

Territory_ConvertDateAndTime (SWI &4304B)

Converts a 5 byte UTC time into a string, giving the date and time

On entry

R0 = territory number, or –1 to use current territory
R1 = pointer to 5 byte UTC time
R2 = pointer to buffer for resulting string
R3 = size of buffer
R4 = pointer to null terminated format string

On exit

R0 = pointer to buffer (R2 on entry)
R1 = pointer to terminating 0 in buffer
R2 = number of bytes free in buffer
R3 = pointer to format string (R4 on entry)
R4 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call converts a 5 byte UTC time into a string, giving the date and time in a territory specific format given by the supplied format string.

The format string is copied directly into the result buffer, except when a '%' character appears. In this case the next two characters are treated as a special field name which is replaced by a component of the current time.

For details of the format field names see the section entitled *Format field names* on page 1-393.

This call is equivalent to the SWI OS_ConvertDateAndTime. You should use it in preference to that call, which just calls this SWI. The resulting string for both calls is in local time for the given territory, and in the local language and alphabet.

Related SWIs

None

Related vectors

None

Territory_ConvertStandardDateAndTime (SWI &4304C)

Converts a 5 byte UTC time into a string, giving the time and date

On entry

R0 = territory number, or –1 to use current territory
R1 = pointer to 5 byte UTC time
R2 = pointer to buffer for resulting string
R3 = size of buffer

On exit

R0 = pointer to buffer (R2 on entry)
R1 = pointer to terminating 0 in buffer
R2 = number of bytes free in buffer
R3 preserved.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call converts a 5 byte UTC time into a string, giving the date and time in a standard territory specific format.

This call is equivalent to the SWI OS_ConvertStandardDateAndTime. You should use it in preference to that call, which just calls this SWI. The resulting string for both calls is in local time for the given territory, and in the local language and alphabet.

Related SWIs

None

Related vectors

None

Territory_ConvertStandardDate (SWI &4304D)

Converts a 5 byte UTC time into a string, giving the date only

On entry

R0 = territory number, or –1 to use current territory
R1 = pointer to 5 byte UTC time
R2 = pointer to buffer for resulting string
R3 = size of buffer

On exit

R0 = pointer to buffer (R2 on entry)
R1 = pointer to terminating 0 in buffer
R2 = number of bytes free in buffer
R3 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call converts a 5 byte UTC time into a string, giving the date only in a standard territory specific format. The resulting string is in local time for the given territory, and in the local language and alphabet.

Related SWIs

None

Related vectors

None

Territory_ConvertStandardTime (SWI &4304E)

Converts a 5 byte UTC time into a string, giving the time only

On entry

R0 = territory number, or −1 to use current territory
R1 = pointer to 5 byte UTC time
R2 = pointer to buffer for resulting string
R3 = size of buffer

On exit

R0 = pointer to buffer (R2 on entry)
R1 = pointer to terminating 0 in buffer
R2 = number of bytes free in buffer
R3 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call converts a 5 byte UTC time into a string, giving the time only in a standard territory specific format. The resulting string is in local time for the given territory, and in the local language and alphabet.

Related SWIs

None

Related vectors

None

Territory_ConvertTimeToOrdinals (SWI &4304F)

Converts a 5 byte UTC time to local time ordinals for the given territory

On entry

R0 = territory number, or −1 to use current territory
R1 = pointer to 5 byte UTC time
R2 = pointer to word aligned buffer to hold ordinals

On exit

R1, R2 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call converts a 5 byte UTC time to local time ordinals for the given territory. The word-aligned buffer pointed to by R2 holds the following:

Offset	Value
0	centi-seconds
4	seconds
8	minutes
12	hours (out of 24)
16	day number in month
20	month number in year
24	year number
28	day of week

Related SWIs

None

Related vectors

None

Territory_ConvertTimeStringToOrdinals (SWI &43050)

Converts a time string to time ordinals

On entry

R0 = territory number, or –1 to use current territory
R1 = reason code:
1 ⇒ format string is %24:%MI:%SE
2 ⇒ format string is %W3, %DY-%M3-%CE%YR
3 ⇒ format string is %W3, %DY-%M3-%CE%YR.%24:%MI:%SE
R2 = pointer to time string
R3 = pointer to word aligned buffer to contain ordinals

On exit

R1 - R3 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call converts a time string to time ordinals. The time string is expected to be in the local language and alphabet for the given territory – as obtained from Territory_ConvertDateAndTime – with the appropriate format string. The word-aligned buffer pointed to by R3 holds the following:

Offset	Value
0	centi-seconds
4	seconds
8	minutes
12	hours (out of 24)

16	day number in month
20	month number in year
24	year number

Values that are not present in the string are set to −1.

Related SWIs

None

Related vectors

None

Territory_ConvertOrdinalsToTime (SWI &43051)

Converts local time ordinals for the given territory to a 5 byte UTC time

On entry

R0 = territory number, or −1 to use current territory
R1 = pointer to block to hold 5 byte UTC time
R2 = pointer to block containing ordinals

On exit

R1, R2 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call converts local time ordinals for the given territory to a 5 byte UTC time. The word-aligned buffer pointed to by R2 holds the following:

Offset	Value
0	centi-seconds
4	seconds
8	minutes
12	hours (out of 24)
16	day number in month
20	month number in year
24	year number

Related SWIs

None

Related vectors

None

Territory_Alphabet (SWI &43052)

Returns the alphabet number that should be selected for the given territory

On entry

R0 = territory number, or –1 to use current territory

On exit

R0 = alphabet number used by the given territory (eg 101 = Latin1)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the alphabet number that will be selected if Territory_SelectAlphabet is issued for the given territory.

Related SWIs

None

Related vectors

None

Territory_AlphabetIdentifier (SWI &43053)

Returns an identifier string for the alphabet that should be used for the given territory

On entry

R0 = territory number, or –1 to use current territory

On exit

R0 = pointer to identifier string for the alphabet used by the given territory

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns an identifier string for the alphabet that will be selected if Territory_SelectAlphabet is issued for the given territory (eg 'Latin1' for the Latin 1 alphabet).

The identifier of each alphabet is guaranteed to be the same no matter which territory returns it, and to consist of ASCII characters only (ie 7 bit characters).

Related SWIs

None

Related vectors

None

Territory_SelectKeyboardHandler (SWI &43054)

Selects the keyboard handler for the given territory

On entry

R0 = territory number, or –1 to use current territory

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call selects the keyboard handler for the given territory.

Related SWIs

None

Related vectors

None

Territory_WriteDirection (SWI &43055)

Returns the direction of writing used in the given territory

On entry

R0 = territory number, or –1 to use current territory

On exit

R0 = bit field giving write direction

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the direction of writing used in the given territory, as a bit field in R0:

Bit	Value	Meaning
0	0	Writing goes from left to right
	1	Writing goes from right to left
1	0	Writing goes from top to bottom
	1	Writing goes from bottom to top
2	0	Lines of text are horizontal
	1	Lines of text are vertical

Bits 3 - 31 are reserved, and are returned as 0.

Related SWIs

None

Related vectors

None

Territory_CharacterPropertyTable (SWI &43056)

Returns a pointer to a character property table

On entry

R0 = territory number, or –1 to use current territory
R1 = code for required character property table pointer

On exit

R0 = pointer to character property table

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns a pointer to a character property table, which is a 256 bit table indicating whether or not each character in the given territory's alphabet has a particular property. If a bit is set, the corresponding character has that property. Current property tables are:

Code	Meaning when bit set
0	character is a control code
1	character is uppercase
2	character is lowercase
3	character is alphabetic character
4	character is a punctuation character
5	character is a space character
6	character is a digit
7	character is a hex digit
8	character has an accent
9	character flows in the same direction as the territory's write direction
10	character flows in the reverse direction from the territory's write direction

A character which doesn't have properties 9 and 10 is a natural character which flows in the same direction as the surrounding text. A character can't have both property 9 and property 10.

Related SWIs

None

Related vectors

None

Territory_LowerCaseTable (SWI &43057)

Returns a pointer to a lower case table

On entry

R0 = territory number, or –1 to use current territory

On exit

R0 = pointer to lower case table

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns a pointer to a lower case table, which is a 256 byte table giving the lower case version of each character in the given territory's alphabet. Characters that do not have a lower case version (eg numbers, punctuation) appear unchanged in the table.

Related SWIs

None

Related vectors

None

Territory_UpperCaseTable (SWI &43058)

Returns a pointer to an upper case table

On entry

R0 = territory number, or –1 to use current territory

On exit

R0 = pointer to upper case table

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns a pointer to an upper case table, which is a 256 byte table giving the upper case version of each character in the given territory's alphabet. Characters that do not have a lower case version (eg numbers, punctuation) appear unchanged in the table.

Related SWIs

None

Related vectors

None

Territory_ControlTable (SWI &43059)

Returns a pointer to a control character table

On entry

R0 = territory number, or –1 to use current territory

On exit

R0 = pointer to control character table

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns a pointer to a control character table, which is a 256 byte table giving the value of each character in the given territory's alphabet if it is typed while the Ctrl key is depressed. Characters that do not have a corresponding control character appear unchanged in the table.

Related SWIs

None

Related vectors

None

Territory_PlainTable (SWI &4305A)

Returns a pointer to an unaccented character table

On entry

R0 = territory number, or –1 to use current territory

On exit

R0 = pointer to unaccented character table

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns a pointer to an unaccented character table, which is a 256 byte table giving the unaccented version of each character in the given territory's alphabet. Characters that are normally unaccented appear unchanged in the table

Related SWIs

None

Related vectors

None

Territory_ValueTable (SWI &4305B)

Returns a pointer to a numeric value table

On entry

R0 = territory number, or –1 to use current territory

On exit

R0 = pointer to numeric value table

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns a pointer to a numeric value table, which is a 256 byte table giving the numeric value of each character in the given territory's alphabet when used as a digit. This includes non-decimal numbers: for example, in English '9' has the numeric value 9, and both 'A' and 'a' have the numeric value 10 (as in the hexadecimal number &9A). Characters that do not have a numeric value have the value 0 in the table

Related SWIs

None

Related vectors

None

Territory_RepresentationTable (SWI &4305C)

Returns a pointer to a numeric representation table

On entry

R0 = territory number, or –1 to use current territory

On exit

R0 = pointer to numeric representation table

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns a pointer to a numeric representation table, which is a 16 byte table giving the 16 characters in the given territory's alphabet which should be used to represent the values 0 - 15. This includes non-decimal numbers: for example, in English the value 9 is represented by '9', and the value 10 by 'A' (as in the hexadecimal number &9A).

Related SWIs

None

Related vectors

None

Territory_Collate (SWI &4305D)

Compares two strings in the given territory's alphabet

On entry

R0 = territory number, or –1 to use current territory
R1 = pointer to *string1* (null terminated)
R2 = pointer to *string2* (null terminated)
R3 = flags:
  bit 0: ignore case if set
  bit 1: ignore accents if set
  bits 2-31 are reserved (must be zero)

On exit

R0 < 0 if *string1* < *string2*
= 0 if *string1* = *string2*
> 0 if *string1* > *string2*
R1 - R3 preserved

N set and V clear if *string1* < *string2* (LT)
Z set if *string1* = *string2* (EQ).
C set and Z clear if *string1* > *string2* (HI)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call compares two strings in the given territory's alphabet. It sets the same flags in the Program Status Register (part of R15, the program counter) as the ARM's numeric comparison instructions do. You should **always** use this call to compare strings.

