Part 7 – The desktop

53 The Window Manager

Introduction

This chapter describes the Window Manager. It provides the facilities you need to write applications that work in the Desktop windowing environment that RISC OS provides.

The Window Manager is an important part of RISC OS because:

- it provides a simple to use graphical interface, that makes your applications more accessible to a wider range of users
- it also provides the means for you to make your applications run in a multi-tasking environment, so they can interact with each other, and with other software.

This chapter also gives guidelines on how your applications should behave so that they are consistent with other RISC OS applications. This should make it easier for users to learn how to use your software, as they will already be familiar with the necessary techniques.

You will find it benefits both you and other programmers if you make all your applications run under the Window Manager (and in a consistent manner), since this will lead to a much richer RISC OS environment.

Overview

The Window Manager is designed to simplify the task of producing programs to run under a WIMP (Windows, Icons, Menus and Pointer) environment. The manager itself is usually referred to as the Wimp. Programs that run under the Wimp are often called tasks, because they are operating under a multi-tasking environment. In this section, the words task, program and application should be treated as synonyms.

An immediately recognisable feature of Wimp programs is their use of overlapping rectangular windows on the screen. These are used to implement a ‘desktop’ metaphor, where the windows represent documents on a desk. The responsibility of drawing and maintaining these windows is shared between the application(s) and the Window Manager.

The Wimp co-operates with the task in keeping the screen display correct by telling the task when something needs to be redrawn. Thus, the task needs to make as few intelligent decisions as possible. It merely has to respond appropriately to the messages it receives from the Wimp, in addition to performing its own processing (using the routines supplied to perform window operations).

Very often, much of the work of keeping a window’s contents up to date can be delegated to the Wimp. This is especially true if a program takes advantage of icons. An icon is a rectangular area in a window whose contents can be text, a sprite, both, or user-drawn graphics. In the first three cases, the Wimp can maintain the icon automatically, even to the point of performing text input without the application’s intervention.

Menus also form an important part of WIMP-based programs. RISC OS Wimp menus are pop-up. That is, they can be made to appear when the user clicks on the appropriate mouse button – the middle Menu button. This is an alternative to the menu bar approach, where an area of the screen is dedicated to providing a fixed set of menu headers. In a multi-tasking environment, pop-up menus are much more useable. Further, they can be context-sensitive, i.e. the menu that pops up is appropriate to the mouse pointer position when the Menu button was pressed.

The Wimp provides support for nested menus, where one menu entry can lead to another menu, to any desired depth. Moreover, the ‘leaf’ of a menu structure can be a general window, not just a fixed text item. This allows for very flexible selections to be made from menus.

A very powerful feature of the RISC OS Wimp is its support for co-operative multi-tasking. Several programs can be active at once. They gain control on return from the Wimp’s polling routine, which is described below. There is normally no pre-emption. Pre-emption means the removal of control from a task at arbitrary times, without its prior knowledge. With polling, a task only relinquishes control when it chooses, so for the system to work, tasks must be well behaved. This means they must

not spend too much time between polling, otherwise other tasks will be prevented from running. However, it is possible to enforce pre-emption for non-Wimp tasks, by running them in for example, the edit application's task window.

To allow several applications to run at once, the Wimp must also perform memory management. This allows each application to 'see' a standard address space starting at &8000 whenever it has control. As far as a task is concerned, it is the only user of the application workspace. The amount of workspace that a task has is settable before it starts up. A program does not therefore have to be written with multi-tasking in mind. A task that does everything correctly will work whether it is the only program running, or one of several.

Communication between tasks is possible. In fact, it is often necessary, as the Task Manager sometimes needs to 'talk' to the programs it is controlling. The Wimp implements a general and very powerful message-passing scheme. Messages are used to inform tasks of such events as screen mode and palette changes, and to implement a general purpose file transfer facility.

The next section gives an overview of the major components of the RISC OS Window Manager.

Technical details

Polling

Central to any program running under the Wimp environment is its polling loop. Wimp programs are event driven. This means that instead of the program directing the user through various steps, the program waits for the user to control it. It responds to events. An event is a message sent to a task by the Wimp, or by another task. Events are usually generated in response to the user performing some action, such as clicking a mouse button, moving the pointer, selecting a menu item, etc. Inter-task (‘user’) messages are also passed through the polling loop.

An application calls the routine Wimp_Poll (page 3-112) to find out which events, if any, are pending for it. This routine returns a number giving the event type, and some event-specific information in a parameter block supplied by the caller. One event is Null_Reason_Code (0), which means nothing in particular needs to be done. The program can use this event to perform any background processing.

In very broad terms, Wimp applications have the following (simplified) structure:

```
SYS"Wimp_Initialise"                    Tell the Wimp about the task
finished = FALSE : DIM blk 255          Get block for Wimp_Poll
REPEAT
  SYS"Wimp_Poll",0,blk TO eventCode     Get the event code to process
  CASE eventCode OF
    WHEN 0:...                          Do Null_Reason_Code
    WHEN 1:...                          Do Redraw_Window_Request
    ...                                 etc.
  ENDCASE
UNTIL finished
SYS"Wimp_CloseDown"                     Tell Wimp we've finished
```

Currently, event codes in the range 0 to 19 are returned, though not all of these are used. A fully specified Wimp program will have `WHEN` (or equivalent) routines to deal with most of them.

Some of the event types are fairly esoteric and can be ignored by many programs. It is very important that tasks do not complain about unrecognised event codes; they should simply ignore them.

Better still is to avoid receiving them in the first place. When calling Wimp_Poll, the program can mask out certain events if it does not want to hear about them at the moment. For example, if the program doesn't need to know about the pointer leaving or entering a window, it could mask out these events. This makes the whole system more efficient, as the Wimp will not bother to pass control to a task which will simply ignore the event. Some events are unmaskable; for example, an application must respond to Open_Window_Request.

As noted above, events are usually generated internally by the Wimp. However, a user task may also send messages, which result in Wimp_Poll events being generated at the destination task. For example, the Madness application moves all of the windows around the screen by sending an Open_Window_Request message to their owners. A more useful use of messages is the data transfer protocol. Most messages sent between tasks are of type User_Message_*xxx* (17, 18 and 19). For details of these see the documentation of Wimp_SendMessage on page 3-193, and the section entitled *Wimp messages* on page 3-237.

Null events

If you don't really need Null_Reason_Code events, you should mask them out when you call Wimp_Poll. This avoids the Wimp passing control to your application, only for your application to immediately return control to the Wimp by calling Wimp_Poll again; this of course would slow the system down. If you do need to take null events you should use Wimp_PollIdle rather than Wimp_Poll, unless the user is directly involved (e.g. when dragging an object) and responsiveness is important.

All of the event types are described in the section entitled *Wimp_Poll (SWI &400C7)* on page 3-112, along with descriptions of how the application should respond to them.

General principles

Much of what is said below is to do with consistency and standards. Providing the user with a consistent, reliable interface is the first step towards producing a powerful environment, and one that the user will want to work with instead of just being forced to. For a full description of the general principles you should adopt in writing an application to run under the Wimp see the chapter entitled *General principles* in the *RISC OS Style Guide*.

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

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

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

Compatibility

The following points should be noted, to ensure that your application is compatible with future versions of the Wimp and behaves as well as it can with old versions of the Wimp.

- Reserved fields must be set to 0, i.e. reserved words must be 0, and all reserved bits unset.
- Unknown Wimp_Poll event codes, message actions etc must be ignored - do not generate errors.
- Applications should check Wimp version number, and either adapt themselves if the Wimp is too old, or report an error to the current error handler (using OS_GenerateError).
- Beware of giving errors if window handles are unrecognised as they may belong to another task and it is sometimes legal for their window handles to be returned to you (e.g. by Wimp_GetPointerInfo).

- Wimp tasks which are modules must obey certain rules (see the section entitled *Relocatable module tasks* on page 3-60).
- Tasks that can receive Key_Pressed events must pass on all unrecognised keys to Wimp_ProcessKey. Failure to do so will result in the 'hot key' facilities not working.

Responsiveness

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

Colour

Covering a wide range of screen modes can seem troublesome when constructing an application, but it allows a wide price-range for the end user, who can choose between resolution and cost. Not relying on screen size allows your program to move easily to new better screens and modes when they become available.

Terminology

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

The Mouse

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

Select and Adjust

Always use Select as the 'primary' button of the mouse, used for pointing at things, dragging etc. Adjust is used for less common or less obvious functions, or for slight variations and speedups. If you have no useful separate operation in any particular context, then make Adjust do nothing rather than duplicating the functionality of Select: this is all part of training the user to use Select first.

Another technique for speedups and variations on mouse operations is to look at the setting of the Shift key when the mouse event occurs. Such combinations should never be necessary to the operation of a program, for example, a user experimenting with your program should not be expected to try all such combinations.

Double clicks

The Wimp automatically detects double clicks, typically used to mean ‘open object’. It should be noted that a double click causes a single click event to be sent to the program first. Some other systems avoid this, which may appear to simplify the task of programming but leads to reduced responsiveness to mouse operations (because the application doesn’t get to hear about the first click until the WIMP system is sure it’s not a double click). A double click should in any case be thought of as a consolidation of a single click.

Various parts of the Wimp enforce the interpretations given for the mouse buttons in the Style Guide. For example, icons may be programmed to respond in various ways to clicks with the Adjust and Select buttons, by setting their button type. On the other hand, a click on the Menu button is always reported in exactly the same way, regardless of where it occurs, as a Mouse_Click event with the button state set to 2. This is to encourage all programs to interpret a click on the middle button in the same way – as a request to open a menu.

Layout of windows

Coordinate system

Windows consist of a visible area, in which the task can draw graphics, and a surrounding ‘system’ area, comprising a Title Bar, scroll bar indicators and so on. The task does not normally draw directly in this area, except the Title Bar. The visible area provides a window into a larger region, called the work area. You can imagine the work area to be the complete document you are working with, and the visible area a window into this.

There are, therefore, two sets of coordinates to deal with when setting up a window. The visible area coordinates determine where the window will appear on the screen and its size. These are given in terms of OS graphics units, with the origin in its default position at the bottom left of the screen.

Then there are the work area coordinates. These give the minimum and maximum x and y coordinate of the whole document. The limits of the work area are sometimes called its extent. The work area is specified when a window is created, but can be altered using the Wimp_SetExtent (page 3-161) call.

Between the work area coordinates and the visible area coordinates is a final pair which join the two together. These are the scroll offsets. They indicate which part of the work area is shown by the visible area – this is called the visible work area.

The scroll offsets give the coordinates of the pixel in the work area which is displayed at the top lefthand corner of the visible region. Suppose the visible region shows the very top left of the work area. Then the x scroll position would be ‘work area x min’, and the y scroll position would be ‘work area y max’.

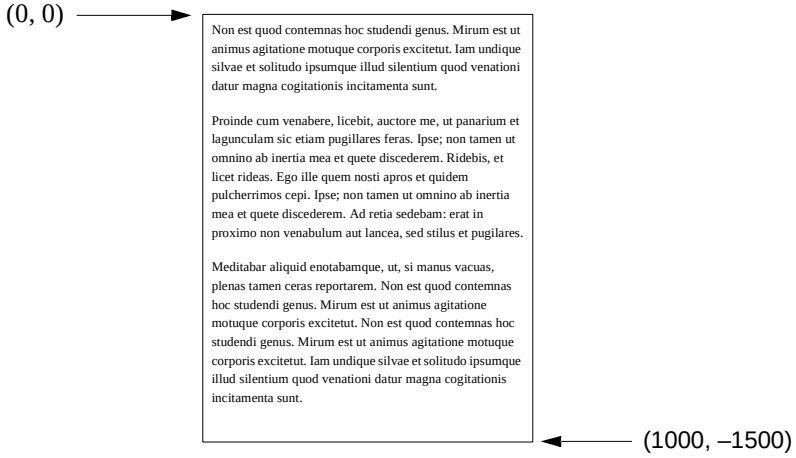
It is common to define the work area such that its origin (0,0) is at the top left of the document. This means that all x scroll offsets are positive (as you can only ever be on or to the right of the work area origin), and all y offsets are zero or negative (as you can only ever be on or below the work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

To summarise, let’s consider which part of the work area will be visible, and where it will appear on the screen, for a typical set of coordinates.

Work area

The following definitions give the total document size:

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

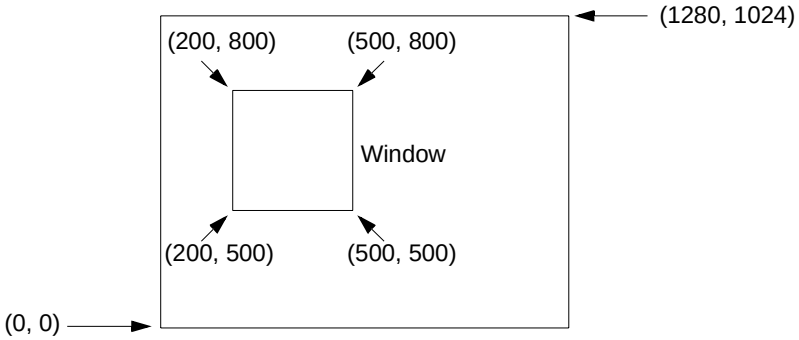


The document is therefore 1000 units wide by 1500 high, with the work area origin at the top left of the document.

Window area

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

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



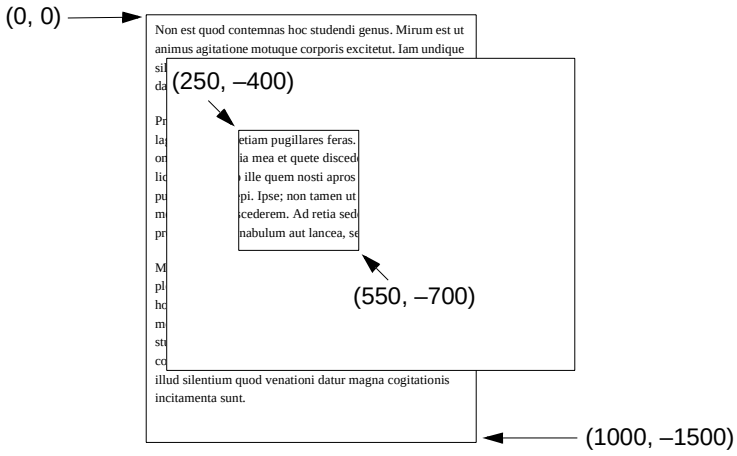
This gives a window 300 units wide by 300 high.

Work area displayed

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

scroll_offset_x = 250
scroll_offset_y = –400

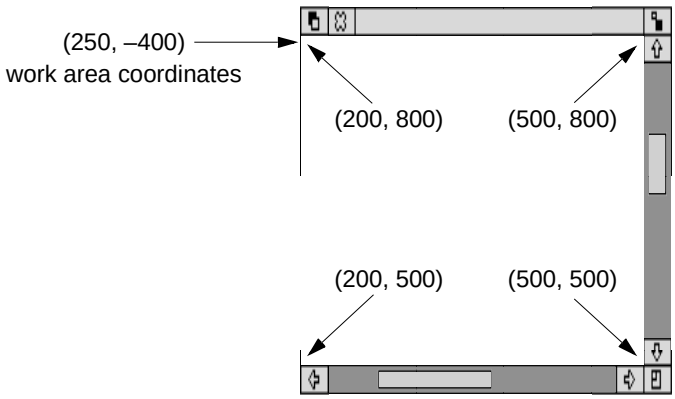
Thus the pixel at the top left of the window is shown on the screen at coordinates (200,800), but represents the point (250,–400) in the work area:



Combining the above bits of information, we can work out what portion of the work area is visible. By definition, the minimum x coordinate and the maximum y coordinate of the visible work area are just the scroll offsets. The maximum x and minimum y can then be derived by adding the width and subtracting the height respectively of the displayed window:

visible_work_area_min_x = scroll_offset_x = 250
visible_work_area_max_y = scroll_offset_y = –400
visible_work_area_max_x = scroll_offset_x + width = 550
visible_work_area_min_y = scroll_offset_y – height = –700

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



drawn by the system have a length proportional to the area that the window displays. The horizontal Slider would therefore occupy about 300/1000 = 0.3 of the horizontal scroll bar, and the vertical one would occupy 300/1500 = 0.2 of the scroll bar.

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

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

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

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

work_area_pixel_at_origin_x = scroll_offset_x – visible_area_min_x
work_area_pixel_at_origin_y = scroll_offset_x – visible_area_max_y

2 Add this to the given screen coordinates.

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

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

Thus the entire formula would be:

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

Generally, when this calculation is needed, the scroll offsets and visible work area coordinates are available (e.g. having been returned from Wimp_Poll). Even if they are not, a call to Wimp_GetWindowState (page 3-132) will secure the information.

In addition to the coordinates described above, several other attributes have to be set when a window is created. These are described in detail in the entry on Wimp_CreateWindow (page 3-87).

Window stacks

Windows can overlap on the screen. In order to determine which windows obscure which, the Wimp maintains ‘depth’ as well as positional information. We say that there is a window stack. The window at the top of the stack obscures all others that occupy the same space on the screen; the one on the bottom of the stack is obscured by any other at the same coordinates.

Certain mouse operations alter a window’s depth in the stack. A click with Select on the Title Bar (see below) brings the window to the top. Similarly you can give a window a Back icon, which, when clicked on, will send the window to the bottom of the stack. On opening a window, you can determine its depth in the stack by specifying the window that it must appear behind. Alternatively you can give its depth absolutely as ‘top’ or ‘bottom’.

Window flags

One 32-bit word of the window block contains flags. These control many of its attributes: which control icons it should have, whether it’s movable, whether Scroll_Request events should be generated etc. Another word of flags control the appearance of the Title Bar, and yet another word set the button type of the work area. Both of these are actually icon attributes, the Title Bar being treated like an icon in many ways.