Related SWIs

None

Related vectors

None

Territory_ReadSymbols (SWI &4305E)

Returns various information telling you how to format numbers

On entry

R1 = reason code (see below)

On exit

R0 = requested value

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns various information telling you how to format numbers, in particular monetary quantities. Current reason codes are:

Code	Meaning
0	Return pointer to null terminated decimal point string.
1	Return pointer to null terminated thousands separator.
2	Return pointer to byte list containing the size of each group of digits in formatted non-monetary quantities (least significant first): 255 — no further grouping 0 — repeat last grouping for rest of number *other* — size of current group; the next byte contains the size of the next most significant group of digits
3	Return pointer to null terminated international currency symbol.
4	Return pointer to null terminated currency symbol in local alphabet.
5	Return pointer to null terminated decimal point used for monetary quantities.
6	Return pointer to null terminated thousands separator for monetary quantities.
7	Return pointer to byte list containing the size of each group of digits in formatted monetary quantities (least significant first): 255 — no further grouping 0 — repeat last grouping for rest of number *other* — size of current group; the next byte contains the size of the next most significant group of digits
8	Return pointer to null terminated positive sign used for monetary quantities.
9	Return pointer to null terminated negative sign used for monetary quantities.
10	Return number of fractional digits to be displayed in a formatted international monetary quantity (ie one using the international currency symbol).
11	Return number of fractional digits to be displayed in a formatted monetary quantity.
12	Return for a non-negative formatted monetary quantity: 1 — If the currency symbol precedes the value. 0 — If the currency symbol succeeds the value.
13	Return for a non-negative formatted monetary quantity: 1 — If the currency symbol is separated by a space from the value. 0 — If the currency symbol is not separated by a space from the value.
14	Return for a negative formatted monetary quantity: 1 — If the currency symbol precedes the value. 0 — If the currency symbol succeeds the value.
15	Return for a negative formatted monetary quantity: 1 — If the currency symbol is separated by a space from the value. 0 — If the currency symbol is not separated by a space from the value.
16	Return for a non-negative formatted monetary quantity: 0 — If there are parentheses around the quantity and currency symbol.

	1	If the sign string precedes the quantity and currency symbol.
	2	If the sign string succeeds the quantity and currency symbol.
	3	If the sign string immediately precedes the currency symbol.
	4	If the sign string immediately succeeds the currency symbol.
17	Return for a negative formatted monetary quantity:	
	0	If there are parentheses around the quantity and currency symbol.
	1	If the sign string precedes the quantity and currency symbol.
	2	If the sign string succeeds the quantity and currency symbol.
	3	If the sign string immediately precedes the currency symbol.
	4	If the sign string immediately succeeds the currency symbol.
18	Return pointer to null terminated list separator.	

Related SWIs

None

Related vectors

None

Territory_ReadCalendarInformation (SWI &4305F)

Returns various information about the given territory's calendar

On entry

R0 = territory number, or –1 to use current territory
R1 = pointer to 5 byte UTC time
R2 = pointer to 12 word buffer

On exit

R0 - R2 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call takes the 5 byte UTC time passed to it, and returns various information about the given territory's calendar in the buffer pointed to by R2:

Offset	Value
0	number of first working day in the week
4	number of last working day in the week
8	number of months in the current year (current = one in which given time falls)
12	number of days in the current month
16	maximum length of AM/PM string
20	maximum length of WE string
24	maximum length of W3 string
28	maximum length of DY string
32	maximum length of ST string (may be zero)

36	maximum length of MO string
40	maximum length of M3 string
44	maximum length of TZ string

Related SWIs

None

Related vectors

None

Territory_NameToNumber (SWI &43060)

Returns the name of the given alphabet

On entry

R0 = territory number, or −1 to use current territory
R1 = pointer to territory name in the given territory's alphabet (null terminated)

On exit

R0 = territory number for given territory (0 if territory unknown)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the name of the given alphabet in the given territory's language and alphabet.

Related SWIs

None

Related vectors

None

* Commands

*Configure DST

Sets the configured value for daylight saving time to ON

Syntax

`*Configure DST`

Parameters

None

Use

*Configure DST sets the configured value for daylight saving time to ON.

The time zone is set when you configure the computer's territory, rather than by this command.

For each territory module that is registered, the territory manager uses the name of that territory's daylight saving time zone to supply an alternative name for this command. For example, if the UK territory module is registered, the command *Configure BST has the same effect as *Configure DST. (BST stands for British Summer Time.)

Example

`*Configure DST`

Related commands

*Configure NoDST

Related SWIs

None

Related vectors

None

*Configure NoDST

Sets the configured value for daylight saving time to OFF

Syntax

`*Configure NoDST`

Parameters

None

Use

*Configure NoDST sets the configured value for daylight saving time to OFF.

The time zone is set when you configure the computer's territory, rather than by this command.

For each territory module that is registered, the territory manager uses the name of that territory's standard time zone to supply an alternative name for this command. For example, if the UK territory module is registered, the command *Configure GMT has the same effect as *Configure NoDST. (GMT stands for Greenwich Mean Time.)

Example

`*Configure NoDST`

Related commands

*Configure DST

Related SWIs

None

Related vectors

None

*Configure Territory

Sets the configured default territory for the machine

Syntax

```
*Configure Territory territory
```

Parameters

territory	The name or number of the territory to use. A list of parameters can be obtained with the *Territories command.

Use

*Configure Territory sets the configured default territory for the machine.

Example

```
*Configure Territory UK
```

Related commands

*Territories

Related SWIs

None

Related vectors

None

*Territories

Lists the currently loaded territory modules

Syntax

```
*Territories
```

Parameters

None

Use

*Territories lists the currently loaded territory modules.

Example

```
*Territories
1 UK
```

Related commands

*Configure Territory

Related SWIs

None

Related vectors

None

61 The Sound system

Introduction

The Sound system provides facilities to synthesise and playback high quality digital samples of sound. Since any sound can be stored digitally, the system can equally well generate music, speech and sound effects. Eight fully independent channels are provided.

The sound samples are synthesised in real time by software. A range of different Voice Generators generate a standard set of samples, to which further ones can be added. The software also includes the facility to build sequences of notes.

The special purpose hardware provided on ARM-based systems simply reads samples at a programmable rate and converts them to an analogue signal. Filters and mixing circuitry on the main board provide both a stereo output (suitable for driving personal hi-fi stereo headphones directly, or connecting to an external hi-fi amplifier) and a monophonic or stereophonic output to the internal speaker(s).

Overview

There are four parts to the software for the Sound system: the DMA Handler, the Channel Handler, the Scheduler, and Voice Generators. These are briefly summarised below, and described in depth in later sections.

The DMA Handler

The DMA Handler manages the DMA buffers used to store samples of sound, and the associated hardware used.

The system uses two buffers of digital samples, stored as signed logarithms. The data from one buffer is read and converted to an analogue signal, while data is simultaneously written to the other buffer by a Voice Generator. The two buffers are then swapped between, so that each buffer is successively written to, then read.

The DMA Handler is activated every time a new buffer of sound samples is required. It sends a Fill Request to the Channel Handler, asking that the correct Voice Generators fill the buffer that has just been read from.

The DMA Handler also provides interfaces to program hardware registers used by the Sound system. The number of channels and the stereo position of each one can be set, the built-in loudspeaker(s) can be enabled or disabled, and the entire Sound system can also be enabled or disabled. The sample length and sampling rate can also be set.

The services of the DMA Handler are mainly provided in firmware requiring privileged supervisor status to program the system devices. It is tightly bound to the Channel Handler, sharing static data space. Consequently, this module must not be replaced or amended independently of the Channel Handler.

The Channel Handler

The Channel Handler provides interfaces to control the sound produced by each channel, and maintains internal tables necessary for the rest of the Sound system to produce these sounds.

The interfaces can be used to set the overall volume and tuning, to attach the channels to different Voice Generators, and to start sounds with given pitch, amplitude and duration.

The following internal tables are built and maintained: a mapping of voice names to internal voice numbers; a record for each channel of its volume, voice, pitch and timbre; and linear and logarithmic lookup tables for Voice Generators to scale their amplitude to the current overall volume setting.

Fill Requests issued by the DMA Handler are routed through the Channel Handler to the correct Voice Generators. This allows any tables involved to be updated.

The Channel Handler is tightly bound to the DMA Handler, sharing static data space. Consequently, this module must not be replaced or amended independently of the DMA Handler

The Scheduler

The Scheduler is used to queue Sound system SWIs. Its most common use is to play sequences of notes, and a simplified interface is provided for this purpose.

A beat counter is used which is reset every time it reaches the end of a bar. Both its tempo and the number of beats to the bar can be programmed.

You may replace this module, although it is unlikely to be necessary.

Voice Generators

Voice Generators generate and output sound samples to the DMA buffer on receiving a Fill Request from the Channel Handler. Typical algorithms that might be used to synthesise a sound sample include calculation, lookup of filtered wavetables, or frequency modulation. A Voice Generator will normally allow multiple channels to be attached.

An interface exists for you to add custom Voice Generators, expanding the range of available sounds. The demands made on processor bandwidth by synthesis algorithms are high, especially for complex sounds, so you must write them with great care.

Technical details

DMA Handler

The DMA Handler manages the hardware used by the Sound system. Two (or more) physical buffers in main memory are used. These are accessed using four registers in the sound DMA Address Generator (DAG) within the Memory Controller chip:

- The DAG *sound pointer* points to the byte of sound to be output
- The *current end* register points to the end of the DMA buffer
- The *next start/end* register pair point to the most recently filled buffer.

The sound pointer is incremented every time a byte is read by the video controller for output. When it reaches the end of the current buffer the memory controller switches buffers: the sound pointer and buffer end registers are set to the values stored in the next start and next end registers respectively. An interrupt is then issued by the I/O controller indicating the buffers have switched, and the DMA handler is entered.

The DMA Handler calls the Channel Handler with a Fill request, asking that the next buffer be filled. (See page 5-342 for details of the Channel Handler.) If this fill is completed, control returns to the DMA Handler and it makes the next start and next end registers point to the buffer just filled. If the fill is not completed then the next registers are not altered, and so the same buffer of sound will be repeated, causing an audible discontinuity.

Configuring the Sound system

The rest of this section outlines the factors that you must consider if you choose to reconfigure the Sound system.

Terminology used

- The *output period* is the time between each output of a byte.
- The *sample period* is the time between each output for a given channel.
- The *buffer period* is the time to output an entire buffer.

There are corresponding rates for each of the above.

- The *sample length* is the number of bytes in the buffer per channel.
- The *buffer length* is the total number of bytes in the buffer.

DMA Buffer period

A short buffer period is desirable to minimise the size of the buffer and to give high resolution to the length of notes; a long buffer period is desirable to decrease the frequency and number of interrupts issued to the processor. A period of approximately one centisecond is chosen as a default value, although this can be changed, for example to replay lengthy blocks of sampled speech from a disc.

Sample rate: maximum

A high sample rate will give the best sound quality. If too high a rate is sought then DMA request conflicts will occur, especially when high bandwidths are also required from the Video Controller by high resolution screen modes. To avoid such contention the output period must not be less than 4µs. Outputting a byte to one of eight channels every 4µs results in a sample period of 32µs, which gives a maximum sample rate of 31.25kHz.

Sample rate: default

The clock for the Sound system is derived from the system clock for the video controller, which is then divided by a multiple of 24. Current ARM based computers use a VIDC system clock of 24MHz; however, 20MHz and 28MHz clocks are also supported. The default output period is the shortest one that can be derived from all three clocks, thus ensuring that speech and music can be produced at the same pitch on any likely future hardware. This is 6µs, obtained as follows:

- 20MHz clock divided by 120 (5 × 24)
- 24MHz clock divided by 144 (6 × 24)
- 28MHz clock divided by 168 (7 × 24)

Outputting a byte to one of eight channels every 6µs results in a sample period of 48µs, which gives a default sample rate of 20.833kHz.