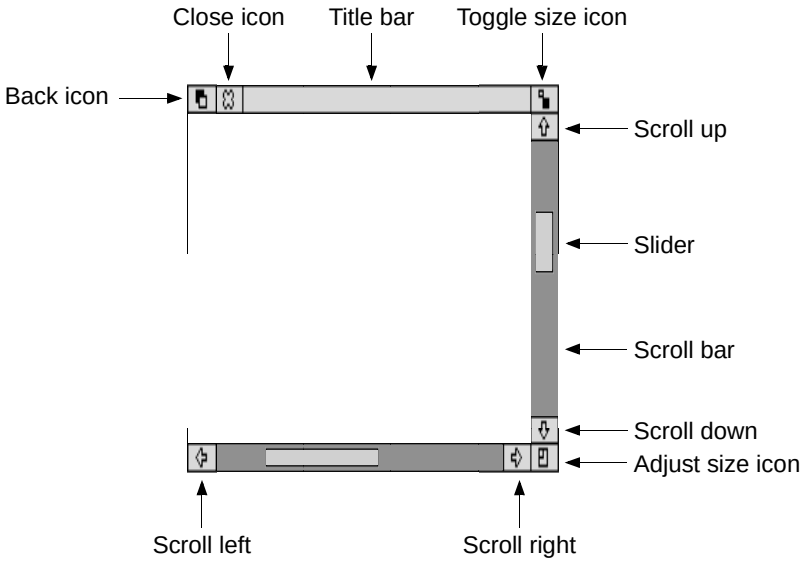
Finally there are miscellaneous properties such as the sprite area address to use for icon sprites, the minimum size of the window, and the icon data for the Title Bar.

Appended to the window definition are any initial icons that it owns. Further icons can be added using the call Wimp_CreateIcon (page 3-93) on page 3-93.

Window system areas

For full details on how windows must behave on the RISC OS desktop see the chapters entitled *Windows* and *Editors* in the *RISC OS Style Guide*.

The window illustrated below has a fully defined system area showing all of the available controls. The control areas, going clockwise from the top-left corner, are described below. Where the effects of using Select and Adjust on them are different, this is noted.



Back icon

A click on this icon causes the window to be moved to the back of the window stack, making it the ‘least visible’ one. A Redraw_Window_Request event is issued to any applications which have windows that were obscured by it and are now visible.

Close icon

A click on this icon requests that a window be closed; the Wimp generates a Close_Window_Request event. It is then up to the application whether it responds with a Wimp_CloseWindow (page 3-111) call, or ignores the event if it has good reason not to, such as unsaved data. Using Adjust should open the ‘parent window’, if such a thing

exists. For example, the Filer closes a directory display, but opens its parent directory; an editor opens the home directory for the loaded document. When a window is closed, the Wimp issues Redraw_Window_Requests to those windows which were obscured by it and are now visible.

Title bar

This contains the name of the window, which is set when the window is created. Dragging the Title Bar causes the whole window to be dragged. If Select is used for the drag, the window is also brought to the top; Adjust leaves it at the same depth. The Title Bar has many of the attributes of an icon (font type, indirection, centring etc). If the whole window is being dragged (and not just its outline), each movement will generate an Open_Window_Request for it, and Redraw_Window_Requests to windows that become unobscured.

Toggle Size icon

A click in this icon toggles the window between its maximum size and the last user-set size. An Open_Window_Request event is generated to ask the application to update the work region of the resized window. The maximum size of a window depends on its work area extent and the size of the screen. Again, using Select uncovers the window; Adjust leaves it at the same depth in the stack. As usual, if the change in window size renders previously obscured window visible, Redraw_Window_Requests will be generated for them. When the window is toggled back to its small size, it goes back to its previous depth in the stack.

Vertical scroll bar

Although this is one object as far as the window definition is concerned, there are five regions within it. They are:

- the scroll up arrow
- the page up area (above the Slider)
- the Slider
- the page down area (below the Slider)
- the scroll down arrow.

If the user clicks on one of the arrows with Select, the scroll offset for the window is adjusted by 32 units in the appropriate direction. Using Adjust scrolls in the reverse direction. Holding down either button causes the scrolling to auto-repeat. A click in the page up/down region adjusts the scroll offsets by the height of the window work area, with Adjust again giving the reverse effect from Select. An Open_Window_Request is generated to update the scrolled window.

If the window had one of the Scroll_Request flags set when it was created, a click in one of the arrows or page up/down areas causes a Scroll_Request event to be generated instead. The application can decide how much to scroll and call Wimp_OpenWindow (page 3-109) to update its contents.

Finally, the Slider may be dragged to set the scroll offsets to any position in the work area. The Open_Window_Request events are returned either continuously or when the drag finishes, depending on the state of the Wimp drag configuration bits.

All scroll operations leave the window's depth unaltered.

Adjust Size icon

Dragging on this icon causes the window to be resized. The limits of the new window size are determined by the work area extent and the minimum size given when the window was created. Depending on the state of the Wimp drag configuration flags the Wimp generates either continuous Open_Window_Requests (and possibly Redraw_Window_Requests for other windows) or a single one at the end of the drag. Select brings the window to the top; Adjust leaves it at the same depth.

Horizontal scroll bar

This is exactly equivalent to the vertical scroll bar described above. For 'up' read 'right' and for 'down' read 'left', i.e. whereas scroll up increases the y scroll offset, scroll right increases the x scroll offset. The five regions within it are:

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

When a window is created, its control regions can be defined in one of two ways. The 'old' way is to use certain flags which specify in a limited fashion which of the regions should be present and which are omitted. The 'new' method uses one flag per control, and is much easier to use. The old way was used in Arthur, while the new is only available in RISC OS.

Redrawing windows

The Wimp and the application must cooperate to ensure that the windows on the screen remain up to date. The Wimp can't do all of the work, as it does not always know what the contents of a window should be.

When the task receives the event code Redraw_Window_Request from Wimp_Poll, it should enter a loop of the following form:

```
REM blk is the Wimp_Poll block
SYS"Wimp_RedrawWindow",,blk TO flag
WHILE flag
        Redraw contents of the appropriate window
        SYS"Wimp_GetRectangle",,blk TO flag
ENDWHILE
Return to polling loop
```

When a window has to be redrawn, often only part of it needs to be updated. The Wimp splits this area into a series of non-overlapping rectangles. The rectangles are returned as x0,y0,x1,y1 where (x0,y0) is inclusive and (x1,y1) is exclusive. This applies to all boxes, e.g. icons, work area, etc. The `WHILE` loop above is used to obtain all the rectangles so that they can be redrawn. The Wimp automatically sets the graphics clipping window to the rectangle to be redrawn. The task can take a simplistic view, and redraw its whole window contents each time round the loop, relying on the graphics window to clip the unwanted parts out. Alternatively, and much more efficiently, it can inspect the graphics window coordinates (which are returned by Wimp_RedrawWindow (page 3-126) and Wimp_GetRectangle (page 3-130)) and only draw the contents of that particular region.

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

The areas to be redrawn are automatically cleared (to the window's background colour) by the Wimp. The task must determine what part of the workspace area is to be redrawn using the visible area coordinates and the current scroll offsets.

When redrawing a window's contents, you should normally use the overwrite `GCOL` action. You should use EOR mode when redrawing any currently dragged object. EOR mode is also useful when updating the window contents, such as dragging lines in Draw. As a rule, the contents of the document should not use EOR mode.

You should not use block operations such as Wimp_BlockCopy (page 3-201) within the redraw or update loop, only outside it to move an area of workspace. These restrictions allow you to use the same code to draw the window contents and to print the document. If you use, for example, exclusive-OR plotting or block moves during the redraw these won't work on, say, a PostScript printer driver.

Updating windows

When a task wants to update a window's contents, it must not simply update the appropriate area of the screen. This is because the task does not know which other windows overlap the one to be updated, so it could overwrite their contents. As with all window operations, it must be done with the Wimp's co-operation. There are two possible approaches. The program can:

- call Wimp_ForceRedraw (page 3-147) so Wimp subsequently returns a Redraw_Window_Request, or
- call Wimp_UpdateWindow, and perform appropriate operations.

In both cases, you provide the window handle and the coordinates of the rectangular area of the work area to be updated. The Wimp works out which areas of this rectangle are visible, and marks them as invalid. If you use the first method, the Wimp will subsequently return a Redraw_Window_Request from Wimp_Poll, which you should respond to as already described. In the second case, a list of rectangles to be redrawn is returned immediately.

When Wimp_ForceRedraw is used, the Wimp clears the update area automatically. This should therefore be used when a permanent change has occurred in the window's contents, e.g. a paragraph has been reformatted in an editor. When you call Wimp_UpdateWindow (page 3-128), no such clearing takes place. This makes this call more suitable for temporary changes to the window, for example, when dragging objects or 'rubber-banding' in graphics programs.

It is simpler to use Wimp_ForceRedraw since, once it has been called, the task just returns to the central loop, from where the Redraw_Window_Request will be received. The code to handle this must already be present for the program to work at all. On the other hand, the second method is much quicker as the redrawing is performed immediately. Also, you can keep the original contents, using EOR to update part of the rectangle; for example, when dragging a line.

Taking over the screen

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

The icon bar

The Window Manager provides an icon bar facility to allow tasks to register icons in a central place. It appears as a thick bar at the bottom of the screen, containing filing system and device icons on the left, and application icons on the right.

When an application is loaded, it registers an icon on the icon bar using Wimp_CreateIcon with window handle = –1 (or –2 for devices). The icon is typically the same as the one used to represent the application directory within the Filer, i.e. !Appl.

If there are so many icons on the icon bar that it fills up, the Wimp will automatically scroll the bar whenever the mouse pointer is moved close to either end of the bar.

When the mouse is clicked on one of the icons, the Wimp returns the Mouse_Click event (with window handle = –2) to the task which created the icon originally. Similarly, Wimp_GetPointerInfo returns –2 for the window handle when the pointer is over (either part of) the icon bar.

Icon bar dimensions

When Wimp_CreateIcon is called to put an icon on the bar, the Wimp uses the x coordinates of the icon only to determine its width, and then horizontally positions the icon as it sees fit. However, for reasons of flexibility, it does not vertically centre the icon, but actually uses both the y coordinates given to determine the icon's position. This means that applications must be aware of the 'standard' dimensions of the bar, in order to position their icons correctly.

Icons that appear on the icon bar should have bounding boxes 68 OS units square.

Positioning icons on the Icon bar

There are two main types of icon which are put onto the icon bar: those consisting simply of a sprite, and those consisting of a sprite with text written underneath (see Wimp_CreateIcon on page 3-93 for details).

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

Icons and sprites

As mentioned earlier, an icon is a rectangular area of a window's workspace. Icons can be created at the same time as a window, by appending their definitions to a window block. Alternatively, you can create new icons as needed by calling Wimp_CreateIcon.

A third possibility is to plot ‘virtual’ icons during a redraw or update loop using Wimp_PlotIcon (page 3-183). The advantage of this last technique is that the icons plotted don’t occupy permanent storage.

Icons have handles that are unique within their parent window. Thus an icon is totally defined by a window/icon handle pair. User icon handles start from zero; the system areas of windows have negative icon numbers when returned by Wimp_GetPointerInfo (page 3-140).

The contents of an icon can be anything that the programmer desires. The Wimp provides a lot of help with this. It will perform automatic redrawing of icons whose contents are text strings, sprites, or both. Moreover, text icons can be writable, that is, the Wimp will deal with user input to the icon, and also handle certain editing functions such as Delete and left and right cursor movements.

Below is an overview of the information supplied when the program defines an icon. For a detailed description, see Wimp_CreateIcon (page 3-93).

Bounding box

Four coordinates define the rectangle that the icon occupies in the window’s workspace. The Wimp uses this region when detecting mouse clicks or movements over the icon, when filling the icon background (if any) and drawing the icon border (if any).

Icon flags

This single word contains much of the information that make icon handling so flexible. It indicates:

- whether the icon contains text, a sprite, or both
- for text icons, the text colours, whether the font is anti-aliased or not (and the font handle), and the alignment of text within the font bounding box
- for sprite icons, whether to draw the icon half size
- whether the icon has a border and/or a filled background
- whether the application has to help redraw the icon’s contents
- whether the icon is indirected
- the button type of the icon
- the exclusive selection group (ESG) of the icon, and how to handle Adjust-type selections of this icon
- whether to shade the icon so that it can’t be selected.

Indirected icons use the last twelve bytes of the icon definition in a different way from non-indirected ones; see below.

The button type of an icon determines how the Wimp will deal with mouse movements and clicks over the icon. There are 16 possible types. Examples are: ignore all movements/clicks; report single clicks, double clicks and drags; select the icon on a single click; make the icon writable, and so on.

When Select is used to select an icon, its selected bit is set regardless of its previous state, and it is highlighted. When Adjust is used, its selected bit is toggled, de-selecting it if it was previously highlighted, and vice versa.

When an icon is selected, the Wimp indicates this visually by inverting the colours that are used to draw its text and/or sprite. Selecting an icon causes all other icons in its exclusive selection group to be de-selected. The ESG is in the range 0 to 31. Zero is special; this puts the icon in a group of its own, so selecting the icon will not affect any other icons, but each selection actually toggles its state.

Imagine a window has three icons with ESG=1. Only one of these can be selected at once: the selection (or toggling by Adjust) of one automatically cancels the other two. However, if the icon has its adjust bit set, then using Adjust to toggle the icon's state will not have any affect on the other icons in the same ESG.

When the icon's shaded bit is set, the Wimp draws the icon in a 'subdued' way, to indicate that it can't be selected. This also prevents selection by clicking.

Icon flags occur in other contexts. A window definition uses the button type bits to determine its work area's button type. The rest of the bits (with some restrictions) are used to determine the appearance of a window's Title Bar. Finally menu items have icon flags to determine their appearance.

Icon data

The last 12 bytes of an icon definition are used in two different ways. If the icon is not indirected, these are used to hold a 12 byte text string. This is the text to be displayed for a text icon, the name of the sprite for a sprite icon, and both of these things for a text and sprite icon. Clearly the last is not very useful; it is unlikely that you will want to display an icon called `sm!arcpaint` along with the text `sm!arcpaint`.

If the icon button type is writable, clicking on the icon will position the caret at the nearest character and you can type into the icon, modifying the 12 byte text.

Indirected icons overcome the limitations of standard icons. Text can be more than 12 bytes long; the sprite in a text plus sprite icon can have a different name from the text displayed; sprite-only indirected icons can have a different sprite area pointer from their

window; writable icons can have validation strings defining the acceptable characters, and anti-aliased text can have colours other than the default white foreground/black background.

The twelve data bytes of an indirected icon are interpreted as three words: a pointer to the icon text or icon sprite, a pointer to the validation string or sprite control block, and the maximum length of the icon text.

Update of writable icons

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

The general routine goes as follows:

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

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

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

Deleting and creating icons

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

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

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

Thus to delete an icon, use:

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

and to re-create it, use:

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

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

Icon sprites

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

The sprites that are used in icons can come from any source: the system sprite pool, the Wimp sprite pool, or a totally independent user area. The use of the system sprites is not recommended as certain operations (such as scaling and colour translation) can't be performed on them (see the section entitled *Use of sprite pools* in the *Sprites and icons* chapter in the *RISC OS Style Guide* for more details). Wimp sprites are useful for obtaining standard shapes without duplicating them for each application. User sprites are used when private sprites are required that aren't available in the Wimp sprite area.

The Wimp sprite area is accessed by specifying a sprite area control block pointer of +1 in a window definition or indirected icon data word. There are actually two parts to the area, a permanent part held in ROM, and a transient, expandable area held in the RMA. The call Wimp_SpriteOp (page 3-198) allows automatic access to Wimp sprites by name. This is read-only access. The only operation allowed on Wimp sprites that changes them is the MergeSpriteFile reason code (11), or the equivalent `*IconSprites` command. These add further sprites to the Wimp area, expanding the RMA if necessary.

Below is a BASIC program to save the ROM sprites to a file. You can then use Paint to examine the sprites it contains.

```
SYS "Wimp_BaseOfSprites" TO rom
SYS "OS_SpriteOp",&10C,rom,"WSprites"
```

Amongst the ROM-based sprites are standard file-type icons (and half size versions of most of them), standard icon bar devices (printers, disk drives etc), common button types (radio buttons, option buttons) and the default pointer shape.

RISC OS System Icons

RISC OS 3 provides the following facilities for icons in addition to those provided in RISC OS 2:

- improved colour support
- window toolkit icons
- alternate resolution icons for applications.

Colour support

From RISC OS 3 onwards, the Wimp uses ColourTrans when preparing sprites for plotting (such as icons); so the palette associated with a sprite defines how its logical colours are mapped to the available physical colours. It also provides support for 8-bit-per-pixel sprites.

Window Icons

RISC OS 3 draws the top, right and bottom bars of the window from icons to allow customisation in the future. A complete window icon set contains 176 icons, which consists of 4 custom sets:

- one for modes 12/15 (*nx*=2, *ny*=4, *bpp*=2,4,8)
- one for mode 0 (*nx*=2, *ny*=4, *bpp*=1)
- one for VGA/SuperVGA (*nx*=2, *ny*=2, *bpp*=2,4,8)
- one for high-resolution monochrome (*nx*=2, *ny*=2, *bpp*=1)

The sets have equivalent designs, rendered as well as possible given the limitations of the various modes. (Note that RISC OS 2 effectively draws different things for 1, 2, 4/8 *bpp*, and for *nx*=2 or 4 and *ny*=2 or 4, and thus has 12 different behaviours; the 4 sets allow most of the main differences to be accommodated, but there will inevitably be slight differences for the 8 behaviours not directly supported.)

The icons are called *xx*, *xx*0, *xx*22 and *xx*23 following the Alternate Resolution Icon methodology for names (see page 3-31). RISC OS 3 displays different icons for 'pressed' icons on the window border. These icons are prefixed by 'p'. The 'p' form of an icon has to repaint over its unpressed form (and vice versa). If a 'p' form is not present, the corresponding unpressed icon is used.

There are 44 distinct designs (176/4) in the complete set. Many of the designs have defaults: all 'p' icons default to the unpressed icon. In addition, the title bar set, right scroll well and bottom scroll well will draw as RISC OS 2 if not present. The minimal set thus contains only the definitions of the corner icons: 10 designs (44 icons).