Buffer length

The DMA buffer length depends on the number of channels, the sample rate, and the buffer period. It must also be a multiple of 4 words. Using the defaults outlined above, the lengths shown in the middle two columns of the following table are the closest alternatives:

Buffer lengths for one centisecond sample, at sample rate of 20.833 kHz:

	Buffer length		Output period
1 channel	208 bytes	224 bytes	48µs
2 channels	416 bytes	448 bytes	24µs
4 channels	832 bytes	896 bytes	12µs
8 channels	1664 bytes	1792 bytes	6µs
Buffer period	0.9984cs	1.0752cs	
Interrupt rate	100.16Hz	93.01Hz	
Bytes per channel	&D0	&E0	

The system default buffer period is chosen as 0.9984 centi-seconds, thus the sample length is 208 bytes, or 52 words (13 DMA quad-word cycles). The buffer length is a multiple of this, depending on how many channels are used.

DMA Buffer format

The sound DMA system systematically outputs bytes at the programmed sample rate; each (16-byte) load of DMA data from memory is synchronised to the first stereo image position. Each byte must be stored as an eight bit signed logarithm, ready for direct output to the VIDC chip.

Multiple channel operation is possible with two, four or eight channels; in this case the data bytes for each channel must be interleaved throughout the DMA buffer at two, four or eight byte intervals. When output the channels are multiplexed into what is effectively one half, one quarter or one eighth of the sample period, so the signal level per channel is scaled down by the same amount. Thus the signal level per channel is scaled, depending on the number of channels; but the overall signal level remains the same for all multi-channel modes.

Showing the interleaving schematically:

Single channel format:

0	byte 0 chan 1	byte 1 chan 1	byte 2 chan 1	byte 3 chan 1	byte 4 chan 1	byte 5 chan 1	byte 6 chan 1	byte 7 chan 1
+8	byte 8 chan 1	byte 9 chan 1	byte 10 chan 1	byte 11 chan 1	byte 12 chan 1	byte 13 chan 1	etc...	

Output rate = 20 kHz
Image registers 0 - 7 programmed identically

Two channel format:

0	byte 0 chan 1	byte 0 chan 2	byte 1 chan 1	byte 1 chan 2	byte 2 chan 1	byte 2 chan 2	byte 3 chan 1	byte 3 chan 2
+8	byte 4 chan 1	byte 4 chan 2	byte 5 chan 1	byte 5 chan 2	byte 6 chan 1	byte 6 chan 2	etc...	

Output rate = 40 kHz
Image registers 0+2+4+8 and 1+3+5+7 programmed per channel

Four channel format:

0	byte 0 chan 1	byte 0 chan 2	byte 0 chan 3	byte 0 chan 4	byte 1 chan 1	byte 1 chan 2	byte 1 chan 3	byte 1 chan 4
+8	byte 2 chan 1	byte 2 chan 2	byte 2 chan 3	byte 2 chan 4	byte 3 chan 1	byte 3 chan 2	etc...	

Output rate = 80 kHz
Image registers 0+4, 1+5, 2+6 and 3+7 programmed per channel

Eight channel format:

0	byte 0 chan 1	byte 0 chan 2	byte 0 chan 3	byte 0 chan 4	byte 0 chan 5	byte 0 chan 6	byte 0 chan 7	byte 0 chan 8
+8	byte 1 chan 1	byte 1 chan 2	byte 1 chan 3	byte 1 chan 4	byte 1 chan 5	byte 1 chan 6	etc...	

Output rate = 160 kHz
Image registers programmed individually.

The Channel Handler manages the interleaving for you by passing the correct start address and increment to the Voice Generator attached to each channel.

Channel Handler

The Channel Handler registers itself with the DMA Handler by passing its address using Sound_Configure. At this address there must be a standard header:

Channel Handler

Offset	Value
0	pointer to fill code
4	pointer to overrun fixup code
8	pointer to linear-to-log table
12	pointer to log-scale table

The fill code handles fill requests from the DMA Handler. The Channel Handler translates the fill request to a series of calls to the Voice Generators, passing the required buffer offsets so that data from all channels correctly interleaves. Any unused channels within the buffer are set to zero by the Channel Handler so they are silent.

The overrun fixup code deals with channels that are not successfully filled within a single buffer period and hence repeat the same DMA buffer. This feature is no longer supported in RISC OS and the Channel Handler simply returns. (In the Arthur OS the offending channel was marked as overrun, the previous Channel Handler was aborted, and a new buffer fill initiated.)

The pointer to the linear-to-log table holds the address of the base of an 8 Kbyte table which maps 32-bit signed integers directly to 8-bit signed volume-scaled logarithms in a suitable format for output to the VIDC chip.

The pointer to the log-scale table holds the address of a 256-byte table which scales the amplitude of VIDC-format 8-bit signed logarithms from their maximum range down to a value scaled to the volume setting. Voice Generators should use this table to adjust their overall volume.

Sound Channel Control Block (SCCB)

The Channel Handler maintains a 256 byte Sound Channel Control Block (SCCB) for each channel. An SCCB contains parameters and flags used by Voice Generators, and an extension area for programmers to pass any essential further data. Such an extension must be well documented, and used with care, as it will lead to Voice Generators that are no longer wholly compatible with each other.

The 9 initial words hold values that are normally stored in R0 - R8 inclusive. They are saved to the SCCB using the instruction LDMIA R9,{R0-R8}

Offset	Value
0	gate bit + channel amplitude (7-bit log)
1	index to voice table
2	instance number for attached voice
3	control/status bit flags
4	phase accumulator pitch oscillator
8	phase accumulator timbre oscillator
12	number of buffer fills left to do (counter)
16	(normally working R4)
20	(normally working R5)
24	(normally working R6)
28	(normally working R7)
32	(normally working R8)
36 - 63	reserved for use by Acorn (28 bytes)
64 - 255	available for users

The flag byte indicates the state of the voice attached to the channel, and may be used for allocating voices in a polyphonic manner. Each time a Voice Generator completes a buffer fill and returns to the Channel Handler it returns an updated value for the Flags field in R0.

It is the responsibility of the Channel Handler to store the returned flag byte, and to update the other fields of each SCCB as necessary.

Note – In the Arthur OS, the flag byte was also used to detect channels that had overrun. If any were found then a call was made indirected through the fix up pointer (see above).

Voice Table

The Channel Handler uses a voice table recording the names of voices installed in the 32 available voice slots. It is always accessed through the SWI calls provided, and so its format is not defined.

Scheduler

Header

The Scheduler registers itself with the DMA Handler by passing its address using Sound_Configure. At this address there must be a pointer to the code for the Scheduler.

Use

Although the Scheduler is principally designed for queuing sound commands it can be used to issue other SWIs. Thus it could be used to control, for example, an external instrument interface (such as a Musical Instrument Digital Interface (MIDI) expansion podule), or a screen-based music editor with real-time score replay.

Extreme care must be used with the Scheduler, as it has limitations. R2 - R7 are always cleared when the SWI is issued, and the error-returning form ('X' form) of the SWI is forced. Return parameters are discarded. If pointers are to be passed in R0 or R1 then the data they address **must** be preserved until the SWI is called. If a SWI will not work within these limitations it must not be called by the Scheduler.

The Scheduler implements the queue as a circular chain of records. A stack listing the free slots is also kept. The number of free slots varies not only according to how many events are queued, but also to how the events are 'clustered'.

The queue is always accessed through the SWI calls provided, and so its precise format is not defined.

Event dispatcher

Every centisecond the beat counter is advanced according to the tempo value, and any events that fall within the period are activated in strict queuing order. Voice and parameter change events are processed and the SCCB for each Voice Generator updated as necessary by the Channel Handler, before fill requests are issued to the relevant Voice Generators.

Voice Generators

A Voice Generator is added to the Sound system by issuing a Sound_InstallVoice call, which passes its address to the Channel Handler. At this address there must be a standard header:

Header

Offset	Contents
0	`B FillCode`
4	`B UpdateCode`
8	`B GateOnCode`
12	`B GateOffCode`
16	`B Instantiate`
20	`B Free`
24	`LDMFD R13!,{pc}`
28	*Offset from start of header to voice name*

The Fill, Update, GateOn and GateOff entries provide services to fill the DMA buffer at different stages of a note, as detailed in the section entitled *Entry points for buffer filling* on page 5-347.

The Instantiate and Free entries provide facilities to attach or detach the Voice Generator to or from a channel, as detailed in the section entitled *Voice instantiation* on page 5-348.

The Install entry was originally to be called when a Voice Generator was initialised. Since Voice Generators are now implemented as Relocatable Modules, which offer exactly this service in the form of the Initialisation entry point, this field is not supported and simply returns to the caller (`LDMFD R13!,{pc}` above).

The voice name is used by the Channel Handler voice table. It should be both concise and descriptive. The offset must be positive relative – that is, the voice name must be **after** the header.

Buffer filling: entry conditions

A fill request to a Voice Generator is made by the Channel Handler using one of the four buffer fill entry points. The registers are allocated as follows:

Register	Function
R6	negative if configuration of Channel Handler changed
R7	channel number
R8	sample period in µs
R9	pointer to SCCB (Sound Channel Control Block)
R10	pointer to end of DMA buffer
R11	increment to use when writing to DMA buffer

R12	pointer to (start of DMA buffer + interleaf offset)
R13	stack (Return address is on top of stack)
R14	**do not use**

Further parameters are available in the SCCB for that channel, which is addressed by R9. See the Channel Handler description for details. The usage of the parameters depends on which of the four entry points is called.

The ARM is in IRQ mode with interrupts enabled.

Buffer filling: routine conditions

The routine must fill the buffer with 8 bit signed logarithms in the correct format for direct output to the VIDC chip:

The ARM is in IRQ mode with interrupts enabled. They must remain enabled to ensure that system devices do not have a lengthy wait to be serviced. The code for a Voice Generator must therefore be re-entrant, and R14 must not be used as a subroutine link register, since an interrupt will corrupt it. Sufficient IRQ stack depth must be maintained for system IRQ handling. You can enter SVC mode if you wish.

Buffer filling: exit conditions

When a Voice Generator has completed a buffer fill it sets a flag byte in R0, and returns to the Channel Handler using LDMFD R13!,{PC}. The flag byte shows the status of each channel, and is used to prioritise fill requests to the Voice Generators.

7							0
Q	K	I	F	A	V	F2	F1

Bit	**Meaning**
Q	Quiet (GateOff flag)
K	Kill pending (GateOn flag)
I	Initialise pending (Update flag)
F	Fill pending
A	Active (normal Fill in progress)
V	oVerrun flag (no longer supported)
F2, F1	2-bit Flush pending counter

Entry points for buffer filling

There are four different entry points for buffer filling, which are used at the different stages of a note. It is the responsibility of the Channel Handler to determine which Voice Generator to call, which entry should be used, and to update the SCCB as necessary when these calls return.

GateOn entry

The GateOn entry is used whenever a sound command is issued that requires a new envelope. Normally any previous synthesis is aborted and the algorithm restarted.

On exit a the A bit (bit 3) of the flag byte is set.

Update entry

The Update entry is used whenever a sound command is issued that requires a smooth change, without a new envelope (using extended amplitudes &180 to &1FF in the *Sound command for example). Normally the previous algorithm is continued, with only the amplitude, pitch and duration parameters supplied by the SCCB updated.

On exit the A bit (bit 3) of the flag byte is returned unless the voice is to stop sounding; for example if the envelope has decayed to zero amplitude. In these cases the F2 bit (bit 1) is set, and the Channel Handler will automatically flush out the next two DMA buffers, before becoming dormant.

Fill entry

The Fill entry is used when the current sound is to continue, and no new command has been issued.

On exit it is normal to return the same flags as for the Update entry.

GateOff entry

The GateOff entry is used to finish synthesising a sound. Simple voices may stop immediately, which is liable to cause an audible 'click'; more refined algorithms might gradually release the note over a number of buffer periods. A GateOff entry may be immediately followed by a GateOn entry.