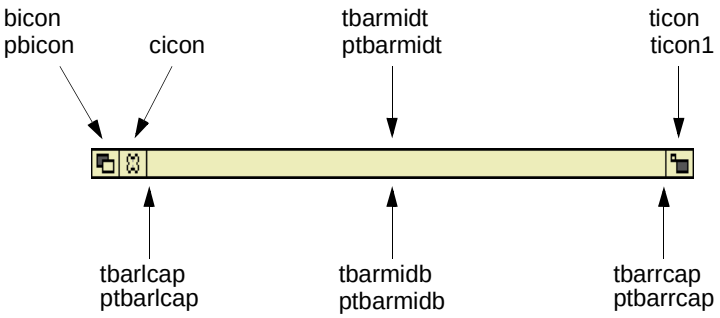
RISC OS 3 gets the sizes of the title bar, vertical scroll bar and horizontal scroll bar by reading the sizes of particular icons. All the other icons lying in the bar have to be of compatible size. There is no requirement that the icons for the different modes have compatible sizes; indeed, RISC OS 2 has bars that are 1 pixel different in size between 24 (e.g. mode 12) and 22 (e.g. mode 27) modes. Nor is there a requirement that the three bars in a mode have the same size.

All icons lying over highlighted sections of the window border (cream when selected, grey when not) must have transparent sections so that the colour can be seen. In the case of the title bar, this is plotted as four sections (left, top, bottom and right) so that a large expanse of transparency is not required.

The top, right and bottom window edge (black line) are drawn by the icons. We strongly recommend that you draw the outer edge of the top, right and bottom bars as a black line too.

The icons are:

Top Bar



bicon/pbicon:	Back icon
cicon:	Close icon
tbarlcap/ptbarlcap:	left hand end cap of title bar
tbarmidt/ptbarmidt:	title bar middle top (replicated as necessary)
tbarmidb/ptbarmidb:	title bar middle bottom (replicated as necessary)
tbarrcap/ptbarrcap:	right hand end cap of title bar
ticon, ticon1:	the two states of the Toggle Size icon (there is no pushed state, since you don't get a chance to see it before the window resizes)

tbarmidt and tbarmidb have to be the same width, but can be different heights. The Window Manager will paint tbarmidt below the top of the title bar and tbarmidb at the bottom, leaving the space between transparent to allow the cream or grey background it paints to show through. All other top bar icons have to be the same height. ticon has to be the same width as the vertical scroll bar.

Icons are plotted in the order:

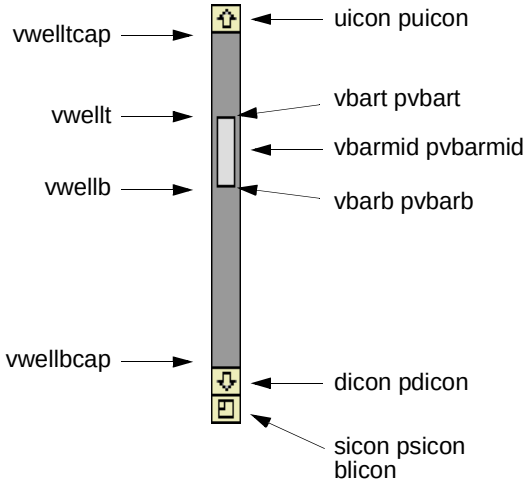
bicon, cicon,
tbarlcap, tbarmidb, tbarmidt, tbarrcap

such that the left pixel of the icon being painted overlaps the right pixel of the previously painted icon. (Left edges can be made transparent if this overlaid information has to be different). ticon is painted as a part of the right bar.

If tbarlcap is missing, the Window Manager paints the title section of the top bar using the RISC OS 2 style; otherwise it assumes that all necessary title bar icons are present.

Note that the top bar of menus is drawn with the same style as the title bar section of windows.

Right Bar



uicon/puicon:	up arrow
vwelltcap:	vertical scroll well top end cap
vwellt:	vertical scroll well top section (replicated as necessary)
vbart/pvbart:	vertical scroll bar top end cap
vbarmid/pvbarmid:	vertical scroll bar middle section (replicated as necessary)
vbarb/pvbarb:	vertical scroll bar bottom end cap
vwellb:	vertical scroll well bottom section (replicated as necessary)
vwellbcap:	vertical scroll well bottom end cap (note: if the vertical scroll well is to be transparent, this icon is the one checked for a mask)
dicon/pdicon:	down arrow
sicon/psicon:	Adjust Size icon
blicon:	blank icon used to replace the Adjust Size icon when it is not present

All these icons have to be the same width – as does ticon. sicon has to be the same height as the horizontal scroll bar.

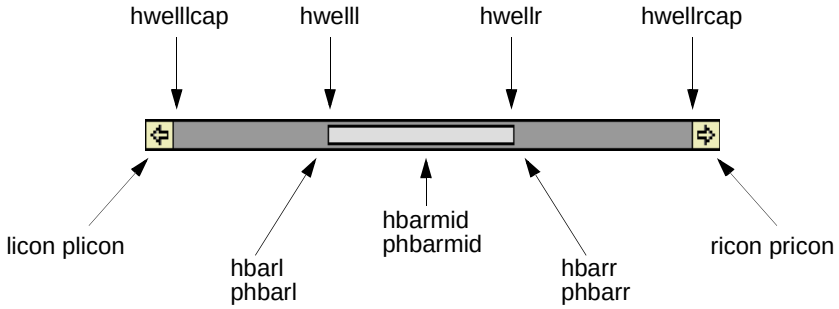
Icons are plotted in the order:

uicon, dicon,
vwellbcap, vwellb, vbarb, vbarmid, vbart, vwellt, vwelltcap,
ticon, sicon

such that the top pixel of the previous icon overlaps the bottom pixel of the current icon. (Top edges can be made transparent if this overlaid information has to be different).

If vwellbcap is missing, the Window Manager paints the scroll bar section of the right bar using the RISC OS 2 style; otherwise it assumes that all necessary scroll bar icons are present.

Bottom Bar



licon plicon:	left arrow
hwelllcap:	horizontal scroll well left end cap (note: if the horizontal scroll well is to be transparent, this icon is the one checked for a mask)
hwelll:	horizontal scroll well left section (replicated as necessary)
hbarl phbarl:	horizontal scroll bar left end cap
hbarmid phbarmid:	horizontal scroll bar middle section (replicated as necessary)
hbarr phbarr:	horizontal scroll bar right end cap
hwellr:	horizontal scroll well right section (replicated as necessary)
hwellrcap:	horizontal scroll well right end cap
ricon pricon:	right arrow

All these icons have to be the same height – as does sicon.

Icons are plotted in the order:

licon, ricon,
hwelllcap, hwelll, hbarl, hbarmid, hbarr, hwellr, hwellrcap

such that the right pixel of the previous icon overlaps the left pixel of the current icon. (Right edges can be made transparent if this overlaid information has to be different).

If hwelllcap is missing, the Window Manager paints the scroll bar section of the bottom bar using the RISC OS 2 style; otherwise it assumes that all necessary scroll bar icons are present.

Standard RISC OS 2 sizes and colours

A RISC OS 2 compatible set has these attributes:

	normal	0	22	23
icons	21×11 pixels	21×11 pixels	21×21 pixels	21×21 pixels
hwelllcap	no mask	mask	no mask	mask
vwellbcap	no mask	mask	no mask	mask

To be compatible with the RISC OS 2 scroll well colours requires some agility, since colours in icons do not get changed to dither patterns (as used in the one bit per pixel modes). Putting the dither pattern in the icon can be done in some cases – such as the scroll bar itself – but there may be a difference in the patterns alignment where the ends of the scroll well meet the well around the scroll bubble. Therefore transparent wells are used in these modes, and the Window Manager displays the dithered grey for the well interior. With solid colours this problem does not arise, and transparency masks in the icon would slow the system down, so the well colours come from the icons: this also has the benefit of improving the appearance of the scroll bar while it is being drawn.

Naturally, the minimal RISC OS 2 identical set would omit definition of the title bar and scroll wells entirely and rely on the Window Manager to draw these sections of the window outline.

Separate Borders

To get the edges of the icons not to touch (if, for example, they are 3D plinthed, and so left and right edges must be different) some pixel rows and columns need to be made transparent:

bicon:	right edge 1 pixel transparent
cicon:	right edge 1 pixel transparent
tbarlcap:	solid
tbarrcap:	solid
ticon:	left edge 1 pixel transparent

sicon:	top edge 1 pixel transparent
dicon:	top edge 1 pixel transparent
vwellbcap:	solid
vwelltcap:	solid
uicon:	top edge 1 pixel transparent
licon:	left edge includes black vertical pixel of left window edge
hwelllcap:	solid
hwellrcap:	solid
licon:	right edge 1 pixel transparent

Speed Concerns

In order to get the best possible speed for drawing window boundaries, we recommend that you draw all the icons with the same number of bits per pixel as the modes with which they are to be used. For example, the default set for mode 12 are all drawn in mode 12; the *xxx*22 set are drawn in mode 20 or 27. The window manager can, like the filer, draw the icon correctly whatever source mode is used, but it will paint the borders more slowly.

In order to avoid time consuming searches for names, the window manager caches the addresses of all the window icons. This cache is updated after a *ToolSprites command, and after mode changes.

The icons which are replicated should be made appropriately wide and tall, but diminishing returns do set in. A special problem for the right and bottom bars is that some of the replicated icons are hardly displayed at all in some circumstances: for example, hwelll spends a lot of time just one pixel wide. Making hwelll very wide to speed the repainting in these situations is counter productive.

We recommended you use the following sizes:

tbarmidt, tbarmidb:	128 pixels wide
vwellt, vwellb:	16 to 32 pixels tall
vbarmid:	128 pixels tall
hwelll, hwellr:	16 to 32 pixels wide
hbarmid:	128 pixels wide

Alternate Resolution Icons

RISC OS 2 allows for one icon file per application, which can be loaded automatically or under your control. Typically this contains the icons that the application needs the system to display on its behalf (eg those icons displayed by the filer). RISC OS 2 has all these icons at one particular resolution and number of colours: *nx*=2, *ny*=4, *bpp*=4 (mode 12). The appearance of these icons is often poor on other display modes, in particular high resolution monochrome and VGA or SuperVGA.

RISC OS 3 will load in different icon files automatically depending on the characteristics of the system's configured WimpMode. The *nx* and *ny* values are added to the end of any file, thus WimpMode 27 (VGA) will look for *file*22 by preference. If this fails *file* is used, which is expected to contain the mode 12 icons. No control over the number of bits per pixel is provided except for *nx*=2, *ny*=2, *bpp*=1 which will look for *file*23. Thus an application can be provided with icon files tailored to the various screens:

!Sprites
!Sprites22
!Sprites23

would be a standard configuration for the application providing icons optimised for normal TV standard monitor (AKA17), a VGA/SuperVGA monitor (or a Multisync monitor used as such) and a high resolution monochrome monitor.

The machine ends up with only one set of icons for the application being loaded into memory, thus using equivalent amounts of memory to RISC OS 2 (contrast with the multiple sets for the window icons). As standard, RISC OS 3 is provided with all its icons in the three above styles, though some are on disc.

Icon set

The provided icon set is subdivided into 7 sections:

Filer icons

directory	small_dir	application
small_app	file_xxx	small_xxx

These six icons are provided first: they are the most frequently used icons, so it makes sense to put them first on the search order. application and file_xxx are used in the event of searching for !foo or file_ded and not finding it, so limiting the time to find them is important.

These are followed by file_fff, small_fff, file_ffe, small_ffe, etc…

Icons in !Sprites are 34×17 and 9×9. The black outline is 2×1 pixels wide.

Icons in !Sprites22 and !Sprites23 are 34×34 and 18×18. The black outline is 1 pixel wide.

Icon bar icons

network	fileserver	small_fs
floppydisc	harddisc	ramfs
palette	romapps	switcher

These icons almost always appear on the icon bar.

Dialog box icons

yes	no	dontcare
radiooff	radioon	optoff
opton	tick	up
down	left	right
3 (menu tick)	⇒ (right submenu)	⇐ (left submenu)

These icons are provided for dialog boxes and menus. The last three are used by the Window Manager for menu pointers to submenus and ‘ticked’ items in menus.

Pointer icons

ptr_default	ptr_double	ptr_menu
ptr_write	ptr_hand	ptr_direct
ptr_cross	ptr_confirm	

These icons are used to change the shape of the pointer (from ptr_default) when providing feedback to the user. The Window Manager automatically shows ptr_double during a double click.

A transparency mask in the pointer icons defines the position of the active point. If the mask is present, it is scanned from the top down and the active point set to the position of the first transparent pixel in the mask, completely ignoring any program specified active point offset (e.g. from OS_SpriteOp 36).

Pinboard icons

ic_edit	ic_filer	ic_draw
ic_paint	ic_?	

These icons are displayed by the Pinboard for iconised applications or documents.

ic_? will be used if ic_*app* is not found.
ic_*app* should consist of ic_? with small_*app* in it. Its size should be the same as file_*xxx*.

Application icons

!edit	sm!edit	etc…

These icons are used by the filer, and should have the same size as file_*xxx* and small_*xxx* icons.

Misc icons

error	acorn

These are used respectively in error dialogue boxes, and for the Task Manager.

Design

Icons in the three sets must be as close to each other as possible. English text is not recommended in icons. Harmonisation of style with the provided icons is appreciated.

Etiquette

All the system provided icons, most particularly the Window Manager icons and the dialog box and menu icons, are used pervasively. Applications **must not** modify these icons for their own purposes (unless they are applications specifically provided to modify the look of the system as a whole).

Using 3D

You may wish to give your application a 3D look and feel. To do so you should provide two sets of icons, one of which provides a standard RISC OS 2 appearance, and the other of which provides a 3D appearance. You should then use OS_Byte 161 (page 1-369) to read the value of the 3D bit in CMOS RAM, which is bit 0 of byte 140; if it is clear load the RISC OS 2 style set, and if it is set load the 3D set.

Menus

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

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

The Wimp provides a way in which a task can define multi-level menu structures. By multi-level we mean that a menu item may have a submenu. The user activates this by moving the pointer over the right-arrow that indicates a submenu. The new menu is opened automatically, the Wimp keeping track of the ‘selection so far’.

The application usually activates a menu by calling Wimp_CreateMenu (page 3-153) in response to a Mouse_Click event of the appropriate type. It passes a pointer to a data structure that describes the list of menu items. Each of those items contains a pointer to its submenu, if required.

The click of the Menu button while the pointer is over a window is always reported, regardless of button types. You can use the window and icon handles to create a menu which accords to the context of the click. For example, the Filer varies its menu according to the current file selection (or pointer position if there is none).

When the user makes his or her menu choice by clicking on any of the mouse buttons while over an item, another event, Menu_Selection, is generated. The application responds to this by decoding the selected menu item(s) and performing appropriate actions.

Because menus can have a complex hierarchical structure (as opposed to the simple single level menus on some systems) a call Wimp_DecodeMenu (page 3-158) is provided to help translate the selection made into a textual form.

Just as icons can be made writable, menu items can have that property too. This makes it very easy to obtain input from the user while a menu is open.

Menus are not restricted to text-only items. A leaf item (i.e. the last in a chain of selections) may be a window, which in turn contains a complete dialogue box. And of course, such windows can have as many icons as required, displaying sprites, text prompts, writable icon fields etc.

It could be annoying that choosing an item from deep within a menu structure causes the whole menu to disappear. For example, the user might be experimenting with different selections from a colour menu, and he doesn't necessarily want to perform the whole menu operation again each time he clicks the mouse. To overcome this, selections made using the Adjust button do not cancel the menu. The Wimp supports this directly, but needs some co-operation from the application to make it work. See Wimp_CreateMenu for details on how to implement persistent menus.

Finally, because the Wimp can inform a task when a submenu is being opened, the menu tree can be built dynamically, according to the selections that have gone before.

Dialogue boxes

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

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

Basic operation

There is no direct way of setting up dialogue boxes under the Wimp. However, because icons can be handled in very versatile ways, it is quite straightforward to set up windows which act as dialogue boxes. If the necessary windows are permanently created and linked to the menu data structure, then the Wimp will handle all opening and closing automatically. The Wimp can be made to deal with button clicks within the window, for example automatically highlighting icons.

Also, because writable icons are available, it is a simple matter to input text supplied by the user, again with the Wimp doing most of the work. If required, the task can restrict the movement of the mouse to within the dialogue box, by defining a mouse rectangle (using OS_Word 21,1 – see page 1-712) which encloses the box. This ensures that the user can perform no other task until he or she responds to the dialogue box. The task

should always reset the mouse rectangle to the whole screen once the dialogue is over. Also, open_window_requests for the dialogue box should cause the box to be reset. Note that usually the pointer is not restricted. The dialogue box is deleted if you click outside it.

Alternatively, the menu tree can be arranged so that the application is informed (by a message from the Wimp) when the dialogue box is being opened; this allows any computed data to be delayed until the last minute. For a large program with many dialogue boxes this is preferable.

This form of dialogue box can be visited by the user without clicking on mouse buttons, just like traversing other parts of the menu tree. This is possible because redraw is typically much faster than on previous systems, so popping up the dialogue box and then removing it does not cause a significant delay.

Informational dialogue boxes

The ‘About this program’ dialogue box is a useful convention. Provide an ‘Info’ item at the top of the application’s menu, and make the dialogue box its submenu. You should also have the ‘Info’ item at the top of the menu that you produce when the user clicks with Menu on your icon bar icon. Use Edit’s template file to obtain an exact copy of the standard layout used in the Applications Suite programs.

Keyboard shortcuts

If a menu operation leading to a dialogue box has a keyboard short-cut, Wimp_CreateMenu should be used to initially open the dialogue box, rather than Wimp_OpenWindow (although Wimp_OpenWindow should still be used in response to an Open_Window_Request event). This will ensure that it has the same behaviour concerning cancellation of the operation etc as when accessed through the menu tree.

Static dialogue boxes

A static dialogue box is opened using Wimp_OpenWindow rather than Wimp_CreateMenu. A static dialogue box matches normal ones in colours, but has a Close icon.

Icons used in dialogue boxes

There are various forms of icon that occur within dialogue boxes, the most common forms are described here to improve consistency between applications.

Writable icons

Writable icons are used for various forms of textual fill-in field. They provide validation strings so that specific characters can be forbidden. Alternatively arbitrary filtering code can be added to the application to ensure that only legal strings (within this particular context) are entered.

When moving to a new writable icon, place the caret at the end of the existing text of the icon. See Wimp_SetCaretPosition for details of how to do this.