On exit the F2 bit (bit 1) is set if the voice is to stop sounding, or the A bit (bit 3) is set if the voice is still being released.

Voice instantiation

Two entry points are provided to attach or detach a voice generator and a sound channel. On entry the ARM is in Supervisor mode, and the registers are allocated as follows:

Register	Function
R0	physical Channel number - 1 (0 to 7)
R14	usable

The return address is on top of the stack. All other registers must be preserved by the routines, which must exit using LDMFD R13!,{pc}

R0 is preserved if the call was successful, else it is altered.

Instantiate entry

The Instantiate entry is called to inform the Voice Generator of a request to attach a channel to it. Each channel attached is likely to need some private workspace. A Voice Generator should ideally be able to support eight channels. The request can either be accepted (R0 preserved on exit), or rejected (R0 altered on exit).

The usual reason for rejection is that an algorithm is slow and is already filling as many channels as it can within each buffer period: for example very complex algorithms, or ones that read long samples off disc.

Free entry

The Free entry is called to inform the Voice Generator of a request to detach a channel from it. The call **must** release the channel and preserve all registers.

Service Calls

Service_Sound (Service Call &54)

Parts of the Sound system are starting or dying

On entry

R0 =	0	DMA handler starting
	1	DMA handler dying
	2	Channel handler starting
	3	Channel handler dying
	4	Scheduler starting
	5	Scheduler dying

R1 = &54 (reason code)

On exit

R0, R1 preserved

Use

This call is made to signal that a part of the Sound system is about to start up or finish.

SWI calls

Sound_Configure (SWI &40140)

Configures the Sound system

On entry

R0 = number of channels, rounded up to 1,2,4 or 8
R1 = sample size (in bytes per channel – default 208)
R2 = sample period (in µs per channel – default 48)
R3 = pointer to Channel Handler (normally 0 to preserve system Handler)
R4 = pointer to Scheduler (normally 0 to preserve system Scheduler)

On exit

R0 - R4 = previous values

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This software interrupt is used to configure the number of sound channels, the sample period and the sample size. It can also be used by specialised applications to replace the default Channel Handler and Scheduler.

All current settings may be read by using zero input parameters.

The actual values programmed are subject to the limitations outlined earlier.

Related SWIs

None

Related vectors

None

Sound_Enable (SWI &40141)

Enables or disables the Sound system

On entry

R0 = new state:
0 for no change (read state)
1 for OFF
2 for ON

On exit

R0 = previous state
1 for OFF
2 for ON

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This software interrupt is used to enable or disable all Sound interrupts and DMA activity. This guarantees to inhibit all Sound system bandwidth consumption once a successful disable has been completed.

Related SWIs

Sound_Speaker (SWI &40143), Sound_Volume (SWI &40180)

Related vectors

None

Sound_Stereo (SWI &40142)

Sets the stereo position of a channel

On entry

R0 = channel (C) to program
R1 = image position:
0 is centre
127 for maximum right
−127 for maximum left
−128 for no change (read state)

On exit

R0 preserved
R1 = previous image position, or −128 if R0 ≥ 8 on entry

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

For N physical channels enabled, this call will program stereo registers C, C+N, C+2N… up to stereo register 8. For example, if two channels are currently in use, and channel 1 is programmed, channels 3, 5 and 7 are also programmed; if channel 3 is programmed, channels 5 and 7 are also programmed, but not channel 1.

This Software call only updates RAM copies of the stereo image registers and the new positions, in fact, take effect on the next sound buffer interrupt.

IRQ code can call this SWI directly for scheduled image movement.

Related SWIs

None

Related vectors

None

Sound_Speaker (SWI &40143)

Enables or disables the speaker(s)

On entry

R0 = new state:
- 0 for no change (read state)
- 1 for OFF
- 2 for ON

On exit

R0 = previous state
- 1 for OFF
- 2 for ON

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This software interrupt enables/disables the monophonic or stereophonic mixed signal(s) to the internal loudspeaker amplifier(s). It has no effect on the external stereo headphone/amplifier output.

This SWI disables the speaker(s) by muting the signal; you may still be able to hear a very low level of sound.

Related SWIs

Sound_Enable (SWI &40141), Sound_Volume (SWI &40180)

Related vectors

None

Sound_Volume (SWI &40180)

Sets the overall volume of the Sound system

On entry

R0 = sound volume (1 - 127) (0 to inspect last setting)

On exit

R0 = previous volume

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets the maximum overall volume of the Sound system. A change of 16 in the volume will halve or double the volume. The command scales the internal lookup tables that Voice Generators use to set their volume; some custom Voice Generators may ignore these tables and so will be unaffected.

A large amount of calculation is involved in this apparently trivial call. It should be used sparingly to limit the overall volume; the volume of each channel should then be set individually.

Related SWIs

Sound_Enable (SWI &40141), Sound_Speaker (SWI &40143)

Related vectors

None

Sound_SoundLog (SWI &40181)

Converts a signed integer to a signed logarithm, scaling it by volume

On entry

R0 = 32-bit signed integer

On exit

R0 = 8-bit signed volume-scaled logarithm

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call maps a 32-bit signed integer to an 8 bit signed logarithm in VIDC format. The result is scaled according to the current volume setting. Table lookup is used for efficiency.

Related SWIs

Sound_LogScale (SWI &40182)

Related vectors

None

Sound_LogScale (SWI &40182)

Scales a signed logarithm by the current volume setting

On entry

R0 = 8-bit signed logarithm

On exit

R0 = 8-bit signed volume-scaled logarithm

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This software interrupt maps an 8-bit signed logarithm in VIDC format to one scaled according to the current volume setting. Table lookup is used for efficiency.

Related SWIs

Sound_SoundLog (SWI &40181)

Related vectors

None

Sound_InstallVoice (SWI &40183)

Adds a voice to the Sound system

On entry

R0 = pointer to Voice Generator (0 for don't change)
R1 = voice slot specified (0 for install in next free slot, else 1 - 32)

On exit

R0 = pointer to name of previous voice (or null terminated error string)
R1 = voice number allocated (0 for FAIL to install)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This software interrupt is used by Voice Modules or Libraries to add a Voice Generator to the table of available voices. If an error occurs, this SWI does **not** set V in the usual manner. Instead R1 is zero on exit, and R0 points directly to a null-terminated error string.

Alternatively, the table of installed voices may be read by setting R0 to 0, and R1 to the slot to examine. If the slot is unused RISC OS gives a null pointer. (The Arthur OS gave a pointer to the string '*** No Voice'.)

Related SWIs

Sound_RemoveVoice (SWI &40184)

Related vectors

None

Sound_RemoveVoice (SWI &40184)

Removes a voice from the Sound system

On entry

R1 = voice slot to remove (1 - 32)

On exit

R0 = pointer to name of previous voice (or error message)
R1 is voice number de-allocated (0 for FAIL)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This software interrupt is used when Voice Modules or Libraries are to be removed from the system. It notifies the Channel Handler that a RAM-resident Voice Generator is being removed. If an error occurs, this SWI does **not** set V in the usual manner. Instead R1 is zero on exit, and R0 points directly to a null-terminated error string.

This call must also be issued before the Relocatable Module Area is Tidied, since the module contains absolute pointers to Voice Generators that are likely to exist in the RMA.

Related SWIs

Sound_InstallVoice (SWI &40183)

Related vectors

None

Sound_AttachVoice (SWI &40185)

Attaches a voice to a channel

On entry

R0 = channel number (1 - 8)
R1 = voice slot to attach (0 to detach and mute channel)

On exit

R0 preserved (or 0 if illegal channel number)
R1 = previous voice number (or 0 if not previously attached)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call attaches a voice with a given slot number to a channel. The previous voice is shut down and the new voice is reset.

Different algorithms have different internal state representations so it is not possible to swap Voice Generators in mid-sound.

Related SWIs

Sound_AttachNamedVoice (SWI &4018A)

Related vectors

None

Sound_ControlPacked (SWI &40186)

Makes an immediate sound

On entry

R0 is AAAACCCC Amp/Channel
R1 is DDDDPPPP Duration/Pitch

On exit

R0,R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call is identical to Sound_Control (SWI &40189), but the parameters are packed 16-bit at a time into low R0, high R0, low R1, high R1 respectively. It is provided for BBC compatibility and for the use of the Scheduler. The Sound_Control call should be used in preference where possible.

Related SWIs

Sound_Control (SWI &40189)

Related vectors

None

Sound_Tuning (SWI &40187)

Sets the tuning for the Sound system

On entry

R0 = new tuning value (or 0 for no change)

On exit

R0 = previous tuning value

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets the tuning for the Sound system in units of 1/4096 of an octave.

The command *Tuning 0 may be used to restore the default tuning.

Related SWIs

None

Related vectors

None

Sound_Pitch (SWI &40188)

Converts a pitch to internal format (a phase accumulator value)

On entry

R0 = 15-bit pitch value:
bits 14 - 12 are a 3-bit octave number
bits 11 - 0 are a 12-bit fraction of an octave (in units of 1/4096 octave)

On exit

R0 = 32-bit phase accumulator value, or preserved if R0 ≥ &8000 on entry

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This software interrupt maps a 15-bit pitch to an internal format pitch value (suitable for the standard voice phase accumulator oscillator).

Related SWIs

None

Related vectors

None

Sound_Control (SWI &40189)

Makes an immediate sound

On entry

R0 = channel number (1 - 8)
R1 = amplitude:
&FFF1 - &FFFF and 0 for BBC emulation amplitude (0 to -15)
&0001 - &000F **BBC envelope not emulated**
&0100 - &01FF for full amplitude/gate control:
bit 7 is 0 for gate ON/OFF
1 for smooth update (gate not retriggered)
bits 6 - 0 are 7-bit logarithm of amplitude
R2 = pitch
&0000 - &00FF for BBC emulation pitch
&0100 - &7FFF for enhanced pitch control:
bits 14 - 12 = 3-bit octave
bits 11 - 0 = 12-bit fractional part of octave
(&4000 is nominally Middle C)
&8000 + n 'n' (in range 0 - &7FFF) is phase accumulator increment
R3 = duration
&0001 - &00FE for BBC emulation in 5 centisecond periods
&00FF for BBC emulation 'infinite' time (converted to &F0000000)
> &00FF for duration in 5 centisecond periods.

On exit

R0 - R3 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call allows real-time control of a specified Sound Channel. The parameters are immediately updated and take effect on the next buffer fill.

Gate on and off correspond to the start and end of a note and of its envelope (if implemented). 'Smooth' update occurs when note parameters are changed without restarting the note or its envelope – for example when the pitch is changed to achieve a glissando effect.

If any of the parameters are invalid the call does not generate an error; instead it returns without performing any operation.

Related SWIs

Sound_ControlPacked (SWI &40186)

Related vectors

None

Sound_AttachNamedVoice (SWI &4018A)

Attaches a named voice to a channel

On entry

R0 = channel number (1 - 8)
R1 = pointer to voice name (ASCII string, null terminated)

On exit

R0 is preserved, or 0 for fail
R1 is preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call attaches a named voice to a channel. If no exact match for the name is found then an error is generated and the old voice (if any) remains attached. If a match is found then the previous voice is shut down and the new voice is reset.

Different algorithms have different internal state representations so it is not possible to swap Voice Generators in mid-sound.

Related SWIs

Sound_AttachVoice (SWI &40185)

Related vectors

None

Sound_ReadControlBlock (SWI &4018B)

Reads a value from the Sound Channel Control Block

On entry

R0 = channel number (1 - 8)
R1 = offset to read from (0 - 255)

On exit

R0 preserved (or 0 if fail, invalid channel, or invalid read offset)
R1 preserved
R2 = 32-bit word read (if R0 non-zero on exit)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call reads 32-bit data values from the Sound Channel Control Block (SCCB) for the designated channel. This call can be used to read parameters not catered for in the Sound_Control calls returned by Voice Generators, using an area of the SCCB reserved for the programmer.

Related SWIs

Sound_WriteControlBlock (SWI &4018C)

Related vectors

None

Sound_WriteControlBlock (SWI &4018C)

Writes a value to the Sound Channel Control Block

On entry

R0 = channel number (1 - 8)
R1 = offset to write to (0 - 255)
R2 = 32-bit word to write

On exit

R0 preserved (or 0 if fail, invalid channel, or invalid write offset)
R1 preserved
R2 = previous 32-bit word (if R0 non-zero on exit)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call writes 32-bit data values to the Sound Channel Control Block (SCCB) for the designated channel. This call can be used to pass parameters not catered for in the Sound_Control calls to Voice Generators, using an area of the SCCB reserved for the programmer.

Related SWIs

Sound_ReadControlBlock (SWI &4018B)

Related vectors

None

Sound_QInit (SWI &401C0)

Initialises the Scheduler's event queue

On entry

No parameters passed in registers

On exit

R0 = 0, indicating success

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call flushes out all events currently scheduled and re-initialises the event queue. The tempo is set to the default, the beat counter is reset and disabled, and the bar length set to zero.

Related SWIs

None

Related vectors

None

Sound_QSchedule (SWI &401C1)

Schedules a sound SWI on the event queue

On entry

R0 = schedule period
  −1 to synchronise with the previously scheduled event
  −2 for immediate scheduling
R1 = 0 to schedule a Sound_ControlPacked call, or SWI code to schedule (of the form &xF000000 + SWI number)
R2 = SWI parameter to be passed in R0
R3 = SWI parameter to be passed in R1

On exit

R0 = 0 for successfully queued
R0 < 0 for failure (queue full)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call schedules a sound SWI call. If the beat counter is enabled the schedule period is measured from the last start of a bar, otherwise it is measured from the time the call is made.

A schedule time of −1 forces the new event to be queued for activation concurrently with the previously scheduled one.

The event is typically a Sound_ControlPacked type call, although any other sound SWI may be scheduled. There are limitations: R2 - R7 are always cleared, and any return parameters are discarded. If pointers are to be passed in R0 or R1 then any

associated data must still remain when the SWI is called (the workspace involved must not have been reused, the Window Manager must not have paged it out, and so on).

Related SWIs

Sound_QFree (SWI &401C3)

Related vectors

None

Sound_QRemove (SWI &401C2)

This SWI call is for use by the Scheduler only. **You must not use it** in your own code.

Sound_QFree (SWI &401C3)

Returns minimum number of free slots in the event queue

On entry

No parameters passed in registers

On exit

R0 = number of guaranteed slots free
R0 < 0 indicates over worst case limit, but may still be free slots

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the minimum number of slots guaranteed free. The calculation assumes the worst case of data structure overheads that could occur, so it is likely that more slots can in fact be used. If this guaranteed free slot count is exceeded this call will return negative values, and the return status of Sound_QSchedule must be carefully monitored to observe when overflow occurs.

Related SWIs

Sound_QSchedule (SWI &401C1)

Related vectors

None

Sound_QSDispatch (SWI &401C4)

This SWI call is for use by the Scheduler only. **You must not use it** in your own code.

Sound_QTempo (SWI &401C5)

Sets the tempo for the Scheduler

On entry

R0 = new tempo (or 0 for no change)

On exit

R0 = previous tempo value

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This command sets the tempo for the Scheduler. The default tempo is &1000, which corresponds to one beat per centisecond; doubling the value doubles the tempo (ie &2000 gives two beats per centisecond), while halving the value halves the tempo (ie &800 gives half a beat per centisecond).

The parameter can be thought of as a hexadecimal fractional number, where the three least significant digits are the fractional part.

Related SWIs

Sound_QBeat (SWI &401C6)

Related vectors

None

Sound_QBeat (SWI &401C6)

Sets or reads the beat counter or bar length

On entry

R0 = 0 to return current beat number
R0 = –1 to return current bar length
R0 < –1 to disable beat counter and set bar length 0
R0 = +N to enable beat counter with bar length N (counts 0 to N–1)

On exit

R0 = current beat number (R0 = 0 on entry), otherwise the previous bar length.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The simplest use of this call is to read either the current value of the beat counter or the current bar length.

When the beat counter is disabled both it and the bar length are reset to zero. All scheduling occurs relative to the time the scheduling call is issued.

When the beat counter is enabled it is reset to zero. It then increments, resetting every time it reaches the programmed bar length (N–1). Scheduling using Sound_QSchedule then occurs relative to the last bar reset; however, scheduling using *QSound is still relative to the time the command is issued.

Related SWIs

Sound_QTempo (SWI &401C5)

Related vectors

None

Sound_QInterface (SWI &401C7)

This SWI call is for use by the Scheduler only. **You must not use it** in your own code.

* Commands

*Audio

Turns the Sound system on or off

Syntax

`*Audio On|Off`

Parameters

On or `Off`

Use

*Audio turns the Sound system on or off. Turning the Sound system off silences it completely, stopping all Sound interrupts and DMA activity. Turning the Sound system back on restores the Sound DMA and interrupt system to the state it was in immediately prior to being turned off.

All Channel Handler and Scheduler activity is effectively frozen during the time the Audio system is off, but software interrupts are still permitted, even if no sound results.

Example

`*Audio On`

Related commands

*Speaker, *Volume

Related SWIs

Sound_Enable (SWI &40141)

Related vectors

None

*ChannelVoice

Assigns a voice to a channel

Syntax

`*ChannelVoice channel voice_number|voice_name`

Parameters

channel	1 to 8
voice_number	1 to 16, as given by *Voices; or 0 to mute the channel
voice_name	name, as given by *Voices

Use

*ChannelVoice assigns a voice (sound) to one of the eight independent channels used for sound output. It is better to specify the voice by name rather than by number, since the name is independent of the order in which the voices are loaded. Note that the name is case sensitive. Alternatively, you can mute a channel by assigning it a voice slot of 0.

By default, only the first of the eight voices will be available. To make others available, use the SWI Sound_Configure, or enter BASIC and type

>VOICES *n*

where *n* is 2, 4 or 8 (the number of sound channels to enable). Do not, however, confuse the VOICES command in BASIC with *Voices, the command described in this manual.

Example

`*ChannelVoice 1 StringLib-Pluck`

Related commands

*Stereo, *Voices

Related SWIs

Sound_Configure (SWI &40140), Sound_AttachVoice (SWI &40185), Sound_AttachNamedVoice (SWI &4018A)

Related vectors

None

*Configure SoundDefault

Sets the configured speaker setting, volume and voice

Syntax

`*Configure SoundDefault` *`speaker volume voice_number`*

Parameters

speaker	0 to disable the internal loudspeaker(s) – although the headphones remain enabled 1 to enable the internal loudspeaker(s)
volume	0 (quietest) to 7 (loudest)
voice_number	1 to 16, as given by *Voices

Use

*Configure SoundDefault sets the configured speaker setting, volume and voice. The voice number is assigned to channel 1 only (the default system Bell channel).

Example

```
*Configure SoundDefault 1 7 1
```

Related commands

None

Related SWIs

None

Related vectors

None

*QSound

Generates a sound after a given delay

Syntax

`*QSound` *`channel amplitude pitch duration beats`*

Parameters

channel	1 to 8
amplitude	0 (silent) and &FFFF (almost silent) down to &FFF1 (loud) for a linear scale – or &100 (silent) to &17F (loud) for a logarithmic scale, where a change of 16 will halve or double the amplitude
pitch	0 to 255, where each unit represents a quarter of a semitone, with a value of 53 producing middle C – or 256 (&100) to 32767 (&7FFF), where the bottom 12 bits give the fraction of an octave, and the top three bits the octave, with a value of 16384 (&4000) producing middle C
duration	0 to 32767 (&8000), giving the duration of the note in twentieths of a second – but a value of 255 (&FF) gives a note of infinite duration (limited by the envelope, if present)
beats	beats delay before the sound is generated, occurring at the rate set by *Tempo

Use

*QSound generates a sound after a given delay. It is identical in effect to issuing a *Sound command after the specified number of beats have occurred. The channel will only sound if at least that number of channels have been selected, and the channel has a voice attached.

Example

```
*QSound 1 &FFF2 &5800 10 50
```

Related commands

*Sound, *Tempo

Related SWIs

Sound_QSchedule (SWI &401C1)

Related vectors

None

*Sound

Generates an immediate sound

Syntax

`*Sound channel amplitude pitch duration`

Parameters

channel	1 to 8
amplitude	0 (silent) and &FFFF (almost silent) down to &FFF1 (loud) for a linear scale – or &100 (silent) to &17F (loud) for a logarithmic scale, where a change of 16 will halve or double the amplitude
pitch	0 to 255, where each unit represents a quarter of a semitone, with a value of 53 producing middle C – or 256 (&100) to 32767 (&7FFF), where the bottom 12 bits give the fraction of an octave, and the top three bits the octave, with a value of 16384 (&4000) producing middle C
duration	0 to 32767 (&8000), giving the duration of the note in twentieths of a second – but a value of 255 (&FF) gives a note of infinite duration (limited by the envelope, if present)

Use

*Sound generates an immediate sound. The channel will only sound if at least that number of channels have been selected, and the channel has a voice attached.

Example

`*Sound 1 &FFF2 &5800 10`

Related commands

*QSound

Related SWIs

Sound_ControlPacked (SWI &40186), Sound_Control (SWI &40189)

Related vectors

None

*Speaker

Turns the internal speaker(s) on or off

Syntax

`*Speaker On|Off`

Parameters

On or `Off`

Use

*Speaker turns the internal speaker(s) on or off. It does not effect the 3.5 mm stereo jack socket, which you can still use to play the sound through headphones or an amplifier.

You may still be able to hear a very low level of sound, as this command mutes the speaker(s) rather than totally disabling them.

Example

`*Speaker Off`

Related commands

*Audio, *Volume

Related SWIs

Sound_Speaker (SWI &40143)

Related vectors

None

*Stereo

Sets the position in the stereo image of a sound channel

Syntax

`*Stereo channel position`

Parameters

channel	1 to 8
position	−127(full left) to +127(full right)

Use

*Stereo sets the position in the stereo image of a sound channel.

Example

`*Stereo 2 100` *set channel 2 output to come predominantly from the right*

Related commands

*ChannelVoice, *Voices

Related SWIs

Sound_Stereo (SWI &40142)

Related vectors

None

*Tempo

Sets the tempo for the Scheduler

Syntax

`*Tempo tempo`

Parameters

tempo	0 to &FFFF (default &1000)

Use

*Tempo sets the Sound system tempo (the rate of the beat counter). The default tempo is &1000, which corresponds to one beat per centisecond; doubling the value doubles the tempo (so &2000 gives two beats per centisecond), while halving the value halves the tempo (so &800 gives half a beat per centisecond).

Example

`*Tempo &1200`

Related commands

*QSound

Related SWIs

Sound_QTempo (SWI &401C5)

Related vectors

None

*Tuning

Alters the overall tuning of the Sound system

Syntax

`*Tuning relative_change`

Parameters

relative_change −16383 to 16383 (0 resets the default tuning)

Use

*Tuning alters the overall tuning of the Sound system. A value of zero resets the default tuning. Otherwise, the tuning is changed relative to its current value in units of 1/4096 of an octave.

Example

`*Tuning 64`

Related commands

None

Related SWIs

Sound_Tuning (SWI &40187)

Related vectors

None

*Voices

Displays a list of the installed voices

Syntax

`*Voices`

Parameters

None

Use

*Voices displays a list of the installed voices by name and number, and shows which voice is assigned to each of the eight channels. A voice can be attached to a channel even if that channel is not currently in use.