Action icons

This term refers to ‘buttons’ on which the user clicks on in order to cause some event to occur, typically the event for which the parameters have just been entered in the dialogue box. An example is the OK button in a ‘Save as’ dialogue box.

The best button type to use is 7 (Menu), with non-zero ESG. This will cause the button to invert while the pointer is over it (like a menu item), and for a button press to be reported.

It is sometimes appropriate to provide keyboard equivalents for action buttons. For instance, if the dialogue box is available via a function key as well as on the menu (see Keystrokes below) then adding key equivalents for action icons may mean that the entire dialogue box can be driven from the keyboard. A conventional use of keys is:

- Return – in the last writable icon. ‘Go’ – perform the obvious action initiated by filling in this dialogue box.
- Escape – cancel the operation; remove the dialogue box. Note that Escape is dealt with by the Wimp automatically in this case, as the dialogue box was opened using Wimp_CreateMenu.
- F2, F3 etc to F11 – if the action icons are arranged positionally at the top or bottom of the dialogue box in a simple row, then define F2, F3 etc as positional equivalents of the action buttons, i.e. F2 activates the left-most one, F3 the next etc. Note that F1 is normally reserved by convention to ‘get help’, so it should be used to provide help, or do nothing. Similarly, F12 should remain a route to the CLI.

Option icons

This term refers to ‘switches’, which can either be on or off.

The best icon to use is a text plus sprite one. The text has the validation string `Soptoff,opton`, where the sprites `optoff` and `opton` are defined in the Wimp ROM sprite area. The HVR bits of the icon flags (3, 4 and 9) are set to 0,1 and 0 respectively (see Wimp_CreateIcon). This generates a box to the left of the text, with a star within it if the option is on (i.e. the icon is selected). The button type is 11.

The ESG can be zero to make Select and Adjust both toggle the icon state, or non-zero (and unique) to make Select select and Adjust toggle the icon state.

The Filer’s menu item `Access` dialogue box for a particular file, uses this type of control (with ESG=0).

Radio icons

This term refers to a set of options where one, and only one, of a set of icons can be selected.

The text plus sprite form is again best, using the validation string `Sradiooff,radioon` from the Wimp sprite area, and a non-zero ESG shared by all the icons in the group, to force exclusive selection. If required, the icons can have their ‘adjust’ bit set to enable Adjust to toggle the state without deselecting the other icons.

Tool windows and ‘panes’

A pane is a window which is ‘fixed’ to another window, but has different properties from it. For example, consider a drawing program. You might have a scrollable, movable main window for the drawing area. This is called the tool window. On the left edge of this might be a fixed window which contains icons for the various drawing options. This lefthand window (the pane) always moves with the main window, but does not have scroll bars, or any other control areas.

Dealing with panes is really entirely up to the task program. However, there are one or two things to bear in mind when using them. If a tool window is closed, all of its panes must be closed too. Similarly, when a tool window is opened (an Open_Window_Request is received), the task must inspect the coordinates of the main window returned by the Wimp, and use them to open the pane in the appropriate position.

One bit in a window’s definition is used to tell the Wimp that this is a pane. This is used by the Wimp in two circumstances:

- if the pane gets the input focus, the tool window is highlighted
- when toggling the tool window size, the Wimp must treat panes as transparent.

There are various optimisations that can be used. If you open the windows in the right order, unnecessary redraws can be avoided.

Keyboard input and text handling

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

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

A task running under the Wimp should perform all of its input using the Wimp_Poll routine, rather than calling OS_ReadC or OS_Byte &81 directly. It is permissible for a program to scan the keyboard using the – ve inkey OS_Bytes. Further details are given in the chapter entitled *Character Output* on page 1-503.

The input focus

One window has what is termed the ‘input focus’. For example, the main text window of an editor might be the current input window, and its system area is highlighted by the Wimp to show this. (A flag can also be read by the program to see if it has the input focus.) The input window or icon also has a caret (vertical bar text cursor) to show the current input position.

A window gains the input focus if it has a writable icon over which the user clicks with Select or Adjust. The caret is positioned and sized automatically by the Wimp in this case. It uses a height of 40 OS units for the system font.

Alternatively, the program can gain the input focus explicitly by calling Wimp_SetCaretPosition (page 3-149). This displays a caret of a specified height and colour at the position specified in the given window and, optionally, icon. If the icon is a writable one, the Wimp can automatically calculate the position and height from the index into the text, if required.

Generally Wimp_SetCaretPosition is called in response to a mouse click over a window's work area. The position within the window must be calculated using the pointer position, the window's screen position, and the current scroll offsets.

Wimp_SetCaretPosition causes a couple of events to occur if the input window actually changes: Gain_Caret and Lose_Caret. This enables tasks to respond to the change in caret position (and possibly the task that owns it) by updating their window contents appropriately. This is especially true if an application is drawing its own caret and not relying on the Wimp's vertical bar. Note that the Wimp's caret is automatically maintained by the Wimp in Wimp_RedrawWindow, so you don't have to redraw it yourself.

Key presses

If the insertion point is within a writable icon, then many key presses are handled by the Wimp. The icon text is updated, and for certain cursor keys, the caret position and index within the string are updated. Other key presses, and all keys when the input focus is not in a writable icon, must be dealt with by the application itself.

A program gets to know about key presses through the Wimp_Poll Key_Pressed event. The data returned gives the standard caret information plus the code of the key pressed. It is up to the application to determine how the key-press is handled. There are certain standard operations for use in dialogue boxes, e.g. cursor down means go to the next item, but generally it will very much depend on what the application is doing.

Function and 'hot' keys

Among the keys that the Wimp cannot respond to automatically are the function keys F1 to F12. These are passed to the application as special codes with bit 8 set (i.e. in the range 256 - 511). If the application can deal with function keys, it should process the key press appropriately. If not, it should pass the key back to the Wimp with the call Wimp_ProcessKey (page 3-170).

If a function key is passed back to the Wimp in this way and the input focus belongs to a writable icon, the Wimp will expand the function key definition and insert (as much as possible of) the string into the icon.

In general, a program should always pass back key presses it doesn't understand to the Wimp. This allows the writing of programs which are activated by 'hot keys', for example, a screen dump that occurs when Print (F0) is pressed. Keys passed to

Wimp_ProcessKey are passed (through the Key_Pressed event) to tasks whose windows have the ‘grab hot keys’ bit set. They are called in the order they appear on the window stack, topmost first.

If a program can act on a hot key, it should perform its magic task and return via Wimp_Poll. If it doesn’t recognise that particular key, it should pass it to the next grab-hot-keys window in the stack by calling Wimp_ProcessKey before it next calls Wimp_Poll.

Note that the caret may well not be in the window with the grab-hot-keys bit set, and of course the caret position returned by Wimp_Poll will correspond to the window with the caret. Also, note that all potential hot key grabbers take priority over icon soft key expansion, and that you should not process a key and hand it back to the Wimp. This could lead to user-confusion.

If the only reason for a window is to allow its creator to grab hot keys, i.e. if it will never appear, it should be created and opened off the screen (with a large negative x position). To allow this, its window flags bit 6 should be set.

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

Special characters

Use Alt as a shifting key rather than as a function key. Different forms of international keyboards have standardised the use of Alt for entering accented characters. See the section entitled *Keyboard shortcuts* in the *Handling input* chapter in the *RISC OS Style Guide* for details of how you should implement modifiers.

Do not forbid the use of top-bit-set characters in your program, as many standard accented characters are available in the ASCII range &A0 - &FF. The Wimp clearly distinguishes between these characters and the function keys, which are returned as codes with bit 8 set.

The Escape key

Due to their frequent polling, Wimp programs do not normally need to use escape conditions. The Wimp sets the Escape key to generate an ASCII ESC (&1B) character. If you perform a long calculation without calling Wimp_Poll, you may set the escape action of the machine to generate escape conditions (using *FX 229,0), as long as you set it back again (using *FX 229,1 and then *FX 124) before calling Wimp_Poll.

One of the Wimp’s start-up actions (the first time Wimp_Initialise (page 3-85) is called) is to make the Escape key return ASCII 27. It does this by issuing an OS_Byte with R0=229, R1=1, R2=0. Thus no Escape conditions or (RISC OS) events are normally generated. The task that has the input focus can respond to ASCII 27 in any way it wants.

If you want to allow the user to interrupt the program by pressing Escape during a long operation, you can re-enable it using OS_Byte with R0=229, R1=0, R2=0. The following restrictions must be observed. Escapes must only be enabled between calls to Wimp_Poll, i.e. you **must not** call that routine with Escape enabled. This is very important. If you detect an Escape, you must disable it before calling the Wimp again and then clear it using OS_Byte with R0=124.

Even if no Escape occurs, you should still disable it before you next call Wimp_Poll; it is a good idea to call OS_Byte with R0=124 just after disabling Escapes.

It is also a good idea to display the Hourglass pointer during long-winded operations, preferably with the percentage of completion if this is possible. The user is less likely to try to interrupt if they can see that the operation is progressing. Note that you should not attempt to change the pointer while the hourglass is still showing.

When Wimp_CloseDown (page 3-172) is called for the last time (i.e. when the last task finishes), the Wimp restores the Escape key to its previous state, along with all the other settings it changed (function keys, cursor keys etc.)

Changing the pointer shape

You should not use the standard OS_Words and OS_Bytes to control the pointer shape under the Wimp. Instead, use the call Wimp_SpriteOp (page 3-198) with R0 = 36 (SetPointerShape). This programs the pointer shape from a sprite definition, performing scaling and colour translation if required. Pointer sprites have names of the form `ptr_xxxxx`. The standard arrow shape is held in the Wimp ROM sprite area and is called `ptr_default`.

The call Wimp_SetPointerShape (page 3-163) which was available before RISC OS 2 should no longer be used, although it is still provided for compatibility.

Pointer shape 1 is used by the Wimp as its default arrow pointer. Any program wishing to use a different shape must use shape 2, and program the pixels appropriately using the above call. Do not use logical colour 2 in pointer sprites, as this is unavailable in very high resolution modes. Shapes 3 and 4 are used by utilities such as the Hourglass module which changes the pointer shape under interrupts. For information about the SWIs supported by this module, refer to the chapter entitled *Hourglass* on page 2-745.

Note that when changing the pointer shape, it is recommended that the pointer palette is also reset. This is held in the sprite. Also, each sprite should have its own palette.

A task should only change the pointer when it is within the work area of one of its windows. The Wimp_Poll routine returns two event codes for detecting this: Pointer_Entering_Window and Pointer_Leaving_Window (5 and 4 respectively).

Whenever the first code is received, the task can change the pointer to shape 2 for as long the pointer stays within the window. On receiving the second code, the task should reset the pointer to shape 1. The best way to achieve this is to use the *Pointer command.

Tasks should trap Message_ModeChange, as a mode change resets the pointer to its default shape. If, on a mode change, the task thinks that it 'owns' the pointer, i.e. it is over one of the task's windows, it should re-program the pointer shape, if required.

Mode independence

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

Programs should work in all screen modes in which the Wimp works. Read the current screen mode rather than setting it when your program is loaded, and call OS_ReadVduVariables (page 1-730) to obtain resolution, aspect ratio, etc, instead of building these into the program.

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

Programs uninterested in colours must also check operation in 256-colour modes, e.g. some EOR (exclusive OR) tricks do not work quite the same. For instance, see Wimp_SetCaretPosition for a description of how the Wimp draws the caret using EOR plotting. Clock uses a similar trick for the second hand of the clock. As another example, Edit uses EORing with Wimp colour 7 (black) to indicate its selection, but redraws the text in 256-colour modes.

In two-colour modes the Wimp uses ECF patterns for Wimp colours 1 to 6 (grey levels). Note that certain EOR-ing tricks do not work on these, and that use of Wimp_CopyBlock can cause alignment problems for the patterns.

An important aspect of Wimp-based applications is that they do not depend for their operation on a particular screen mode. A corollary of this is that they should not explicitly change display attributes such as mode or colours. The motivation for this rule is to ensure that many separate tasks can be active without mutual interference.

To help programs operate in a consistent manner regardless of, say, the number of screen colours, the Wimp provides a variety of utility functions, such as colour translation and the scaling of sprites and text. In fact many of these features are provided by other parts of RISC OS, but are given Wimp calls to facilitate a more uniform interface.

Colour

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

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

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

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

There are several colours used in drawing a window. For harmonious operation with other applications, several of these have been standardised: you should set the Title Bar colours, the scroll bar inner and outer colours and highlighted title colour to the values given in the table in the following section on colour handling, unless you have some good reason not to. On the other hand, the work area colours (which are set for you before an update or redraw) can be assigned any values required.

Colour handling

The Wimp's model of the display centres on the 16-colour modes. There are 16 Wimp colours defined, listed below. In other modes, the Wimp performs a mapping between these standard colours and those which are actually available. When setting colours for graphics (including VDU 5 text), or anti-aliased fonts, the application specifies standard colours to the appropriate Wimp routine, which translates them and generates the necessary VDU calls.

Here are the standard colours, and their usages:

standard colour	usage
0 - 7	grey scale from white (0) to black (7) colour 1 is icon bar and scroll bar inner colour colour 2 is standard window title background colour colour 3 is the scroll bar outer colour colour 4 is the desktop background colour
8	dark blue
9	yellow
10	green
11	red
12	cream, window title background for input focus owner
13	army green
14	orange
15	light blue

In non-16 colour modes, these standard colours are represented as follows:

2-colour modes	logical colour 0 is set to Wimp colour 0, i.e. white logical colour 1 is set to Wimp colour 7, i.e. black
0	logical colour 0
1 - 6	decreasing brightness stippled patterns
7	logical colour 1
8 - 15	logical colour 0 or 1, whichever is closer to standard colour's brightness level
4-colour modes	logical colour 0 is set to Wimp colour 0, i.e. white logical colour 1 is set to Wimp colour 2, i.e. light grey logical colour 2 is set to Wimp colour 4, i.e. dark grey logical colour 3 is set to Wimp colour 7, i.e. black
0 - 15	set to the logical colour closest in brightness to the standard one
256-colour modes	the default palette is used
0 - 15	set to the closest colour to the standard one obtainable

As an example of the use of colour translation, if you were to set the graphics colour to 2 in a two-colour mode, using Wimp_SetColour (page 3-191), then the Wimp would actually set up an ECF pattern (number 4 is used) to be a lightish stippled pattern, and issue a GCOL to make ECF 4 the current graphics colour. On the other hand, in a 256-colour mode it would calculate the GCOL and TINT which gives the closest match to the standard light grey, and issue the appropriate VDUs.

In 256-colour modes, exact representations of the Wimp colours 0 - 7 (the grey scale) are available, but only approximate (albeit pretty close) representations of Wimp colours 8 - 15 can be obtained.

The Wimp utilises its colour translation mechanism in the following circumstances:

- when using the colours given in a window's definition, unless bit 10 of the window flags is set. In this case, the colour is used directly. NB in a 256-colour mode an untranslated colour is given as `%cccccctt`, i.e. bits 0 - 1 give bits 6 - 7 of the `TINT` and bits 2 - 7 give bits 0 - 5 of the `GCOL`.
- when using the colours in an icon's definition. Text colours are translated, except that the stippled patterns can't be used in two-colour modes. Sprites are plotted using the OS_SpriteOp PutSpriteScaled reason code with an appropriate colour table and scaling factors.
- when using the text caret colour, unless translation is overridden.

The palette utility produces a broadcast message when the user changes the palette settings, allowing such programs to repaint for the new palette. A module called ColourTrans (used by Paint and Draw) gives the closest setting possible to a given RGB value. This module is provided in the RISC OS 3 ROM and is available as a RAM loaded module for RISC OS 2.

If you want to override the Wimp's translation of colours, you can use the ColourTrans module and PutSpriteScaled to perform more sophisticated colour matching. The Draw and Paint applications do this.

System font handling

The system font is the standard 8 by 8 pixel character set. It is used by OS_WriteC text printing codes. Under the Wimp, the system font is defined to be 16 units wide by 32 OS units high. This is true regardless of the actual screen resolution. The consequence of this is that system font characters are the same physical size, independent of the screen mode.

To obtain the appropriate sizing of characters, the Wimp uses the VDU driver's ability to scale characters printed in VDU 5 mode. Thus in mode 4, where a pixel is 4 OS units wide, system font characters are only four pixels wide, to maintain their 16 OS unit width. Similarly in 512-line modes, characters are plotted double height to give them the same appearance as in mode 12.

Dragging

Dragging boxes

One of the recognisable features of most window systems is the ability to 'drag' items around the screen. The RISC OS Wimp is no exception, and provides extensive facilities for dragging objects.

Icons and window work areas can be given a button type which causes the Wimp to detect drag operations automatically. A 'drag' is defined as the Select or Adjust button being pressed for longer than about 0.2s. Alternatively, if the user clicks and then moves the mouse outside the icon rectangle before releasing, this also counts as a drag. The result is that a Mouse_Click event is returned by Wimp_Poll. Note that before a drag event is generated, the application will also be informed of the initial click, and the drag could in turn be followed by a double click event, depending on the button type.

The call Wimp_DragBox (page 3-142) initiates a dragging operation. The user supplies the initial position and size of the box to be dragged, and a 'parent' rectangle within which the dragging must be confined. Normally, the initial position of the box will be

such that the mouse pointer is positioned somewhere within the box. However, this is not mandatory; the Wimp, while performing the dragging, ensures that the relative positions of the pointer and the box remain constant.

There are two main types of drag operation: system and user. System types work on a given window, and drag its size, position or scroll offsets. These drags are normally performed automatically if the window has the appropriate control icon (e.g. a Title Bar to drag its position). However, you might want to allow a non-titled window to be moved, or a window without an Adjust Size icon to be resized; the system drag types cater for this sort of operation.

User drag boxes can be fixed size, where the whole of the box is moved along with the pointer, or variable sized, where the top left of the box is fixed, and the bottom-right moves with the pointer. (The fixed and movable corners can be varied by specifying the box's top left and bottom right coordinates in the reverse order.) The Wimp displays the drag box using dashed lines whose dash pattern changes cyclically.

There is an 'invisible' type of drag box. In this case, the mouse is simply constrained to the parent rectangle, which must be a single window, and the initial box coordinates are ignored. It is up to the task to draw the object being dragged. This usually involves setting a 'dragging' flag in the main poll loop, and the use of Wimp_UpdateWindow (page 3-128). The task must also ensure that the dragged object is redrawn if a Redraw_Window_Request is issued, and enable Null event codes and use them to perform tracking.

Finally, a program can arrange for the Wimp to call its own machine code routines during dragging, for the ultimate in flexibility. This enables the program to drag any object it likes, so long as it can draw it and then remove it without affecting the background. In this case, the object can go outside the window. The Wimp will ask for it to be removed at the appropriate times.

In all cases, the task is notified when the drag operation ends (when the user releases all mouse buttons) by Wimp_Poll returning the event code User_Drag_Box.

Drag operations within a window

The Wimp's drag operations are specifically for drags that must occur outside all windows. As well as the cycling dashed box form, they allow the use of user-defined graphics, allowing arbitrary objects to be dragged between windows.

If you build drag operations within your window, check that redraw works correctly when things move in the background (the Madness application is useful for testing this). Also, it is important to note that such 'within-window' dragging must use Wimp_UpdateWindow to update the window, rather than drawing directly on the screen.

If the drag works with the mouse button up then menu selection and scrolling can happen during the drag, which is often useful. Stop following the drag on a Pointer_Leaving_Window event, and start again on a Pointer_Entering_Window event.

If the drag works with the button down, then it may continue to work if the pointer is moved out of the window with the button still down. Alternatively for button-down drags, you can restrict the pointer to the visible work area, and automatically scroll the window if the pointer gets close to the edge.

Editors

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

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

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

Terminology

Each document being edited is typically displayed in a window. Such windows are referred to as editor windows.

Most editors record, for each document currently being edited, whether the user has made any adjustments yet to the document. This is known as an updated flag.

Some editors are capable of editing several documents of the same type concurrently, while others can edit only one object at a time. Being able to edit several documents is frequently useful, and removes the need for multiple copies of the program to be loaded. Such programs are referred to here as multi-document editors. Edit, Draw and Paint are all multi-document editors, while Maestro and FormEd are not.

File types

Editors use RISC OS file types to distinguish which application belongs to which file. Application `!Boot` files should define `Alias$@RunType_ttt` and `File$Type_ttt` variables, and `!appl`, `sm!appl`, `file_ttt` and `small_ttt` sprites (in the Wimp sprite area), as described earlier. File types are allocated as described in the section entitled *Filetypes* on page 4-551.

The user interface

The user interface of RISC OS concerning loading and saving documents is rather different from that of other systems, because of the permanent availability of the Filer windows. This means that there is no need for a separate ‘mini-Filer’ which presents access to the filing system in a cut-down way. Although this may feel unusual at first to experienced users of other systems, it soon becomes natural and helps the feeling that applications are working together within the machine, rather than as separate entities.

Editor icons

Icons that appear on the icon bar should have bounding boxes 68 OS units square. Icons with a different height are strongly discouraged, as they will have their top edges aligned within the Filer Large icon display. A wider icon is permissible, but the size above should be thought of as standard. If the width is greater than 160 OS units then the edges will not be displayed in the Filer Large icon display.

Icons are often displayed half size to save screen space. The Filer will use `sm!appl` and `small_ttt` if these are defined, or scaled versions of `!appl` and `file_ttt` if not.

Starting an editor

The standard ways to start an editor are to:

- double-click on the application icon within the directory display, or
- double-click on a document icon within the directory display.

The action taken in the first case is to load a new copy of the application (by running its !Run file). The only visible effect to the user is that the application icon appears on the icon bar. So when you start up with no command line arguments, use Wimp_CreateIcon to put an icon containing your !app sprite onto the icon bar, then enter your polling loop quietly.

In the second case, create the icon bar icon, load the specified document and open a window onto it. This typically occurs by the activation of the run-type of the document file, which in turn will invoke the application by name with the pathname of the document file as its single argument.

For example, the run-type for a Draw file (type &AFF) is:

```
*Run <disc>.!Draw.!Run %*0
```

where `<disc>` is the name of the disc on which the Draw application resides. So when the user double-clicks on a type &AFF file, the Filer executes `*Run pathname`, which in turn executes `<disc>.!Draw.!Run pathname`.

Typically, the !Boot file of the application sets up the run-type for its data files when the application is first seen by the filer. In the case of Draw, the boot file says:

```
*Set Alias$@RunType_AFF *Run <Obey$Dir>.!Run %%*0
```

See the section entitled *Application resource files* on page 3-56 for details.

When a document icon is double-clicked, and a multi-object editor of the appropriate type is already loaded, it is not necessary to reload the application. In this case, the active application will notice the broadcast message from the Filer announcing that a double click has occurred, and will open a window on the document itself. For details, see the section entitled *Message_DataOpen (5)* on page 3-262.

A further way of opening an existing document is to drag its icon from the Filer onto the icon bar icon representing the editor. In this case, a DataLoad message is sent by the Filer to the editor, which can edit the file. This form is important because it specifies the intended editor precisely. For instance if both Paint and FormEd are being used (both can edit sprite files) then double-clicking on a sprite file could load into either.

Newly opened windows on documents should be horizontally centred in a mode 12 screen, and should not occupy the entire screen. This emphasises that the application does not replace the existing desktop world, but is merely added to it. Subsequent windows should not totally obscure ones that this application has already opened. Use a –48 OS unit y offset with each new window.

Creating new documents

The window created from the loading or creation of a document should be no larger than about 700 OS units wide by 500 high. The first window should be centred horizontally and vertically on the screen. Open subsequent windows 48 OS units lower than the previous one, but if this would overlap the icon bar then return to the original starting position. The initial size and position of windows should be user-configurable, by editing a template file.

Editing existing documents

To open an existing document, double-click on the document in the Filer. This will cause a broadcast DataOpen message from the Filer, so if your editor can edit multiple documents it can intercept this and load the document into the existing editor.

To insert one document into another, drag the icon for the file to be inserted into the open window of the target document. The Filer will then send a message to that window, giving the type and name of the file dragged. The target (if the file is of a type that can be inserted) can now read the file. If the file is not of a type that can be inserted in this document then the editor should do nothing, i.e. it should not give an error.

More details of these operations can be found in the section entitled *Wimp_SendMessage (SWI &400E7)* on page 3-193.

Saving documents

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

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

Closing document windows

If the user clicks on the Close icon of a document window, and there is unsaved data, then you should pop up a dialogue box asking:

- Do you want to save your edited file? (if the document has no title)
- Do you want to save edited file ‘name’?

You can copy this dialogue box from Edit’s template file. If the answer is `Yes` then pop up a Save dialogue box, and if the result is saved then close the document window. If the answer is `No`, or any cancel-menu (e.g. Escape) occurs, then the operation is abandoned.

If the user clicks Adjust on the Close icon, call Wimp_GetPointerInfo on receipt of the Close_Window_Request. Also, you must open the file’s home directory after closing it. This can be obtained by removing the leafname from the end of the file’s name and sending a Message_FilerOpenDir broadcast to open the directory.

Quitting editors

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

See *Message_PreQuit (8)* on page 3-228 for details of what to do if your editor receives a PreQuit broadcast message.

Memory management

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

Part of the Wimp's job is to manage the system's memory resources. There are several areas: the screen, system sprites, fonts, the RMA, application space etc. Many of these are controllable through the Task Manager's bar display. The user can drag, say, the font cache bar to set the desired size.

The remainder, when all of the other requirements have been met, is called the free pool. The Wimp can 'grab' memory from this to increase another area's size, or to start a new application, and extend it when another area is made smaller, or an application terminates. Because the allocation of memory is always under the user's control, he or she can make most of the decisions concerned with effective utilisation.

Two important bars in the Task Manager's display are the 'Free' and 'Next' ones. These give respectively the size of the free memory pool, and the amount of memory that will be given to the next application. They can be dragged to give the desired effect. For example, the user can decrease the RAM disc slot to increase the 'Free' size, which will in turn allow another resource, e.g. the screen size, to be increased. This is only used if the task doesn't issue an explicit *WimpSlot command, though most will do so.

Using the memory mapping capabilities of the MEMC chip, the Wimp can make all applications' memory appear to start at address &8000. This is called logical memory, and is all the application need worry about. Logical memory is mapped via the MEMC into the physical memory of the machine. The smallest unit of mapping is called a page, and its size is typically 8K or 32K bytes. Before giving control to a task through Wimp_Poll, the Wimp ensures that the correct pages of physical memory are mapped into the application workspace at address &8000.

In general, then, the application need not concern itself with memory allocation. However, there are times when direct interaction between a task and the Wimp's allocation is desirable. For example, a program may need a certain minimum amount of memory to operate correctly. Conversely, when running an application might decide that it doesn't need all of the memory that was allocated to it, and give some back.

The SWI Wimp_SlotSize (page 3-203) allows the size of the current task's memory and the 'Next' slot to be read or altered. See the description of that call for details of its entry and exit parameters and examples of its use. The command *WimpSlot uses the call.

A program may need a large amount of memory for a temporary buffer. Just as it is possible to claim the screen memory using OS_ClaimScreenMemory, a program can call Wimp_ClaimFreeMemory (page 3-208) to obtain exclusive use of the Wimp's free pool. Only programs executing in SVC (supervisor) mode can make use of this memory, as it is protected against user-mode access. Furthermore, while the memory is claimed, the Wimp cannot dynamically alter the size of other areas, so programs should not 'hog' it for extended periods (i.e. across calls to Wimp_Poll).

Finally, just as built-in resources such as RMA size and sprite area size are alterable by dragging their respective bars, the Task Manager allows the user to perform the same operation on task bars. This is only possible with the task's cooperation. When a task starts up, the Task Manager asks it, by sending a message, if it will allow dynamic sizing of its memory allocation. If the program responds, the Task Manager will allow dragging of its bar, otherwise it won't. See the section entitled *Message_SetSlot (&400C5)* on page 3-239 for details.

Applications with complex requirements can arrange to call Wimp_SlotSize at run-time to take (and give back) memory. BASIC programs may use the `END=&xxxx` construct to call Wimp_SlotSize.

C programs should call Wimp_SlotSize directly or use 'flex' (available with Release 3 or later of the Acorn C Compiler), which provides memory allocation for interactive programs requiring large chunks of store.

If Wimp_SlotSize is used directly, the language run-time library (and malloc()) will be entirely unaware that this is happening and so you must organise the extra memory yourself. A common way of doing this is to provide a shifting heap in which only large blocks of variable size data live. By performing shifting on this memory, pages can be given back to the Wimp when documents are unloaded.

Important:

- Do not reconfigure the machine.
- Do not kill off modules to get more workspace.

Such sequences are quite likely to be hardware-dependent and OS version-dependent.

Template files

To facilitate the creation of windows, a ‘template editor’, called FormEd, has been written for the Wimp system. This allows you to use the mouse to design your own window layouts, and position icons as required. An extensive set of hierarchical menus provides a neat way of setting up all the relevant characteristics of the various windows and icons.

Once a window ‘template’ has been designed, it can be given an identifier (not necessarily the same as the window title) and saved in a template file along with any other templates which have been set up and identified. The Wimp provides a Wimp_OpenTemplate (page 3-165) call, which makes it very simple for a task, on start up, to load a set of window definitions. The task can load a named template from the file, which can then be passed straight to Wimp_CreateWindow (page 3-87), or it can look for a wildcarded name, calling Wimp_LoadTemplate (page 3-167) repeatedly for each match found.

Many of the templates used by the system are resident in ROM. They are held in Resources:$.Resources.*, where * is the name of the module. You can base your own templates on these by loading a ROM file into the template editor (FormEd – available with Release 3 or later of the Acorn C Compiler), modifying it and re-saving it in your own file. For example, the palette utility template file contains the ‘Save as’ dialogue box, which all applications should use (with a change of sprite name).

It is also possible to override the system’s use of the ROM template files by setting App$Path, where App is the application name. These variables contain a comma-separated list of prefixes, usually directory names, in which the Wimp will search for the directory Templates when opening template files. Their default value points to the ROM, but you could change it to, say, ADFS::MyDisc.<old values> to make it look for modified, disc-resident versions of the standard template files first. Note that directory names must end in a dot.

There are two issues associated with the loading of window templates from a file. These concern the allocation of external resources:

- resolving references to indirected icons
- resolving references to anti-aliased font handles.

In the first case, what happens is that the relevant indirected icon data is saved in the template file. When the template is loaded in, the task must provide a pointer to some free workspace where the Wimp can put the data, and redirect the relevant pointers to it. The workspace pointer will be updated on exit from the call to Wimp_LoadTemplate. If there is not enough room, an error is reported (the task must also provide a pointer to the end of the workspace). Having loaded the template, the program can inspect the icon block to determine where the indirected data has been put.

The issue concerning font handles is more difficult to solve. The template file provides the binding from its internal font handles to the appropriate font names and sizes. In addition, the Wimp must also have some way of telling the task which font handles it actually bound the font references to when the template was loaded. This is so the task can call Font_LoseFont as required when the window is deleted (or alternatively, when the task terminates).

To resolve this, the task must provide a pointer to a 256 byte array of font 'reference counts' when calling Wimp_LoadTemplate. Each element must be initialised to zero before the first call. Font handles received by the Wimp when calling Font_FindFont are used as indices into the array. Element i is incremented each time font handle i is returned.

So, when Load_Template returns, the array contains a count of how many times each font handle was allocated. On closing the window or terminating, the program must scan the array and call Font_LoseFont the given number of times for non-zero entries. As with icon pointers, the program can find out the actual font handles used by examining the window block returned by Wimp_LoadTemplate.

It is up to the programmer to decide whether it is sufficient to provide just one array of font reference counts, so that the fonts can be closed only when all the windows are deleted (or the task terminates), or whether a separate array is needed for each window. Of course, considerable space optimisations could be made in the latter case if the array were scanned on exit from Wimp_LoadTemplate and converted to a more compact form.

If a task is confident that its templates do not contain references to anti-aliased fonts, then the array pointer can be null, in which case the Wimp reports an error if any font references are encountered.

Note that if anti-aliased fonts are used, the program must also rescan its fonts when Message_ModeChange is received. This involves calling Font_ReadDefn for each relevant font handle, changing to the correct xy resolution, and calling Font_FindFont again. The new font handle can be put back in the window using Wimp_SetIconState.

Application resource files

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

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

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

If an application is intended for international use then all textual messages within the program should be placed in a separate text file, so that they can be replaced with those of a different language. It may be unhelpful for the application to read such messages one by one, however, as this forces the user of a floppy disc-based system to have the disc containing the application permanently in the drive. Error messages should all be read in when the application starts up, so that producing an error message does not cause a `Please insert disc` *`disc title`* message to appear first.

Note that Obey$Dir and obey files are important here. Applications must always be invoked with their full pathnames, so that Obey$Dir is set correctly. For example, if a resource file is accessed later when the current directory has changed, using a full pathname means it will work OK.

Resources may also be updated by the program during the course of execution. For instance, if an application has user-settable options which should be preserved from one invocation of the program to the next, then saving them within the application directory means that the user does not have to worry about separate files containing such data. As a source of user-settable options this technique is preferable to reading an environment string, since with the latter system the user has to understand how to set up a boot file.

The !Appl.!Sprites file

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

This file must be of type ‘Sprite’.

The !Appl.!Run file

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

Example

Here is an example !Run file:

```
WimpSlot –min 260K –max 260K
RMEnsure FPEmulator 2.60 RMLoad System:Modules.FPEmulator
RMEnsure FPEmulator 2.60 Error You need FPEmulator 2.60 or
later
|
|         also RMEnsure SharedCLibrary and ColourTrans modules
|
Set Draw$Dir <Obey$Dir>
Set Draw$PrintFile printer:
Run "<Draw$Dir>.!RunImage" %*0
```

The action of these commands is to respectively

- call *WimpSlot to ensure that there is enough free memory to start the application.
  Draw, like many applications, knows exactly how much memory it should be loaded with. It acquires more memory once executing (without the knowledge of the language system underneath) by calling SWI Wimp_SlotSize. Paint, Draw and Edit all maintain shifting heaps above the initial start-up limit, ensuring that extra memory is always given back to the central system when it is not needed.
  Applications can also arrange to have the user control dynamically how much memory they should have, by dragging the relevant bar in the Task Manager display. See the section entitled *Message_SetSlot (&400C5)* on page 3-239 for details.
- ensure that any soft-loaded modules that the application requires are present, using *RMEnsure. If your call to *RMEnsure can load a module from outside your application directory then you should call it twice, to ensure that the newly loaded module is indeed recent enough. If the *RMLoaded module comes from your application directory, one *RMEnsure is sufficient.

- set an environment variable called Draw$Dir from Obey$Dir. (Note that you should not use the variable `Obey$Dir` as another macro could quite likely change the setting of `Obey$Dir`, so it is safer to make a copy.) This allows Draw to access its application directory once the program itself is running, enabling it to access, for example, template files by passing the pathname `<Draw$Dir>.Templates` to Wimp_OpenTemplate. In general you should use the variable *`Appl`*`$Dir` if the application is called `!`*`Appl`*.
- set another environment variable. Different applications will have their own requirements.
- run the executable image file. !RunImage is the conventional name of the actual program. It is also used by the Filer to provide the date-stamp of an application in the Full info display. Note that this time there is only a single % to mark the parameter, as the parameters passed to the *Obey command must be substituted immediately. If this line is at the end of the !Run file it must not have a terminating CR/LF, otherwise the !Run file will remain open until the application (and hence the !RunImage file) is quit.

Other possible actions that may occur within !Run files are

- execute !Boot. This will usually have been done already, but in the presence of multiple applications with the same name the !Boot file of a different one may have been seen first. This can be done explicitly using a command such as `*Run <Obey$Dir>.!Boot`, or you could just edit the !Boot file into the !Run file.
- if shared system resources are used then ensure that System$Path is defined, and produce a clean error message if it is not. For example:

  *If "<System$Path>" = "" Then Error 0 System resources cannot be found
- loading a module can take memory from the current slot size, so the *WimpSlot call must be called after loading modules. If you do it both before and after, you avoid loading modules in the case where the application definitely won't fit anyway.

  However, some applications wish to ensure that there is also some free memory after they have loaded, for example if they use the shifting heap strategy outlined above. Such applications may call *WimpSlot again just before executing !RunImage, with a slightly smaller slot setting, to leave just the right amount in the current slot while at the same time ensuring that there is some memory free.