Example

```
*Voices
          Voice Name
12            1 Wave-Synth-Beep
  34          2 StringLib-Soft
              3 StringLib-Pluck
              4 StringLib-Steel
              5 StringLib-Hard
    56        6 Percussion-Soft
              7 Percussion-Medium
      78      8 Percussion-Snare
              9 Percussion-Noise
^^^^^^^^ Channel Allocation Map
```

Related commands

*ChannelVoice, *Stereo

Related SWIs

Sound_InstallVoice (SWI &40183)

Related vectors

None

*Volume

Sets the maximum overall volume of the Sound system

Syntax

`*Volume volume`

Parameters

volume 1 (quietest) to 127 (loudest)

Use

*Volume sets the maximum overall volume of the Sound system. A change of 16 in the volume parameter will halve or double the actual volume.

The command scales the internal lookup tables that Voice Generators use to set their volume (Some custom Voice Generators may ignore these tables and so will be unaffected.) A large amount of calculation is involved in this. You should therefore use this command sparingly, and only to limit the overall volume of all channels; if a single channel is too loud or soft, you should alter just that channel's volume.

Example

`*Volume 127`

Related commands

*Audio, *Configure SoundDefault, *Speaker

Related SWIs

Sound_Volume (SWI &40180)

Related vectors

None

Application notes

The most likely change to the Sound system is to add Voice Generators, thus providing an extra range of sounds. Each Voice Generator must conform to the specifications given earlier in the section entitled *Voice Generators* on page 5-345, and those given below. The speed and efficiency of Voice Generator algorithms is paramount, and requires careful attention to coding; some suggested code fragments are given to help you.

Code will not run fast enough in ROM, so ROM templates or user code templates must be copied into the Relocatable Module Area where they will execute in fast sequential RAM. If the RMA is to be tidied, all installed voices must be removed using the Sound_RemoveVoice call, then reinstalled using the Sound_InstallVoice call.

Voice libraries are an efficient way of sharing common code and data areas; these must be built as Relocatable Modules which install sets of voices, preferably with some form of library name prefix.

Buffer filling algorithms

The Channel Handler sets up three registers (R12,11,10) which give the start address, increment and end address for correct filling with interleaved sound samples. The interleave increment has the value 1, 2, 4 or 8, and is equal to the number of channels. This code is an example of how these registers should be used:

```
.loop
        ...
        ...                  ; e.g. form VIDC format 8 bit signed log in Rs
        STRB  Rs,[R12],R11   ; store, and bump ptr
        CMPS  R12,R10        ; check for end
        BLT   loop           ; and loop until fill complete
```

The DMA buffer is always a multiple of 4 words (16 bytes) long, and word aligned. Loop overheads can therefore be cut down by using two byte store operations. A further improvement is possible if R11, the increment, is one; this implies that values are to be stored sequentially, so word stores may be used.

Example code fragments

The fundamental operations performed by nearly all voice generators involve Oscillators, Table lookup and Amplitude modulation. In addition, some algorithms (plucked string and drum in particular) require random bit generators. Simple in-line code fragments are briefly outlined for each of these.

In all cases the aim is to produce the most efficient, and wherever possible highly sequential, ARM machine code; in most algorithms the aim must be to get as many working variables into registers as possible, and then adapt the synthesis algorithms wherever possible to use the high-speed barrel shifter to effect.

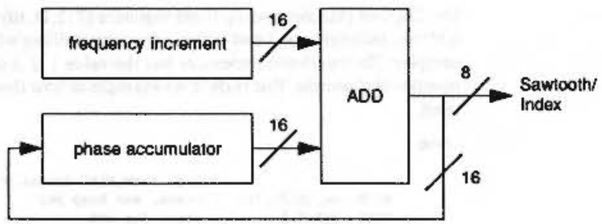
Oscillator coding

The accumulator-divider is the most useful type of oscillator for most voices. A frequency increment is added to a phase accumulator register and the high-order bits of the resulting phase provide the index to a wavetable. Alternatively, the top byte can be directly used as a sawtooth waveform.

The frequency of the oscillator is linearly related to the frequency increment. Vibrato effects can be obtained by modulating the frequency increment.

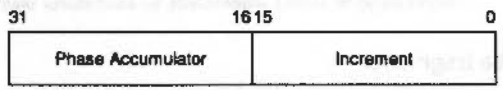
Sixteen-bit registers provide good audible frequency resolution, and are used in many digital hardware synthesizer products. The 32-bit register width of the ARM is ideally split 16/16 bits for phase/increment.

Schematically



Coding

Register field assignment: Rp




```
ADD   Rp,Rp,Rp,LSL #16   ; phase accumulate
```


Changing parameters or the voice table being used is best done at or close to zero-crossing points, to avoid noise generation. If wavetables are arranged with zero-crossing aligned to the start and end of the table then it is simple to add a branch to appropriate code.

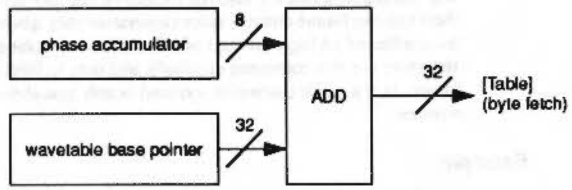
```
ADDS  Rp,Rp,Rp,LSL #16  ; phase accumulate
BCS   Update            ; only take branch if past zero crossing
```

Wavetable access coding

Normally fixed-length (256-byte or a larger power of two) wavetables are used by most voice generator modules. The high bits of the phase accumulator are added to a wavetable base pointer to access the sample byte within the table:

Schematically

For a 256-byte table:



Coding

```
LDRB Rs,[Rt,Rp,LSR #24]
```

where the most significant 8 bits of *Rp* contain the Phase index, *Rt* is the Table base pointer, and *Rs* is the register used to store the sample.

Amplitude modulation coding

The amplitude of the resultant byte may be altered for three reasons: firstly to scale for the overall volume setting, secondly to scale for the channel's volume setting, and lastly to provide enveloping.

Overall volume

If the overall volume setting changes, then your Update entry point will be called. You can cope with the change in two ways. The first is to re-scale all the values in the wavetable, using the SWI calls Sound_SoundLog or Sound_LogScale. This has the advantage that buffer filling is faster as the values are already scaled, but has the disadvantage that the wavetables might be stored to a lower resolution resulting in increased noise levels.

The alternative is to re-scale the values between reading them from the wavetable and outputting them, as in the example voice given later. The reverse then applies: buffer filling is slower, but noise is reduced. This method is preferred, so long as the algorithm is still able to fill the buffer within the required period.

Channel volume

The channel's volume setting should be used by all well-behaved Voice Generators. The volume is passed to the Voice Generator by the Channel Handler in the SCCB, as a signed 8 bit logarithm, but in a different format to that used by the VIDC chip:

Amplitude Byte Data Format:

7	6	5	4	3	2	1	0
0	Logarithm						

VIDC 8-bit sample format:

7	6	5	4	3	2	1	0
Logarithm							S
							Sign bit

Coding

The coding is easiest if the values are treated as fractional quantities, and is then reduced to subtracting logarithms and checking for underflow:

Ra contains amplitude in range 0 to 127
Rs contains sample data in range –127 to +127 (sign bit LSB)

```
; do this each time Voice Generator is entered
RSB   Ra,Ra,#127          ; make attenuation factor

; do this inside loop, before each write to buffer
SUBS  Rs,Rs,Ra,LSL #1     ; note shift to convert to VIDC format
MOVMI Rs,#0               ; correct for underflow
```

Note – The example voice shows how this can be combined with use of the volume-scaled lookup table to scale for both the overall and channel volume on each fill.

Envelope coding

Envelopes (if used) must be coded within the Voice Generator. A lookup table must be defined giving the envelope shape. This is then accessed in a similar manner to a wavetable, using the timbre phase accumulator passed in the SCCB. The sample byte is then scaled using this value, as shown above.

If you continue after a gate off, you must store your own copy of the volume, as any value in the SCCB will be overwritten.

Linear to logarithmic conversion

Algorithms which work with linear integer arithmetic may use the Channel Handler linear-log table directly to fill buffers efficiently. The table is 8 Kbyte in length, to allow the full dynamic range of the VIDC sound digital to analogue converter to be utilised. The format is chosen to allow direct indexing using barrel-shifted 32-bit integer values. The values in the table are scaled according to the current volume setting.

Coding

```
; to access the lookup table pointer during initialisation:
        MOV     R0,#0
        MOV     R1,#0
        MOV     R2,#0
        MOV     R3,#0       ; get Channel Handler base
        MOV     R4,#0
        SWI     "XSound_Configure"
        BVS     error_return
        LDR     R8,[R3,#8] ; lin-to-log pointer

  ; in line buffer filling code:
  ; linear 32-bit value in R0
        LDRB    R0,[R8,R0,LSR #19] ; lin -> log
        STRB    R0,[R12],R11        ; output to DMA buffer
```

Random bit generator code

An efficient pseudo-random bit generator can be implemented using two internal registers. This provides noise which is necessary for some sounds, percussion in particular. One register is used as a multi-tap shift register, loaded with a seed value; the second is loaded with an XOR bit mask constant (&1D872B41). The sequence produced has a length of 4294967295. The random carry bit setting by the simple code fragment outlined below allows conditional execution on carry set (or cleared):

Coding

```
MOVS  R8,R8,LSL #1 ; set random carry
EORCS R8,R8,R9
xxxCC  ; do this...
yyyCS  ; ...or alternately this
```