It should be emphasised that the presence of multiple applications with the same name should be thought of as an unusual case, but should not cause anything to crash. Also, complain 'cleanly' if your resources can no longer be found after program start-up.

One point to note here is that when an application is starting up from its *Run file, if a screen mode change is to take place, you must call `*WimpSlot 0 0` before the change and reset the slot size afterwards.

Shared resources

The recommended approach is to create an application directory whose !Boot file sets up an environment variable which other applications use to access the shared resources (within the shared resource directory).

!System is an example of such a shared resource, which provides shared resources for the RISC OS welcome disc applications. Note that other applications may rely on using !System resources, **but** further resources **must not** be put into !System. These should instead go into their own shared resource directories, with names obtained by applying to Acorn. (See the section entitled *Shared resources* on page 4-555.)

This approach ensures that users can view shared resources as fixed objects that must be present for other applications to work, and not have to worry about what is inside them.

Where upgrades of a particular shared resource are concerned, the old copy should be archived and deleted from view, to avoid the possibility of accidental access to the old information. Note that if this does occur, the resulting error messages should make it clear to the user what to do next.

Relocatable module tasks

A program using the Wimp can be loaded from disc into the application memory (&8000), or may be a relocatable module resident in the RMA (relocatable module area). In the main, Wimp tasks of both varieties work in the same way and have similar structures. However, module tasks must additionally cope with service calls generated at various times by the Wimp. They must also be able to terminate when asked to, e.g. during an *RMTidy operation.

In this section we describe the special requirements of module tasks, but not how to write modules from scratch. See the chapter entitled *Modules* on page 1-201 for details. You may also like to read the sections on *Wimp_Initialise (SWI &400C0)* on page 3-85 and *Wimp_CloseDown (SWI &400DD)* on page 3-172 before going over the listings below.

Much of the following is concerned with service call handling. A general, and very important, aspect of this is register usage. A module service handler can modify registers R0 - R6 that have been explicitly stated to be return parameters for each individual service call. However, these registers should not be modified, except to produce a particular effect as defined below. Badly behaved service code which does not adhere to this can produce bugs which are very difficult to track down and cause the system to fail in unpredictable ways.

Task initialisation

Tasks are started using a * Command. This is decoded by the module's command table and the appropriate code to handle the command is called automatically. This is standard module code, and looks like this:

```
;This is pointed to by the entry for the module's * Command
myCommandCode
        STMFD   SP!, {LR}       ;Save the link register
        MOV     R2, R0          ;R2 points at command tail
        ADR     R1, titleStr    ;R1 points at title string of module
        MOV     R0, #2          ;Module 'Enter' reason code
        SWI     XOS_Module      ;Enter the module as a language
        LDMFD   SP!, {PC}       ;Return (in case that failed)
WIMP_VER * 200
titleStr
        DCB     "MyModule",0    ;as returned by *Modules
        ALIGN
TASK  DCB"TASK"

;This is the module's language entry point
startCode
        LDR     R12, [R12]      ;Get workspace pointer claimed in Init entry
        LDR     R0, taskHandle
        TEQ     R0, #0          ;Are we already running?
        LDRGT   R1, TASK        ;Yes, so close down first
        SWIGT   XWimp_CloseDown
        MOVGT   R0, #0          ;Mark as inactive
        STRGT   R0, taskHandle
;Now claim any workspace etc. required before initing the Wimp
;...
;If all goes well, we end up here
        MOV     R0, #WIMP_VER   ;(re)start the task
        LDR     R1, TASK
        ADR     R2, titleStr
        SWI     XWimp_Initialise
        BVS     startupFailed   ;Tidy up and exit if something went wrong
        STR     R1, taskHandle  ;Save the non-zero handle
...
```

Thus when the user enters the appropriate * Command, the module is started as a language and the start code is called using the word at offset 0 in the module header. It is entered in user mode with interrupts enabled, and R12 pointing at its private word.

On entry, the task checks to see if it is already active. If it is, it closes down (to avoid running as two tasks at once). It also resets its `taskHandle` variable to indicate that it is inactive. It then performs any necessary pre-Wimp_Initialise code, such as claiming workspace from the RMA. If this succeeds, it calls Wimp_Initialise and saves the returned task handle.

Errors

Always check error returns from Wimp calls. Beware errors in redraw code; they are a common form of infinite loops (because the redraw fails, the Wimp asks you again to redraw, and so on). A suddenly missing font, for instance, should not lead to infinite looping. Check that the failure of Wimp_CreateWindow or Wimp_CreateIcon does not lead you to crash or lose data.

Check cases concerning running out of space.

If the user is asked to insert a floppy disc and selects `Cancel`, you get an error `Disc not present` (&108D5) or `Disc not found` (&108D4) from ADFS. If you get either of these errors from an operation you need not call Wimp_ReportError, just cancel the operation. This avoids the user getting two error boxes in a row.

Do not have phrases like '`at line 1230`' in error messages from BASIC programs; '`(internal error code 1230)`' is preferable.

Error messages

&280 Wimp unable to claim work area
The RMA area is full

&281 Invalid Wimp operation in this context
Some operations are only allowed after a call to Wimp_Initialise

&282 Rectangle area full
Screen display is too complex
(this error message only appears under RISC OS 2)

&283 Too many windows
Maximum 64 windows allowed
(this error message only appears under RISC OS 2)

&284 Window definition won't fit
No room in RMA for window

&286 Wimp_GetRectangle called incorrectly

&287 Input focus window not found

&288 Illegal window handle

&289 Bad work area extent
Visible window is set to display a non-existent part of the work area

&29F Bad parameter passed to Wimp in R1
The address in R1 was less than &8000, i.e. outside of application space

Most of the above errors are provided as debugging aids to development programmers, and should not occur when the system is working properly, except for `Too many windows`, which can happen if a task program allows the user to bring up more and more windows. The error is not serious, as long as the task program's error trapping is written properly – when creating a window, you should only update any data structures relating to it once the window has been successfully created.

Time

There are two clocks that keep track of real time in the system, the hardware clock and a software centi-second timer. The two can diverge by a few seconds a day, but are resynchronised at machine reset. For consistency, always use the centi-second timer.

When using Wimp_PollIdle, remember that monotonic times can go negative (i.e. wrap round in a 32-bit representation) after around six weeks. So when comparing two times the expression

(newtime – oldtime) > 100

is a better comparison than

newtime > oldtime + 100.

Wimp behaviour under RISC OS 3

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

However, it is occasionally necessary to make changes that could potentially confuse an application which was not aware of them. In order to cope with this, the Wimp allows an application to inform it of how much it knows when it calls Wimp_Initialise, by supplying in R0 the version number of the latest release of the Wimp which the programmers have taken into account.

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

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

Applications written for RISC OS 2 should all have R0 set to 200 when calling Wimp_Initialise.

Under RISC OS 3 an application can only pass 200, 300, or 310 to Wimp_Initialise. The Wimp will give an error if any other value is passed in.

Service Calls

The next section describes those service calls that are of particular relevance to you when you are writing modules to run under the Window Manager. The remaining service calls that RISC OS provides are documented in the chapter entitled *Modules* on page 1-201.

Service Calls

Service_Memory (Service Call &11)

Memory controller about to be remapped

On entry

R0 = amount application space will change by
R1 = &11 (reason code)
R2 = current active object pointer (CAO)

On exit

R1 = 0 to prevent re-mapping taking place

Use

This is issued when the contents-addressable memory in the memory controller is about to be remapped, which alters the memory map of the machine. You should claim this call if you don't want the remapping to take place.

A module will initially be given the current slot size for its application workspace starting at &8000. However, modules do not generally need this area, as they use the RMA for workspace. Therefore, when a task calls Wimp_Initialise, the Wimp inspects the CAO. If this is within application workspace, the Wimp does nothing. However, if the CAO is outside of application space (a module's CAO is its base address in the RMA or ROM), the Wimp will reduce the current slot size to zero automatically, except as described below.

Some modules, notably BASIC, do require application workspace. Therefore the Wimp makes this service call just before returning the application space to its free pool. A task can object to the remapping taking place by claiming the call. The Wimp will then leave the application space as it is.

Service_Reset (Service Call &27)

Post-Reset

On entry

R1 = &27 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

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

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

See also page 2-502 and page 3-70.

Service_StartWimp (Service Call &49)

Start up any resident module tasks using Wimp_StartTask

On entry

R1 = &49 (reason code)

On exit

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

Use

The Desktop will try to start up any resident module tasks when it is called (using *Desktop or by making the task the start-up language). It does this by issuing a service call Service_StartWimp (&49). If this call is claimed, the Desktop starts the task by passing the * Command returned by the module to Wimp_StartTask. It then issues the service again, and repeats this until no-one claims it.

A module's service call handler should deal with this reason code as follows:

```
serviceCode
        LDR     R12, [R12]              ;Load workspace pointer
        STMFD   SP!, {LR}               ;Save link and make R14 available
        TEQ     R1, #Service_StartWimp  ;Is it service &49?
        BEQ     startWimp               ;Yes
        ...                             ;Otherwise try other services
        LDMFD   SP!, {PC}               ;Return

startWimp
        LDR     R14, taskHandle         ;Get task handle from workspace
        TEQ     R14, #0                 ;Am I already active?
        MOVEQ   R14, #-1                ;No, so init handle to -1
        STREQ   R14, taskHandle         ;R12 relative
        ADREQ   R0, myCommand           ;Point R0 at command to start task
        MOVEQ   R1, #0                  ;(see earlier) and claim the service
        LDMFD   SP!, {PC}               ;Return
```

Note that the `taskHandle` word of the module's workspace must be zero before the task has been started. This word should therefore be cleared in the module's initialisation code. If the task is not already running, the `startWimp` code should set the handle to –1, load the address of a command that can be used to start the module, and claim the call. Otherwise (if `taskHandle` is non-zero) it should ignore the call.

The automatic start-up process is made slightly more complex by the necessity to deal elegantly with errors that occur while a module is trying to start up. If the appropriate code is not executed, the Desktop can get into an infinite loop of trying to initialise unsuccessful modules.

This is avoided by the task setting its handle to –1 when it claims the StartWimp service. If the task fails to start, this will still be –1 the next time the Wimp issues a Service_StartWimp, and so it will not claim the service.

Service_StartedWimp (Service Call &4A)

Service_Reset (Service Call &27)

Request to task modules to set `taskHandle` variable to zero

On entry

R1 = &4A or &27 (reason codes)

On exit

Module's `taskHandle` variable set to zero

Use

A task which failed to initialise would have its `taskHandle` variable stuck at the value –1, which would prevent it from ever starting again (as Service_StartWimp would never be claimed). In order to avoid this, the two service calls above should be recognised by task modules. On either of them, the task handle should be set to zero:

```
serviceCode
        STMFD   sp!, {R14}
        LDR     R12, [R12]                      ;Get workspace pointer
...
        TEQ     R1, #Service_StartedWimp        ;Service &4A?
        BEQ     Service_StartedWimp
tryServiceReset
        TEQ     R1, #Service_Reset              ;Reset reason code?
        MOVEQ   R14, #0                         ;Yes, so zero handle
        STREQ   R14, taskHandle
        LDMFD   SP!, {PC}                       ;Return
...

        LDR     R14, taskHandle                 ;taskHandle = –1?
        CMN     R14, #1
        MOVEQ   R14, #0                         ;Yes, so zero it
        STREQ   R14, taskHandle
        LDMFD SP!, {PC}                         ;Return
```

Service_StartedWimp is issued when the last of the resident modules has been started, and Service_Reset is issued whenever the computer is soft reset.

Closing down

Generally a module task will terminate itself in the usual fashion by calling Wimp_CloseDown just before it calls OS_Exit. This might be in response to a Quit selection from a menu, or after a Message_Quit has been received. Modules also have finalisation entry point, and Wimp_CloseDown should be called from within this:

```
finalCode
        STMFD   sp!, {R14}
        LDR     R12, [R12]              ;Get workspace pointer
        LDR     R0, taskHandle          ;Check task is active
        TEQ     R0, #0
        LDRGT   R1, TASK                ;If so, close it down
        SWIGT   XWimp_CloseDown
        MOV     R1, #0                  ;always mark it as inactive
        STR     R1, taskHandle
;perform general finalisation code, possibly according to the value of R10
;(fatality indicator).
        LDMFD   sp!, {PC}               ;Return with V and R0 intact in case
                                        ;an error occurred
```

It is important that when Wimp_CloseDown is called from the finalise code, the task handle is quoted, as the module may not necessarily be the currently active Wimp task. Additionally, whenever Wimp_CloseDown is called, even outside of the finalisation code, the `taskHandle` variable should be cleared to zero.

Service_MouseTrap (Service Call &52)

The Wimp has detected a significant mouse movement

On entry

R0 = mouse x coordinate
R1 = &52 (reason code)
R2 = button state (from OS_Mouse)
R3 = time of mouse event (from OS_ReadMonotonicTime)
R4 = mouse y coordinate (NB R1 is already being used!)

On exit

All registers preserved

Use

It is possible to write programs which record changes in the mouse button state and pointer position. The recording can be played back later to simulate the effect of a human manipulating the mouse. This is very useful for setting up unattended demonstrations.

To save memory or disc space, such programs usually only record the mouse position when the button state changes, or after a certain time interval, e.g. ten times a second. Some Wimp events are dependent on a change of mouse position, not button state. It is therefore possible for a mouse recorder program to miss a critical mouse movement if it doesn't happen to choose the correct time to make its recording. The replay will then give different results from the original.

Service_MouseTrap is designed to overcome the problem. Whenever the Wimp detects a significant mouse movement, e.g. the pointer moving over a submenu right arrow, it issues this call. A mouse recorder should include the data in its output, in addition to any other mouse movements and button events that it would ordinarily log.

Programs which react to particular mouse movements (e.g. certain types of dragging) should themselves generate this event, where there is no mouse button transition.

A mouse recorder program should also trap INKEY of positive and negative numbers.

Service_WimpCloseDown (Service Call &53)

Notification that the Window Manager is about to close down a task

On entry

R0 = 0 if Wimp_CloseDown called (i) or
R0 > 0 if Wimp_Initialise called in task's domain (ii)
R1 = &53 (reason code)
R2 = handle of task being closed down, (i) and (ii)

On exit

R0 preserved (i) or (ii), or set to error pointer (ii)

Use

The Wimp passes this service around when someone calls Wimp_CloseDown. Usually a task knows that it has called Wimp_CloseDown, so this might not appear to be particularly informative. However, there are a couple of situations where the Wimp actually makes the call on a task's behalf. It is on these occasions that the service is useful.

- If a task calls OS_Exit without having called Wimp_CloseDown first, the Wimp does so on the task's behalf. This can arise when an error is generated that is not trapped by the task's error handler. The Wimp will report the error, then call OS_Exit for the task. The task should perform the operations it would have performed if it had called Wimp_CloseDown itself, and return preserving all registers. It must not call Wimp_CloseDown.
- A task might call Wimp_Initialise from within the same domain as the currently active task. For example, if a program allows the user to issue a * Command, the user might use it to try to start another Wimp task. The Wimp will try to close down the original task before starting the new one by issuing this service with R0>0.

If the original task does not want to be closed down, it should alter R0 so that it contains the pointer to a standard error block. The text 'Wimp is currently active' is regarded as a suitable message. (The task should compare the handle in R2 to its own to ensure that it is the task that is being asked to die.) The call should not be claimed, in order to allow others to receive the service, and R0 should not be altered except to point to an error.

If, on return from the service, R0 points to an error, the Wimp will return this to the new task trying to start up (it will also set the V flag). Thus, if the task is detecting errors correctly, it will abort its attempt to start up and call OS_Exit. This will happen if, for example, you try to start the Draw application from within a task window.

Service_WimpReportError (Service Call &57)

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

On entry

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

On exit

All registers preserved

Use

This service is provided so that certain tasks which usually trap VDU output (e.g. the VDU module) can be asked to suspend their activities temporarily while an error window is displayed.

If the state of the trapping module is ‘active’ and the service call is received with R0=1, the module should stop trapping and set its state to ‘suspended’. Similarly, if the state is suspended and the service is received with R0=0, the error window has disappeared and the module should re-enter the active state.

By taking note of this call, tasks running in an Edit window allow the standard filing system ‘up-call’ mechanism to continue operating, whereby users are asked to insert discs which the Filer cannot find in a drive.

Service_WimpSaveDesktop (Service Call &5C)

Save some state to a desktop boot file

On entry

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

On exit

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

Use

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

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

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

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

This call is not available under RISC OS 2.

Service_WimpPalette (Service Call &5D)

Palette change

On entry

R1 = &5D (reason code)

On exit

All register preserved

Use

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

This service call should not be claimed.

This call is not available under RISC OS 2.

Service_DesktopWelcome (Service Call &7C)

Desktop starting

On entry

R1 = &7C (reason code)

On exit

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

Use

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

This call is not available under RISC OS 2.

Service_ShutDown (Service Call &7E)

Switcher shutting down

On entry

R1 = &7E (reason code)

On exit

R1= 0 to claim and stop shutdown.

Use

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

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

This call is not available under RISC OS 2.

Service_ShutdownComplete (Service Call &80)

Shutdown completed

On entry

R1 = &80 (reason code)

On exit

This service call should not be claimed.

Use

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

This service call is not issued by RISC OS 2.

Service_WimpSpritesMoved (Service Call &85)

Wimp sprite pools have moved

On entry

R1 = &85 (reason code)
R2 = pointer to ROM area
R3 = pointer to RAM area

On exit

All registers preserved

Use

This service is provided if the sprite pools have to move. You must not claim it.

This service call is not issued by RISC OS 2.

Service_WimpRegisterFilters (Service Call &86)

Allows the Filter Manager to install filters with the Window Manager

On entry

R1 = &86 (reason code)

On exit

—

Use

When the Window Manager is reset this service call is issued to allow tasks to install filters with it. This is used by the Filter Manager to register itself.

This is issued when the Wimp resets the filter table back to its default state.

This service should not be used unless you are providing a replacement for the Filter Manager.

See the chapter entitled *The Filter Manager* on page 3-301 for more information on how to register filters for tasks.

This service call is not issued by RISC OS 2.

SWI Calls

In the following section, we list all of the SWI calls provided by the Window Manager module. It is possible to make some generalisations about the routines, though there are inevitably exceptions:

- R0 is often used to hold or return a handle, be it task, window or icon.
- All Wimp calls do not preserve R0.
- Other registers are preserved unless used to return results.
- Flags are preserved unless overflow is set on exit.
- R1 is used as a pointer to information blocks, e.g. window definitions, icon definitions, Wimp_Poll blocks.
- The contents of a Wimp_Poll block are usually correctly set up for the most obvious routine to call for the returned event code. For example, for an Open_Window_Request, the block will contain the information that Wimp_OpenWindow requires.
- All Wimp routines should not be executed with IRQs enabled due to the re-entrancy problems which may occur.
- Wimp routines may be called in User or SVC mode, except for Wimp_Poll, Wimp_PollIdle and Wimp_StartTask. These may only be called in User mode, as they rely on call-backs for their operation.
- As the Wimp uses the CallBack handler to do task swaps, it is not possible for a task to change the CallBack handler under interrupts. However language libraries can use the CallBack handler by setting it up when they start and using OS_SetCallBack (page 1-313)

The following SWIs can only operate on windows owned by the task that is active when the call is made, and will report the error `Access to window denied` if an attempt is made to access another task's window:

Wimp_CreateIcon	except in the icon bar
Wimp_DeleteWindow	
Wimp_DeleteIcon	except in the icon bar
Wimp_OpenWindow	send Open_Window_Request instead
Wimp_CloseWindow	send Close_Window_Request instead
Wimp_RedrawWindow	
Wimp_SetIconState	except in the icon bar
Wimp_UpdateWindow	
Wimp_GetRectangle	
Wimp_SetExtent	
Wimp_BlockCopy	

This also means that a task cannot access its own windows unless it is a ‘foreground’ process, i.e. it has not gained control by means of an interrupt routine, or is inside its module Terminate entry.

Wimp_Initialise (SWI &400C0)

Registers a task with the Wimp

On entry

R0 = last Wimp version number known to task × 100 (310 for RISC OS 3 applications)
R1 = 'TASK' (low byte = 'T', high byte = 'K', i.e. &4B534154)
R2 = pointer to short description of task, for use in Task Manager display
R3 = pointer to a list of message numbers terminated by a 0 word (not if R0 is less than 300). If Wimp version number is 310 then specifying 0 indicates that all messages are important to this task

On exit

R0 = current Wimp version number × 100
R1 = task handle

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call registers a task with the Wimp, and must be called once only when the task starts up. The following is done when the first task starts up and when a 'grubby' task exits (i.e. a task that starts from and returns to the Desktop but does not use it) and there are more tasks running.

- redefines some soft characters in the raanges &80 to &85 and &88 to &8B for the window system (dependent on the version of RISC OS)

- programs function, cursor, Tab and Escape key statuses, remembering their previous settings
- issues *Pointer to initialise the mouse and pointer system
- uses Wimp_SetMode to set the mode to the configured WimpMode, or to the last mode the Wimp used if this is different
- sets up the palette.

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

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

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

Related SWIs

Wimp_CloseDown (page 3-172)

Related vectors

None

Wimp_CreateWindow (SWI &400C1)

Tells the Wimp what the characteristics of a window are

On entry

R1 = pointer to window block

On exit

R0 = window handle

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call tells the Wimp what the characteristics of a window are. You should subsequently call Wimp_OpenWindow (page 3-109) to add it to the list of active windows (ones that are to be displayed). The format of a window block is as follows:

R1+0	visible area minimum x coordinate (inclusive)
R1+4	visible area minimum y coordinate (inclusive)
R1+8	visible area maximum x coordinate (exclusive)
R1+12	visible area maximum y coordinate (exclusive)
R1+16	scroll x offset relative to work area origin
R1+20	scroll y offset relative to work area origin
R1+24	handle to open window behind (–1 means top, –2 means bottom)
R1+28	window flags – see below
R1+32	title foreground and window frame colour – &FF means that the window has no control area or frame
R1+33	title background colour
R1+34	work area foreground colour

R1+35	work area background colour – &FF means ‘transparent’, so the Wimp won’t clear the rectangles during a redraw operation
R1+36	scroll bar outer colour
R1+37	scroll bar inner (Slider) colour
R1+38	title background colour when highlighted for input focus
R1+39	reserved – must be 0
R1+40	work area minimum x coordinate
R1+44	work area minimum y coordinate
R1+48	work area maximum x coordinate
R1+52	work area maximum y coordinate
R1+56	Title Bar icon flags – see below
R1+60	work area flags giving button type – see below
R1+64	sprite area control block pointer (+1 for Wimp sprite area)
R1+68	minimum width of window NB two-byte quantities
R1+70	minimum height of window 0,0 means use title width instead
R1+72	title data – see below
R1+84	number of icons in initial definition (can be 0)
R1+88	icon blocks, 32 bytes each – see Wimp_CreateIcon (page 3-93)

Note that the entries from R1+0 to R1+24 are not used unless Wimp_GetWindowState is called.

From RISC OS 3 onwards the Window extent is automatically rounded to be a whole number of pixels (and is re-rounded on a mode change).

Note: this call does not affect the screen unless the window handle is –2 (i.e. the iconbar). You must make a call to Wimp_ForceRedraw (page 3-147) to remove the icon(s) deleted, passing a bounding box containing the icons.

Fields requiring further explanation are:

Window flags

Window flags and status information are held in the word at offsets +28 to +31.

Bit	Meaning when set
0 *	window has a Title Bar
1	window is moveable, i.e. it can be dragged by the user
2 *	window has a vertical scroll bar
3 *	window has a horizontal scroll bar
4	window can be redrawn entirely by the Wimp, i.e. there are no user graphics in the work area. Redraw window requests won’t be generated if this bit is set
5	window is a pane, i.e. it is on top of a tool window
6	window can be opened (or dragged) outside the screen area (see also *Configure WimpFlags)

7 *	window has no Back icons or Close icons
8	a Scroll_Request event is returned when a mouse button is clicked on one of the arrow icons (with auto-repeat) or in the outer scroll bar region (no auto-repeat)
9	as above but no auto-repeat on the arrow icons
10	treat the window colours given as `GCOL` numbers instead of standard Wimp colours. This allows access to colours 0 - 254 in 256-colour modes (255 always has a special meaning)
11	don't allow any other windows to be opened below this one (used by the icon bar, and the backdrop for pre-RISC OS style applications)
12	generate events for 'hot keys' passed back through Wimp_ProcessKey if the window is open
13	forces window to stay on screen (not in RISC OS 2)
14	ignore right-hand extent if the size box of the window is dragged (not in RISC OS 2)
15	ignore lower extent if the size box of the window is dragged (not in RISC OS 2)

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

The five bits below are set by the Wimp and may be read using Wimp_GetWindowState (page 3-132).

Bit	Meaning when set
16	window is open
17	window is fully visible, i.e. not covered at all
18	window has been toggled to full size
19	the current Open_Window_Request was caused by a click on the Toggle Size icon
20	window has the input focus
21	force window to screen once on the next Open_Window

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

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

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

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

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

This bit is not supported in RISC OS 2.

Bit	Meaning when set
22 - 23	reserved; must be 0

The eight bits below provide an alternative way of determining which control icons a window has when it is created. If bit 31 is set, bits 24 to 30 determine the presence of one system icon, otherwise the ‘old style’ control icon flags noted above are used.

Bit	Meaning when set
24	window has a Back icon
25	window has a Close icon
26	window has a Title Bar
27	window has a Toggle Size icon
28	window has a vertical scroll bar
29	window has a Adjust Size icon
30	window has a horizontal scroll bar
31	use bits 24 - 30 to determine the control icons, otherwise use bits 0, 2, 3 and 7

A window may only have a quit and/or Back icon if it has a Title Bar, and a Size icon if it has one or two scroll bars. A Toggle Size icon needs a vertical scroll bar or a Title Bar. We recommend that new applications use the bit 31 set method of determining the control icons.

Bits 24 to 30 are also returned by Wimp_GetWindowState, updated to reflect what actually happened, so you can use this to ensure that the control icons used by the Wimp are as specified when the window was created, i.e. it was a valid specification.

Title bar flags

Title bar flags are held in the four bytes +56 to +59 of a window block. They correspond to the icon flags used in an icon block, described under Wimp_CreateIcon below. They determine how the contents of the Title Bar are derived and displayed. Note the following differences from proper icon flags though:

- The Title Bar always has a border, i.e. bit 2 is ignored.
- The title background is filled, i.e. bit 5 is ignored.
- The Wimp redraws the title, i.e. bit 7 is ignored.
- Any flags to do with button types, ESGs and selections are ignored. Dragging on the Title Bar always drags the window.
- If an anti-aliased font, or sprite, is used, you should bear in mind that the height of the Title Bar is fixed at 44 OS units, or 36 if you subtract the top and bottom frame lines. Thus only font sizes of about 10 to 12 points can be accommodated, and fairly small sprites. Also remember that lines will vary in width according to the screen mode used
- Bits 24 - 31 (when used as text colours) are ignored; the Title Bar colours are given in other window definition bytes.

So, the title may be text or a sprite, may be indirected (but not writable), use normal or anti-aliased text, and may be positioned within the Title Bar as required.

Title data

Title data is held in the twelve bytes at +72 to +83 of a window block. It has the same interpretation as the icon data bytes described under Wimp_CreateIcon. In summary:

- if text, then up to 12 bytes of text including a terminating control code
- if a sprite, then the name of the sprite (12 bytes)
- if the Title Bar is indirected, then the following three words: a pointer to a buffer containing the text, a pointer to a validation string (–1 if none), and the length of the buffer.

See the section on icon data under *Wimp_CreateIcon (SWI &400C2)* on page 3-93 for more details.

Window button types

The word at offset +26 in a window block is used to determine the ‘button type’ of the work area. Only bits 12 to 15 of this word are used. The 16 possible button types are much as described in the section on icon creation below. Note though that there is no concept of a window’s work area being ‘selected’ by the Wimp; the user is simply informed of button clicks through the Mouse_Click event.

Note that as stated previously, the button type only determines how Select and Adjust are handled; Menu is always reported. The interpretations of the button types for windows then are:

Bits 12 - 15	**Meaning**
0	ignore all clicks

1	notify task continually while pointer is over the work area
2	click notifies task (auto-repeat)
3	click notifies task (once only)
4	release over the work area notifies task
5	double click notifies task
6	as 3, but can also drag (returns button state * 16)
7	as 4, but can also drag (returns button state * 16)
8	as 5, but can also drag (returns button state * 16)
9	as 3
10	click returns button state*256 drag returns button state*16 double click returns button state*1
11	click returns button state drag returns button state*16
12 - 14	reserved
15	mouse clicks cause the window to gain the input focus.

Icons

The handles of any icons defined in this call are numbered from zero upwards, in the same order that they appear in the block. For details of the 32-byte definitions, see the next section.

Note: the Wimp_CreateWindow call may produce a `Bad work area extent` error if the visible area and scroll offsets combine to give a visible work area that does not lie totally within the work area extent.

Related SWIs

Wimp_DeleteWindow (page 3-105) Wimp_OpenWindow (page 3-109)

Related vectors

None

Wimp_CreateIcon (SWI &400C2)

Tells the Wimp what the characteristics of an icon are

On entry

R0 = icon handle or priority
R1 = pointer to block

On exit

R0 = icon handle

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call tells the Wimp what the characteristics of an icon are.

The block contains the following:

R1+0	window handle or: –1 for right of icon bar –2 left of icon bar The following are not available in RISC_OS 2: –3 ⇒ create icon on icon bar to left of icon handle R0 –4 ⇒ create icon on icon bar to right of icon handle R0 –5 ⇒ create icon on left side, scanning from the left –6 ⇒ create icon on left side, scanning from the right –7 ⇒ create icon on right side, scanning from the left –8 ⇒ create icon on right side, scanning from the right
R1+4	icon block

where an icon block is defined as:

+0	minimum x coordinate of icon bounding box
+4	minimum y coordinate of icon bounding box
+8	maximum x coordinate of icon bounding box
+12	maximum y coordinate of icon bounding box
+16	icon flags
+20	12 bytes of icon data

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

Icon blocks are also used in the Wimp_CreateWindow block and returned by Wimp_GetWindowInfo (page 3-134).

Once you have defined the icon, you can only make these changes to it:

- you can change its flags using the call Wimp_SetIconState (page 3-136).
- you can change indirected text. The icon must then be redrawn using the call Wimp_SetIconState, leaving the flags unchanged if necessary.
- you can change its text if its button type is 15 (writable). The Wimp does this for you automatically, handling the caret positioning and text updating. For further details, see the following sections:

  Wimp_SetCaretPosition (SWI &400D2) on page 3-149

  Wimp_GetCaretPosition (SWI &400D3) on page 3-151

  Wimp_Poll *Key_Pressed 8* event on page 3-119.

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

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

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

The following features are not available in RISC_OS 2:

The window handle at R1+0 can also be:

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

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

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

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

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

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

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

Window handle values	**Priority**
handle = –1	0
handle = –2	0
handle = –3, R0 = icon handle	same as matched icon
handle = –3, R0 = –1	&78000000
handle = –4, R0 = icon handle	same as matched icon
handle = –4, R0 = –1	&78000000

The various Desktop modules create icons with the following priorities:

Module	**Priority**
Task Manager	&60000000
!Help	&40000000
Palette Utility	&20000000
Applications	0
ADFS hard discs	&70000000
ADFS floppy discs	&60000000
‘Apps’ icon	&50000000
RAM disc	&40000000
Ethernet	&30000000
Econet	&20000000
Other filing systems	&10000000
Printer drivers	&0F000000
TinyDir	&0E000000

The icon block

The bounding box coordinates at R1+4 are given relative to the window's work area origin, except that the horizontal offset may be applied to an icon created on the icon bar. Note that if an icon is writable, the icon bounding box determines how much of the string is displayed at once. Typing into the icon or moving the caret left or right can cause the string to scroll within this box. The buffer length entry in the icon data determines the maximum number of characters that can be entered into a writable icon. One character is used for the terminator.

Note that icon strings can be terminated by any character from 0 to 31, and are preserved during editing operations by the Wimp. However, in template files, the terminator must be 13 (Return).

Icon flags

As noted earlier, subsets of these flags are used in Wimp_CreateWindow blocks to control how the contents of a window's Title Bar is defined, and the button type bits are used to determine how clicks within a window's work area are processed.

The full list of flags for a proper icon is:

Bit	Meaning when set
0	icon contains text
1	icon is a sprite
2	icon has a border
3	contents centred horizontally within the box
4	contents centred vertically within the box
5	icon has a filled background
6	text is an anti-aliased font (affects meaning of bits 24 - 31)
7	icon requires task's help to be redrawn
8	icon data is indirected
9	text is right-justified within the box
10	if selected with Adjust don't cancel others in the same ESG
11	display the sprite (if any) at half size
12 - 15	icon button type
16 - 20	exclusive selection group (ESG, 0 - 31)

21	icon is selected by the user and is inverted
22	icon cannot be selected by the mouse pointer; it is shaded
23	icon has been deleted
24 - 27	foreground colour of icon (if bit 6 is cleared)
28 - 31	background colour of icon (if bit 6 is cleared)
or	
24 - 31	font handle (if bit 6 is set). Font colours may be passed in an indirected icon's validation string.

Icon button types

These are much the same as window button types. However, icons can be 'selected' (inverted) by the Wimp automatically, so there are some additional effects to those already described for windows:

0	ignore mouse clicks or movements over the icon (except Menu)
1	notify task continuously while pointer is over this icon
2	click notifies task (auto-repeat)
3	click notifies task (once only)
4	click selects the icon; release over the icon notifies task; moving the pointer away deselects the icon
5	click selects; double click notifies task
6	as 3, but can also drag (returns button state * 16)
7	as 4, but can also drag (returns button state * 16) and moving away from the icon doesn't deselect it
8	as 5, but can also drag (returns button state * 16)
9	pointer over icon selects; moving away from icon deselects; click over icon notifies task ('menu' icon)
10	click returns button state*256 drag returns button state*16 double click returns button state*1
11	click selects icon and returns button state drag returns button state*16
12 - 13	reserved
14	clicks cause the icon to gain the caret and its parent window to become the input focus and can also drag (writable icon). For example, this is used by the FormEd application
15	clicks cause the icon to gain the caret and its parent window to become the input focus (writable icon)

All the above return Mouse_Click events (6), where the button state is:

Bit	Meaning when set
0	Adjust pressed
1	Menu pressed
2	Select pressed, or combination of above

A drag is initiated by the button being held down for more than about a fifth of a second. A double click is reported if the button is clicked twice in one second and the second click is within 16 OS units of the first. Note that button types which report double clicks will also report the initial click first.

Icon data

The icon data at +20 to +31 is interpreted according to the settings of three of the icon flags. The three bits are Indirected (bit 8), Sprite (bit 1) and Text (bit 0). The eight possible combinations and the eight interpretations of the icon data are:

IST	Meaning of 12 bytes/3 words
000	non-indirected, non-sprite, non-text icon
	+20 icon data not used in this case
001	non-indirected, text-only icon
	+20 the text string to be used for the icon, control-terminated
010	non-indirected, sprite-only icon
	+20 the sprite name to be used for the icon, control-terminated
011	non-indirected, text plus sprite icon
	+20 the text and sprite name to be used – not especially useful
100	indirected, non-sprite, non-text icon
	+20 icon data not used in this case
101	indirected, text-only icon
	+20 pointer to text buffer
	+24 pointer to validation string – see below
	+28 buffer length
110	indirected, sprite-only icon
	+20 pointer to sprite or to sprite name; see +28
	+24 pointer to sprite area control block, +1 for Wimp sprite area
	+28 0 if [+20] is a sprite pointer, length if it’s a sprite name pointer
111	indirected, text plus sprite icon
	+20 pointer to text buffer
	+24 pointer to validation string, which can contain sprite name
	+28 buffer length

Note that the icon bar's sprite area pointer is set to +1, so icons there use Wimp sprites. If you want to put an icon on the icon bar that isn't from the Wimp area, you must use an indirected sprite-only icon, type 110 above.

It is not possible to set the caret in the icon bar, so writable icons should not be used.