Example program

This program shows a complete Voice Generator. It builds a wavetable containing a sine wave at maximum amplitude. Scaling is performed when the table is read:

```
REM -> WaveVoice
 :
 DIM WaveTable% 255
 DIM Code%     4095
 :
 SYS "Sound_Volume",127 TO UserVolume
 FOR s%=0 TO 255
   SYS "Sound_SoundLog",&7FFFFFFF*SIN(2*PI*s%/256) TO WaveTable%?s%
 NEXT s% : REM build samples at full volume
 SYS "Sound_Volume",UserVolume TO UserVolume
 REM and restore volume to value on entry
 :
 FOR C=0 TO 2 STEP 2
 P%=Code%
 [ OPT C
 ;**************************************
 ;*   VOICE CO-ROUTINE CODE SEGMENT    *
 ;**************************************
 ; On installation, point Channel Handler voice
 ; pointers to this voice control block
 ;  (return address always on top of stack)
 .VoiceBase
    B        Fill
    B        Fill                 ; update entry
    B        GateOn
    B        GateOff
    B        Instance             ; Instantiate entry
    LDMFD    R13!,{PC}            ; Free entry
    LDMFD    R13!,{PC}            ; Initialise
    EQUD     VoiceName - VoiceBase
 ;
 .VoiceName EQUS "WaveVoice"
            EQUB 0
    ALIGN
 ;**************************************
 .LogAmpPtr EQUD 0
 .WaveBase  EQUD WaveTable%
 ;**************************************
 .Instance ; any instance must use volume scaled log amp table
    STMFD    R13!,{R0-R4}         ; save registers
    MOV      R0,#0
    MOV      R1,#0
    MOV      R2,#0
    MOV      R3,#0
    MOV      R4,#0
    SWI      "XSound_Configure"
    LDRVC    R0,[R3,#12]          ; get address of volume scaled log amp table
    STRVC    R0,LogAmpPtr         ; and store
    STRVS    R0, [R13]            ; return error pointer
```

```
   LDMFD   R13!,{R0-R4,PC}    ; restore registers and return
;**************************************
;*  VOICE BUFFER FILL ROUTINES        *
;**************************************
; on entry:
;    r0-r8 available
;    r9  is SoundChannelControlBlock pointer
;    r10 DMA buffer limit (+1)
;    r11 DMA buffer interleave increment
;    r12 DMA buffer base pointer
;    r13 Sound system Stack with return address and flags
;      on top (must LDMFD R13!,{...,pc}
; NO r14 - IRQs are enabled and r14 is not usable
.GateOn
   LDR     R0,WaveBase         ; wavetable base
   STR     R0,[R9,#16]         ; set up in SCCB as working register 5
   LDR     R0,LogAmpPtr        ; volume scaled log amp table
   STR     R0,[R9,#20]         ; set up as working register 6
;**************************************
.Fill
   LDMIA   R9,{R1-R6}          ; pick up working registers from SCCB
   AND     R1,R1,#&7F          ; mask R1 so only channel amplitude remains
; R1 is amp (0-127)        R2 is pitch phase acc
; R3 is timbre phase acc  R4 is duration
; R5 is wavetable base    R6 is amp table base
                               ; move sign bit -> VIDC format log
   LDRB    R1,[R6,R1,LSL #1]   ; and lookup amp scaled to overall volume
   MOV     R1,R1,LSR #1        ; move sign bit back again
   RSB     R1,R1,#127          ; make attenuation factor
.FillLoop
   ADD     R2,R2,R2,LSL #16    ; advance waveform phase
   LDRB    R0,[R5,R2,LSR #24]  ; get wave sample
   SUBS    R0,R0,R1,LSL #1     ; scale amplitude for overall & channel volumes
   MOVMI   R0,#0               ; and correct underflow
   STRB    R0,[R12],R11        ; generate output sample
   ADD     R2,R2,R2,LSL #16    ; repeated in line four times...
   LDRB    R0,[R5,R2,LSR #24]
   SUBS    R0,R0,R1,LSL #1
   MOVMI   R0,#0
   STRB    R0,[R12],R11
   ADD     R2,R2,R2,LSL #16
   LDRB    R0,[R5,R2,LSR #24]
   SUBS    R0,R0,R1,LSL #1
   MOVMI   R0,#0
   STRB    R0,[R12],R11
   ADD     R2,R2,R2,LSL #16
   LDRB    R0,[R5,R2,LSR #24]
   SUBS    R0,R0,R1,LSL #1
   MOVMI   R0,#0
   STRB    R0,[R12],R11        ; end of repeats...
   CMP     R12,R10             ; check for end of buffer fill
   BLT     FillLoop            ; loop if not
; check for end of note
   SUBS    R4,R4,#1            ; decrement centisec count
   STMIB   R9,{R2-R5}          ; save registers to SCCB
```



```
   MOVPL   R0,#%00001000       ; voice active if still duration left
   MOVMI   R0,#%00000010       ; else force flush
   LDMFD   R13!,{PC}           ; return to level 1
;**************************************
.GateOff
   MOV     R0,#0
.FlushLoop
   STRB    R0,[R12],R11        ; fill buffer with zeroes
   STRB    R0,[R12],R11
   STRB    R0,[R12],R11
   STRB    R0,[R12],R11
   CMP     R12,R10
   BLT     FlushLoop
; CAUSE level 1 TO FLUSH once more
   MOV     R0,#%00000001       ; set flag to flush one more buffer
   LDMFD   R13!,{PC}           ; return to level 1
]
NEXT C
:
DIM OldVoice%(8)
SYS "Sound_InstallVoice",VoiceBase,0 TO a%,Voice%
FOR v%=1 TO 8
  SYS "Sound_AttachVoice",v%,0 TO z%,OldVoice%(v%)
  VOICE v%,"WaveVoice"
NEXT
:
ON ERROR PROCRestoreSound : END
:
VOICES 8
*voices
SOUND 1,&17F,53,10 :REM activate channel 1!
PRINT''"any key to make a noise, <ESCAPE> to finish"
:
C%=1
REPEAT
  K%=INKEY(1)
  IF K%>0 THEN
    SOUND C%,&17F,K%,100
    C%+=1 : IF C%>8 THEN C%=1
  ENDIF
UNTIL 0
:
DEF PROCRestoreSound
  ON ERROR OFF
  REPORT:PRINT ERL
  SYS "Sound_RemoveVoice",0,Voice%
  FOR v%=1 TO 8
    SYS "Sound_AttachVoice",v%,OldVoice%(v%)
  NEXT
  VOICES 1
  *voices
  PRINT''
ENDPROC
```

62 WaveSynth

Introduction

WaveSynth is a module that provides a voice generator which is used for the default system bell.

You can load a new wavetable into WaveSynth as a module initialisation parameter, eg:

```
REM > Source
obj$="<Obey$Dir>.!RunImage"
DIM MC%1000,L%-1
FOR I%=8 TO 10 STEP 2
P%=MC%
[OPTI%
.start
 MOV R0, #14
 ADR R1, instantiation
 SWI "XOS_Module"
 MOV PC, R14

.instantiation
 EQUS "WaveSynth%Organ <Obey$Dir>.Organ01"+CHR$0
]:NEXT
OSCLI "Save "+obj$+" "+STR$~start+" "+STR$~P%
OSCLI "SetType "+obj$+" Utility"
OSCLI "Stamp "+obj$
```

In RISC OS 2.0 it also provided a SWI for its own internal use. This has since been removed

For more information about the use of sound in RISC OS, refer to the chapter entitled *The Sound system* on page 5-335.

63 The Buffer Manager

Introduction and Overview

The buffer manager acts as a global buffer managing system, providing a set of calls for setting up a buffer, inserting and removing data from a buffer, and removing a buffer. It is also possible to setup an area of memory to be used as a buffer. The buffer manager extends the INSV, REMV and CNPV vector calls to provide access to these buffers and allow block transfers.

The buffer manager is used by DeviceFS to provide buffers for the various devices that can be accessed. A device may be linked to a buffer, supply routines to be called when data enters the buffer and also a routine to be called when a buffer is removed (or a new device is attached).

When registering or creating a buffer it is possible to force a specific buffer handle, if this feature is not used then the manager will assign a unique handle to it. It should be noted that buffer handles are no longer stored as eight bit quantities.

Block transfers are signalled by setting bit 31 of the buffer handle, anything which can be performed on a byte by byte basis can also be performed on a block, for example, examining the buffer contents.

A number of vectors, events, service calls and upcalls have been modified or created to enable the buffer manager to function efficiently.

Vectors

The SWIs for the buffer manager module allow you to modify the actual buffer itself, but do not supply a way of inserting and removing data from these buffers. Extensions have been made to the following vectors to handle the inserting and removing of data from the buffers, these calls have also been extended to allow block inserts. For more details of these vector calls see the chapter entitled *Software vectors* on page 1-59.

- InsV – inserts a byte in a buffer
- RemV – removes a byte from a buffer
- CnpV – counts number of entries in a buffer, or purges contents of a buffer

Events

The changes required to enable InsV, RemV and CnpV to cope with the new buffers and block transfers have required the following events to be extended to enable them to indicate that a block transfer occurred. For more details of these events see the chapter entitled *Events* on page 1-137.

- Event_OutputEmpty – issued when the last character is removed from a buffer
- Event_InputFull – generated when a character or block is inserted and it failed
- Event_CharInput – issued when the flags word is set up to indicate that character entering buffer events should be generated.

Service calls

The service call Service_BufferStarting has been added to allow modules which wish to register buffers with the buffer manager to do so. For more details of this service call see *Service_BufferStarting (Service Call &6F)* on page 5-409.

Upcalls

Allows the buffer manager to communicate with owners. For more details of these upcalls see the chapter entitled *Communications within* RISC OS on page 1-167

- Upcall_BufferFilling – passed when data is inserted into the buffer and the free space passes by the specified threshold
- Upcall_BufferEmptying – issued by the buffer manager when the free space in the buffer is greater than the current threshold.

Service Calls

Service_BufferStarting (Service Call &6F)

Allows modules to register buffers with the buffer manager

On entry

R1 = &6F (reason code)

On exit

All registers preserved

Use

This call is passed around after the module has been initialised or reset. It allows modules which wish to register with the buffer manager to do so.

When the service is received all SWIs are valid.

SWI calls

Buffer_Create (SWI &42940)

Claims an area of memory from the RMA and links it into the buffer list

On entry

R0 = flags for the buffer:

bit 0	0 ⇒	buffer is dormant and should be woken up when data enters it
bit 1	1 ⇒	buffer generates 'output buffer empty' events
bit 2	1 ⇒	buffer generates 'input buffer full' events
bit 3	1 ⇒	buffer generates 'upcalls when free space threshold crossed' events
bits 4-31	⇒	reserved for future expansion, should be set to 0 on creation

R1 = size of buffer to be created
R2 = buffer handle to assign to buffer (–1 ⇒ does not matter)

On exit

V = 0 ⇒ R0 = buffer handle
= 1 ⇒ R0 = pointer to error block

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI claims an area of memory from the RMA and links it into the buffer list:

If R2 = –1 the buffer manager will attempt to find a unique handle; else the buffer manager will check that the handle specified is unique and if it is assign it to that buffer.

The flags word is used to indicate what should happen when data is being inserted and removed from the buffer.

Bit 0 used to indicate if the character entering buffer events should be issued.

Bit 1 used to indicate if 'output buffer empty' events should be issued for this buffer.

Bit 1 is used to indicate if the device attached to the buffer has been woken up. If this bit = 0 then the device is dormant and needs to be woken. When a device is attached and data is put into the buffer this bit is checked, if it = 0 then the wake up code for the device will be called allowing any device to wake up any hardware it may be driving and to start processing data within the buffer.

Bit 2 used to indicate if 'input buffer full' events should be issued for this buffer.

Bit 3 used to indicate if 'upcalls when free space threshold crossed' events should be issued for this buffer.

On exit R0 contains the buffer handle being used or a pointer to an error block.

Related SWIs

Buffer_Remove (SWI &42941)

Related vectors

None

Buffer_Remove (SWI &42941)

Deregisters the buffer from the active list

On entry

R0 = handle of buffer to be removed

On exit

V = 1 ⇒ R0 = pointer to error block, else preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI should only be made on buffers created using Buffer_Create.

It attempts to deregister the buffer from the active list. If successful it will attempt to free the memory relating to that buffer.

Related SWIs

Buffer_Create (SWI &42940)

Related vectors

None

Buffer_Register (SWI &42942)

Registers an area of memory as a buffer

On entry

R0 = flags for buffer:

bit 0	0 ⇒	buffer is dormant and should be woken up when data enters it
bit 1	1 ⇒	buffer generates 'output buffer empty' events
bit 2	1 ⇒	buffer generates 'input buffer full' events
bit 3	1 ⇒	buffer generates 'upcalls when free space threshold crossed' events
bits 4-31	⇒	reserved for future expansion, should be set to 0 on creation

R1 = pointer to start of memory for buffer
R2 = pointer to end of buffer (+1)
R3 = handle to be assigned to buffer (−1 ⇒ if to be generated)

On exit

V = 0 ⇒ R0 = buffer handle
= 1 ⇒ R0 = pointer to error block

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI registers an area of memory as a buffer, the routine accepts similar parameters to Buffer_Create, but instead of the call claiming that area of memory for you, you must actually specify the buffers start and end.

It is not advised that you put buffers in the application workspace, this area of memory can be switched out when someone else tries to access it. It is however possible for your task if it is going to be the only one using this buffer, and it will only be accessed whilst the task is currently paged in to register a buffer within its workspace.

For further details about the flags word and the specified buffer handle see *Buffer_Create* (SWI &42940) on page 5-410.

Related SWIs

Buffer_Deregister (SWI &42943)

Related vectors

None

Buffer_Deregister (SWI &42943)

Unlinks a buffer from the active list

On entry

R0 = handle of buffer to be deregistered

On exit

V = 0 ⇒ all preserved
= 1 ⇒ R0 = pointer to error block

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI will simply unlink a buffer from the active list, the data within the buffer will be purged and any access to this buffer via INSV, REMV and CNPV will be ignored.

Do not use this call if you have created the buffer using Buffer_Create, instead use Buffer_Remove which releases any memory that may have been claimed.

Related SWIs

Buffer_Register (SWI &42942)

Related vectors

None

Buffer_ModifyFlags (SWI &42944)

Modifies the flags word stored with each buffer

On entry

R0 = handle of buffer to be modified
R1 = EOR mask
R2 = AND mask

On exit

R1 = old value
R2 = new value

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI allows you to modify the flags word stored with each buffer, the SWI allows two registers to be AND'd and then EOR'd with the current flags word. On exit the old and new values of this word are returned to the caller.

R1 and R2 are applied as follows:

new = (old AND R2) EOR R1

The caller should not modify reserved flag bits when issuing this call, i.e. bits 4-31 should be set in R2 and clear in R1.

Related SWIs

None

Related vectors

None

Buffer_LinkDevice (SWI &42945)

Links a set of routines to the specified device

On entry

R0 = buffer handle
R1 = pointer to code to wake up device when needed (0 ⇒ none)
R2 = pointer to code to call when device is to be detached (0 ⇒ cannot be detached)
R3 = private word to be passed in
R4 = pointer to workspace for above routines

On exit

V = 1 ⇒ R0 = pointer to error block, else all preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI links a set of routines to the specified device. The caller supplies two routines, one to be called when data enters the buffer and another to be called when someone else attempts to link to the buffer.

R1 contains a pointer to the routine to be called when data enters the buffer and it is currently marked dormant. The routine can be entered in any mode and with FIQs or IRQs enabled or disabled. The mode should be preserved as should the interrupt state.

The registers to the wake up code are setup as follows:

On entry

R0 = buffer handle
R8 = private word (specified in R3 in SWI Buffer_LinkDevice)
R12 = pointer to workspace for routine (specified in R3 in SWI Buffer_LinkDevice)

On exit

all should be preserved, including PSR

The buffer manager automatically marks the buffer as active (non-dormant) before calling the wake up code.

If the caller to Buffer_LinkDevice specifies a routine pointer equal to zero then no wake up call is made.

The second routine supplied is a routine to be called whenever the owner of the buffer is about to change; if this value is zero then the device is indicating that the owner can never be changed and changing it will result in an error.

The routine if supplied gets called as follows:

On entry

R0 = buffer handle
R8 = private word
R12 = pointer to workspace for the calls

On exit

V = 0 ⇒ all preserved
= 1 ⇒ R0 = pointer to error block

On return from this routine the routine can return an error, any errors returned halt the detach process. The detach routines are called when someone attempts to kill the buffer manager module, this results in an error and the buffer manager refuses to die.

When attaching to a buffer it is possible that the SWI will fail, this is likely to be because the current owner is refusing to detach itself.

Related SWIs

Buffer_ModifyFlags (SWI &42944)

Related vectors

None

Buffer_UnlinkDevice (SWI &42946)

Unlinks a device from a buffer

On entry

R0 = buffer handle

On exit

V = 0 ⇒ all preserved and device detached
= 1 ⇒ R0 = pointer to error block

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI will unlink a device from a buffer, no warning is given of the detach and the data that is currently stored within the buffer is purged.

This call should only be used by the actual device that called Buffer_LinkDevice, anyone else calling this SWI could confuse the system.

Related SWIs

Buffer_LinkDevice (SWI &42945)

Related vectors

None

Buffer_GetInfo (SWI &42947)

Returns data about the buffer

On entry

R0 = buffer handle

On exit

V = 0 ⇒ R0 = flags relating to buffer
R1 = pointer to start of buffer in memory
R2 = pointer to end of buffer in memory (+1)
R3 = insert index for buffer
R4 = remove index for buffer
R5 = remaining free space in buffer
R6 = number of characters in buffer
V = 1⇒ R0 = pointer to error block

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI returns data about the buffer, its position in memory, flags, insert and remove offsets, and the amount of free space.

Related SWIs

None

Related vectors

None

Buffer_Threshold (SWI &42948)

Sets or reads the warning threshold of the buffer

On entry

R0 = buffer handle
R1 = threshold (0 = none, –1 ⇒ to read)

On exit

R1 = previous value

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI is used to set or read the warning threshold of the buffer. This is used to trigger UpCalls if bit 3 of the buffer flags is set.

The Upcalls are issued when the amount of free space in the buffer crosses the threshold value (see the chapter entitled *Communications within* RISC OS on page 1-167).

Related SWIs

None

Related vectors

None

64 Squash

Introduction and Overview

This module provides general compression and decompression facilities of a lossless nature through a SWI interface. The algorithm is 12-bit LZW, however, this may change in future releases.

The interface is designed to be restartable, so that compression or decompression can occur from a variety of locations. Operations involving file IO can easily be constructed from the operations provided.

Errors

The following errors can be returned by the Squash module:

Error number	Error text
&920	SWI value out of range for module Squash
&921	Bad address for module Squash
&922	Bad input for module Squash
&923	Bad workspace for module Squash
&924	Bad parameters for module Squash

SWI calls

Squash_Compress (SWI &42700)

Provides general compression of a lossless nature

On entry

R0 = flags:
- bit 0: 0 = start new operation; 1 = continue existing operation (using existing workspace contents)
- bit 1: 0 = end of the input; 1 = more input after this
- bit 2: reserved (must be zero)
- bit 3: 0 = no effect; 1 = return the work space size required and the maximum output size in bytes (all other bits must be 0)
- bits 4 - 31 reserved (must be zero)

R1 = input size (−1 ⟹ do not return maximum output size) – if bit 3 of R0 is set; else = workspace pointer – if bit 3 of R0 is clear
R2 = input pointer – if bit 3 of R0 is clear
R3 = number of bytes of input available – if bit 3 of R0 is clear
R4 = output pointer – if bit 3 of R0 is clear
R5 = number of bytes of output space available – if bit 3 of R0 is clear

On exit

R0 = required work space size – if bit 3 of R0 set on input; else = output status – if bit 3 of R0 clear on input:
- 0 = operation completed
- 1 = operation ran out of input data (R3 = 0)
- 2 = operation ran out of output space (R5 < 12)

R1 = maximum output size (−1 ⟹ don't know or wasn't asked) – if bit 3 of R0 set on input; else preserved – if bit 3 of R0 clear on input
R2 = updated to show first unused input byte – if bit 3 of R0 clear on input
R3 = updated to show number of input bytes not used – if bit 3 of R0 clear on input
R4 = updated to show first unused output byte – if bit 3 of R0 clear on input
R5 = updated to show number of output bytes not used – if bit 3 of R0 clear on input

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 general compression of a lossless nature. It acts as a filter on a stream of data. The SWI returns if either the input or the output is exhausted.

It is recommended that you use the following facility to determine the maximum output size rather than attempting to calculate it yourself:

Call the SWI first with bit 3 of R0 set and the input size placed in R1. The maximum output size is then calculated and returned on exit in R1. You can use this value to allocate the required amount of space and call the SWI again setting the registers as appropriate.

If for any reason the SWI cannot calculate the maximum output size it will return −1 on exit in R1.

The workspace size required is returned on exit in R0.

The algorithm used by this module is 12-bit LZW, as used by the UNIX 'compress' command (with −b 12 specified). If future versions of the module use different algorithms, they will still be able to decompress existing compressed data.

If bits 0 and 1 of R0 are clear, and the output is definitely big enough, a fast algorithm will be used.

The performance of compression on an 8Mhz A420 with ARM2 is approximately as follows:

Store to store	**Fast case**
24 Kbytes per second	68 Kbytes per second

where Fast case is store to store, with all input present and output buffer assumed large enough.

Related SWIs

Squash_Decompress (SWI &42701)

Related vectors

None

Squash_Decompress (SWI &42701)

Provides general decompression of a lossless nature

On entry

R0 = flags:
- bit 0 — 0 = start new operation
  1 = continue existing operation (using existing workspace contents)
- bit 1 — 0 = end of the input
  1 = more input after this
- bit 2 — 0 = normal
  1 = You may assume that the output will all fit in this buffer (allows a faster algorithm to be used, if bits 0 and 1 are both 0)
- bit 3 — 0 = no effect
  1 = return the work space size required and the maximum output size in bytes (all other bits must be 0)
- bits 4 - 31 reserved (must be zero)

R1 = input size (–1 ⟹ do not return maximum output size) – if bit 3 of R0 is set; else = workspace pointer – if bit 3 of R0 is clear
R2 = input pointer – if bit 3 of R0 is clear
R3 = number of bytes of input available – if bit 3 of R0 is clear
R4 = output pointer – if bit 3 of R0 is clear
R5 = number of bytes of output space available – if bit 3 of R0 is clear

On exit

R0 = required work space size – if bit 3 of R0 set on input; else
= output status – if bit 3 of R0 clear on input:
- 0 = operation completed
- 1 = operation ran out of input data (R3 < 12)
- 2 = operation ran out of output space (R5 = 0)

R1 = maximum output size (–1 ⟹ don't know or wasn't asked) – if bit 3 of R0 set on input; else preserved – if bit 3 of R0 clear on input
R2 = updated to show first unused input byte – if bit 3 of R0 clear on input
R3 = updated to show number of input bytes not used – if bit 3 of R0 clear on input
R4 = updated to show first unused output byte – if bit 3 of R0 clear on input
R5 = updated to show number of output bytes not used – if bit 3 of R0 clear on input

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 general decompression of a lossless nature.

Note: The current algorithm cannot predict what the size of decompressed output will be. This means that, currently, –1 is always returned on exit in R1. In future releases this may change, it is therefore recommended that you call the SWI first with bit 3 of R0 set and the input size placed in R1. If R1 is not equal to –1 then you can use this value to allocate the required amount of space and call the SWI again setting the registers as appropriate. If R1 is equal to –1 you must attempt to calculate the maximum output size yourself.

The workspace size required is returned on exit in R0.

In the case where R3 < 12, the unused input must be resupplied.

The performance of decompression on an 8Mhz A420 with ARM2 is approximately as follows:

Store to store	Fast case
48 Kbytes per second	280 Kbytes per second

where Fast case is store to store, with all input present and output buffer assumed large enough.

Related SWIs

Squash_Compress (SWI &42700)

Related vectors

None

65 ScreenBlank

Introduction and Overview

This module is not available in RISC OS 2.

Service Calls

Service_ScreenBlanked (Service Call &7A)

Screen blanked by screen blanker

On entry

R1 = &7A (reason code)

On exit

All registers must be preserved.

Use

This service call is issued by the screen blanker, after the screen has been blanked. This service call should not be claimed.

Service_ScreenRestored (Service Call &7B)

Screen restored by screen blanker

On entry

R1 = &7B (reason code)

On exit

All registers must be preserved.

Use

This service call is issued by the screen blanker, after the screen has been restored. This service call should not be claimed.

* Commands

*BlankTime

Sets the time of inactivity before the screen blanks

Syntax

`*BlankTime [W|O] [time]`

Parameters

`W`	writing to the screen finishes screen blanking
`O`	writing to the screen does not finish screen blanking
`time`	time of inactivity before the screen blanks

Use

*BlankTime sets the time in seconds before the screen blanks. If, during this time, there is no activity (ie no keyboard or mouse input is received, and – with the `W` option – there is no writing to the screen) the screen then blanks. This saves 'burn in' on the phosphor of your monitor, which occurs when the monitor consistently displays a particular image, such as the desktop.

Screen blanking finishes as soon as there is activity (see above).

If no option is specified, `O` is assumed.

Example

`*BlankTime W 600` *blanks the screen if neither input nor output occur for 10 minutes*

Related commands

None

Related SWIs

None

Related vectors

WrchV (claimed by `W` option)