Validation Strings

An indirected text icon can have a validation string which is used to pass further information to the Wimp, such as what characters can be inserted directly into the string and which should be passed to the user via the Key_Pressed event for processing by the application. The syntax of a validation string is:

- validation-string ::= command { ; command }*
- command ::= a allow-spec | d char | f hex-digit hex-digit | l {decimal-number} | s text-string {,text-string} | r decimal-number {,decimal-number} | K (R|A|T|D|N)|P
- allow-spec ::= { char-spec }* { ~ { char-spec }* }*
- char-spec ::= char | char-char
- char ::= \– | \; | \\ | \~ | any character other than – ;

The spaces in the above definition are for clarity only, and a validation string will normally have no spaces in it.

In simple terms, a validation string consists of a series of 'commands', each starting with a single letter and separated from the following command by a semicolon. { }* means zero or more of the thing inside the { }. The following commands are available:

A command

The (A)llow command tells the Wimp which characters are to be allowed in the icon. Characters are inserted into the string if:

- a key is typed by the user
- the key returns a character code in the range 32 - 255
- the input focus is inside the icon
- the validation string allows the character within the string.

Otherwise:

- control keys such as the arrow keys and Delete are automatically dealt with by the Wimp
- other keys are returned to the task via the Key_Pressed event.

Each char-spec in the 'allow' string specifies a character or range of characters; the ~ character toggles whether they are included or excluded from the icon text string:

`A0-9a-z~dpu`	allows the digits 0 - 9 and the lower-case letters a - z, except for ‘d’, ‘p’ and ‘u’

If the first character following the A command is a ~ all normal characters are initially included:

`A~0-9`	allows all characters except for the digits 0 - 9

If you use any of the four special characters - ; ~ \ in a char-spec you must precede them with a backslash \:

`A~\-\;\~\\`	allows all characters except the four special ones - ; ~ \

D command

The (D)isplay command is used for password icons to avoid onlookers seeing what is typed. It is followed by a character that is used to echo all allowed characters:

`D*`	displays the password as a row of asterisks

Note that if the character is any of the four ‘special’ characters above, you must precede it by a \:

`D\–`	displays the password as a row of dashes

F command

The (F)ont colours command is used to specify the foreground and background colours used in text icons with an anti-aliased font. The F is followed by two hexadecimal digits, which specify the background and foreground Wimp colours respectively:

`Fa3`	sets background to 10 (&a hex), and foreground to 3.

This command uses the call Wimp_SetFontColours (page 3-218). If you do not use this command, the colours 0 and 7 (black on white) are used by default.

K command

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

Option	Action
R	If the icon is not the last icon in the window, pressing Return in the icon will move the caret to the beginning of the next writable icon in the window. If the icon is the last writable icon in the window then Return (code 13) will be passed to the application.
A	Pressing the up or down arrow keys will move the caret to the end of the next writable icon in the window. Pressing the up arrow key in the first writable icon in a window will move the caret to the last writable icon. Pressing the down arrow key in the last icon will move the caret to the first icon.
T	Pressing Tab in the icon will move the caret to the beginning of the next writable icon in the window. Pressing Shift-Tab will move the caret to the beginning of the previous writable icon in the window. The caret wraps around from last to first in the same way as in the A option.
D	Pressing any of Copy, Delete, Shift-Copy, Ctrl-U, or Ctrl-Copy will notify the application with the appropriate key codes as well as doing its defined action as specified in the section entitled *Key_Pressed 8* on page 3-119.
N	The application will be notified about all key presses in the icon, even if they are handled by the Wimp.

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

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

The (K)eys command is not available in RISC OS 2. In future releases of RISC OS this command will restrict the caret to icons in the same ESG group, rather than cycling through all icons.

L command

The (L)ine spacing command is used to tell the Wimp that a text icon may be formatted. If the text is too wide for the icon it is split over several lines. You should follow the L with a decimal number giving the vertical spacing between lines of text in OS units – if omitted, the default used is 40 units. (A system font character is 32 OS units high.)

The current version of RISC OS ignores the number following the L, so no number can be specified. However, this option may be implemented in future versions of RISC OS.

This option can only be used with icons which are horizontally and vertically centred, and do not contain an anti-aliased font. The icon must not be writable, since the caret would not be positioned correctly inside it.

P command

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

`P`*`spritename`*`,active_x,active_y` or

`P`*`spritename`*`;` coordinates default to (0, 0)

The sprites must be 4-colour sprites in the Wimp sprite area.

The (P)ointer command is not available in RISC OS 2

R command

The Bo(R)der command sets the border type for the icon. The border will only be drawn if the border bit for the icon is also set. This command will override the Wimp's default border for the icon.

`R` *`Type Slab_in_colour`*

`Type`
0 ⇒ normal single pixel border
1 ⇒ slab out
2 ⇒ slab in
3 ⇒ ridge
4 ⇒ channel
5 ⇒ action button (highlights when icon selected)
6 ⇒ default action button (highlights when icon selected)
7 ⇒ editable field
≥8 ⇒ normal single pixel border

`Slab_in_colour` relates to the highlight colour applied to border types 5 & 6. By default this is 14, but the validation string can over-ride this, when the icon is selected the foreground colour is retained and the background changes to the highlight colour.

The Bo(R)der command is not available in RISC OS 2, and does not correctly highlight fancy font icons under RISC OS 3.

S command

The (S)prite name command is used to give a text and sprite icon a different sprite name from the text it contains, for example, `Sfile_abc`. No space should follow the S, and the sprite name should be no more than 12 characters long.

If a second name is given, separated from the first by a comma, this is used when the icon is highlighted. If it is omitted, the sprite is highlighted by plotting it with its original colours exclusive-OR'ed with the icon foreground colour.

Text plus sprite icons

If an icon has both its text and sprite bits (0 and 1) set, then it will contain both objects. The text must be indirected, so that the validation string can be used to give the sprite name(s) to use (see the S command above).

Three flags in the icon flags are used to determine the relative positions of the text and sprite. These are the Horizontal, Vertical and Right justified bits (3, 4, and 9 respectively). The eight possible combinations of these bits, and how they position the sprite and text within the icon bounding box, are as follows:

HVR	Horizontal	Vertical
000	text and sprite left justified	text at bottom, sprite at top
001	text and sprite right justified	text at bottom, sprite at top
010	sprite at left, text +12 units right of it	text and sprite centred
011	text at left, sprite at right	text and sprite centred
100	text and sprite centred	text at bottom, sprite at top
101	text and sprite centred	text at top, sprite at bottom
110	text and sprite centred (text on top)	text and sprite centred
111	text at right, sprite at left	text and sprite centred

The following points should be noted about text plus sprite icons:

- the text part can be writable, but every time a key is pressed the sprite will be redrawn and so can flicker
- the text part of the icon always has its background filled
- if the text uses an anti-aliased font, the icon should not have a filled background, as the drawing of the text's background will obscure the sprite
- as usual, the whole of the icon area is used to delimit mouse clicks or movements over the icon, so clicks cannot be associated separately with the text and sprite (so clicking over the sprite would still cause the text of a writable icon to gain the caret)

An important use of this type of icon is displaying a text plus sprite pair in the icon bar.

Related SWIs

Wimp_DeleteIcon (page 3-107)

Related vectors

None

Wimp_DeleteWindow (SWI &400C3)

Closes a specified window if it is still open, and then removes its definition

On entry

R1 = pointer to block

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call closes the specified window if it is still open, and then removes the definition of the window and of all the icons within it. The memory used is re-allocated, except for the indirected data, which is in the task's own workspace.

The block contains the following:

R1+ 0	window handle

Errors

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

This error is not returned under RISC OS 2.

Related SWIs

Wimp_CreateWindow (page 3-87)

Related vectors

None

Wimp_DeleteIcon (SWI &400C4)

Removes the definition of a specified icon

On entry

R1 = pointer to block

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call removes the definition of the specified icon. If the icon is not the last one in its window's list it is marked as deleted, so that the handles of the other icons within the window are not altered. If the icon is the last one in the list, the memory is reallocated.

The block contains the following:

R1+ 0	window handle (–2 for icon bar)
R1+ 4	icon handle

Note: this call does not affect the screen unless the window handle is –2 (i.e. the icon bar). You must make a call to Wimp_ForceRedraw (page 3-147) to remove the icon(s) deleted, passing a bounding box containing the icons.

Related SWIs

Wimp_CreateIcon (page 3-93)

Related vectors

None

Wimp_OpenWindow (SWI &400C5)

Updates the list of active windows (ones that are to be displayed)

On entry

R1 = pointer to block

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call updates the list of active windows (ones that are to be displayed). The window may either be a new one being displayed for the first time, or an already open one that has had its parameters altered.

The block contains the following:

R1+ 0	window handle
R1+4	visible area minimum x coordinate
R1+8	visible area minimum y coordinate
R1+12	visible area maximum x coordinate
R1+16	visible area maximum y coordinate
R1+20	scroll x offset relative to work area origin
R1+24	scroll y offset relative to work area origin
R1+28	handle to open window behind –1 means top of window stack –2 means bottom –3 means the window behind the Wimp's backwindow, hiding it from sight (–3 not available in RISC OS 2)

Note that coordinates (x0,y0,x1,y1,scroll x,scroll y) are **all** rounded down to whole numbers of pixels. This also happens on a mode change automatically.

If a window that has the input focus is opened behind the backdrop (behind window –3) the input focus will be taken away from it before it is opened.

Related SWIs

Wimp_CloseWindow (page 3-111)

Related vectors

None

Wimp_CloseWindow (SWI &400C6)

Removes the specified window from the active list

On entry

R1 = pointer to block

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call removes the specified window from the active list; it is no longer marked as one to be displayed. The Wimp will issue redraw requests to other windows that were previously obscured by the closed one.

The block contains the following:

R1+ 0	window handle

Related SWIs

Wimp_OpenWindow (page 3-109)

Related vectors

None

Wimp_Poll (SWI &400C7)

Polls the Wimp to see whether certain events have occurred

On entry

R0 = mask
R1 = pointer to 256 byte block (used for return data)
R3 = pointer to poll word if R0 bit 22 set (not in RISC OS 2)

On exit

R0 = event code
R1 = pointer to block (data depends on event code returned)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call polls the Wimp to see whether certain events have occurred, and oversees such things as screen updating, keyboard and mouse handling, and menu selections. You must call it in the main loop of any program you write to run under the Wimp, and provide handlers for each event code it can return.

Errors

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

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

These errors are not returned under RISC OS 2.

The following event codes may be returned:

Code	Reason
0	Null_Reason_Code
1	Redraw_Window_Request
2	Open_Window_Request
3	Close_Window_Request
4	Pointer_Leaving_Window
5	Pointer_Entering_Window
6	Mouse_Click
7	User_Drag_Box
8	Key_Pressed
9	Menu_Selection
10	Scroll_Request
11	Lose_Caret
12	Gain_Caret
13	Poll word non-zero
14 - 16	reserved
17	User_Message
18	User_Message_Recorded
19	User_Message_Acknowledge

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

The remaining event codes are next and the lowest priority type is 0.

You can disable some of the event codes; they are neither checked for nor returned, and need not have handlers provided. You must do this for as many codes as possible, especially the Null_Reason_Code, if your task is to run efficiently under the Wimp. Some of the remaining event codes can be temporarily queued to prevent their return at times when they would otherwise interfere with the task running.

Both the above are done by setting bits in the mask passed in R0:

Bit	Meaning when set
0	do not return Null_Reason_Code
1	do not return Redraw_Window_Request; queue for later handling
2 - 3	must be 0
4	do not return Pointer_Leaving_Window
5	do not return Pointer_Entering_Window
6	do not return Mouse_Click; queue for later handling
7	must be 0
8	do not return Key_Pressed; queue for later handling
9 - 10	must be 0
11	do not return Lose_Caret
12	do not return Gain_Caret
13	do not return PollWord_NonZero (not in RISC OS 2)
14 - 16	must be 0
17	do not return User_Message
18	do not return User_Message_Recorded
19	do not return User_Message_Acknowledge
20 - 21	must be 0
22	R3 on entry is pointer to poll word (not in RISC OS 2)
23	scan poll word at high priority (not in RISC OS 2)
24	save or restore floating point registers (not in RISC OS 2)
25 - 31	must be 0

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

Saving floating point registers

If R0 bit 24 is set (not available in RISC OS 2) the floating point registers will be preserved over calls to Wimp_Poll.

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

- The task is controlling arbitrary applications ‘underneath’ it, which may use floating point instructions. An example of such a controlling task is the TaskWindow module.

- The task requires to set up a floating point status register value that is different from that used by the C run-time system (which happens to be &70000).

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

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

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

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

Event codes

As you can see, certain events cannot be masked out and the task must always be prepared to handle them. Each event code has one Wimp SWI that is most likely to be called in response. The block returned by Wimp_Poll is formatted ready to be passed directly to this call.

The event codes are as follows:

Null_Reason_Code 0

This event code is returned when none of the others are applicable. It should be masked out whenever possible to minimise the overheads incurred by the Wimp, so it doesn't have to set-up the task's memory and return control to it, only to find the task isn't interested anyway.

Redraw_Window_Request 1

The returned block contains:

R1+0	window handle

This event code indicates that some of the window is out of date and needs redrawing. You should call Wimp_RedrawWindow (page 3-126) using the returned block, and then call Wimp_GetRectangle (page 3-130) as necessary. See their entries for further details and a scheme of the code required.

Open_Window_Request 2

The returned block contains:

R1+0	window handle
R1+4	visible area minimum x coordinate
R1+8	visible area minimum y coordinate
R1+12	visible area maximum x coordinate
R1+16	visible area maximum y coordinate
R1+20	scroll x offset relative to work area origin
R1+24	scroll y offset relative to work area origin
R1+28	handle to open window behind (–1 means top of window stack, –2 means bottom)

This event code is returned as a result of the Adjust Size icon or the Title Bar of a window being selected, or as a result of the scroll bars being dragged to a new position. The dragging process is performed by the Wimp itself before it returns this event code to the task.

Following detection, the Wimp sets five bits that determine the action on the window. These bits can be read using Wimp_GetWindowState (page 3-132) – refer to Wimp_CreateWindow (page 3-87) for more information.

You should call Wimp_OpenWindow (page 3-109) using the returned block and also call it for any pane windows that are attached to this one, using the coordinates in the block to determine the pane's position.

Close_Window_Request 3

The returned block contains:

R1+0	window handle

This event code is returned when you click with the mouse on the Close icon of a window.

You should normally call Wimp_CloseWindow (page 3-111) using the returned block. You may also need to issue further calls of Wimp_CloseWindow to close any dependent windows, e.g. panes. However, if you do not want to close the window immediately, you could open an error box, or ask the user for confirmation.

Programs such as Edit conventionally open the directory which holds the edited file if its window is closed using the Adjust button. This is done by calling Wimp_GetPointerInfo when the Close_Window_Request is received, and performing the appropriate action.

Pointer_Leaving_Window 4

The returned block contains:

R1+0	window handle

This event code is returned when the pointer has left a window's visible work area. You might use it to make the pointer revert to its default shape when it is no longer over your window's work area. However, it is not recommended that you use it to make dialogue boxes disappear as soon as the mouse pointer leaves them.

Note that this event doesn't only occur when the pointer leaves the window's visible work area, but whenever the window stops being the most visible thing under the pointer. So, for example, popping up a menu at the pointer position would cause this event.

Pointer_Entering_Window 5

The returned block contains:

R1+0	window handle

This event code is returned when the pointer has moved onto a window. You might use it to bring a window to the top as soon as the pointer enters its work area, or to change the pointer shape when it over the visible work area.

As with the previous event type, Pointer_Entering_Window doesn't just happen when the pointer is physically moved into a window's visible work area. It could occur because a menu is removed or a window is closed, revealing a new uppermost window.

Mouse_Click 6

The returned block contains:

R1+0	mouse x (screen coordinates – not window relative)
R1+4	mouse y
R1+8	buttons (depending on window/icon button type)
R1+12	window handle (–1 for background, –2 for icon bar)
R1+16	icon handle (–1 for work area background)

This event code is returned when:

- the state of the mouse buttons has changed, and
- the conditions of the button type have been met, and
- the Wimp does not automatically deal with the change in some other way.

For example:

- if an icon has button type 6, a click with Select will generate this event with buttons = 4, whereas a drag with Adjust will give buttons = 1 followed by another event with buttons = 16
- if the change took place over a window's Close icon, this event code will not be returned as Close_Window_Request is used instead
- a click on the Menu button is always reported with buttons = 2.

The window and icon handles indicate which window and icon the mouse pointer was over when the button change took place. Operations such as highlighting an icon when it is selected and the cancellation of the other selections in the same ESG are all done automatically by the Wimp. See the section on *Icon button types* on page 3-97 for details of the various icon button modes and mouse return codes.

User_Drag_Box 7

The returned block contains:

R1+0	drag box minimum x coordinate (inclusive)
R1+4	drag box minimum y coordinate (inclusive)
R1+8	drag box maximum x coordinate (exclusive)
R1+12	drag box maximum y coordinate (exclusive)

This event code is returned when you release all the mouse buttons to finish a User_Drag operation. The block contains the final position of the drag box.

A user drag operation starts when the task calls Wimp_DragBox with a drag type of 5 to 11, usually in response to a drag code returned in a Mouse_Click event.

During the user drag operation (particularly with drag type 7), you may wish to keep track of the pointer position. To do this, call Wimp_GetPointerInfo (page 3-140) each time you receive a null event from Wimp_Poll. You can use the coordinates returned to redraw the dragged object (use Wimp_UpdateWindow (page 3-128) to do this).

When this event code is returned the drag is over; you should then stop reading the pointer information and, if appropriate, redraw the dragged object in its final position.

Key_Pressed 8

The returned block contains:

R1+0	window handle with input focus
R1+4	icon handle (–1 if none)
R1+8	x offset of caret (relative to window origin)
R1+12	y offset of caret (relative to window origin)
R1+16	caret height and flags (see Wimp_SetCaretPosition)
R1+20	index of caret into string (undefined if not in an icon)
R1+24	character code of key pressed (NB this is a word, not a byte)

This event code is returned to tell a task that a key has been pressed while the input focus belonged to one of its windows. The task should process the key if possible. Otherwise the task should pass it to Wimp_ProcessKey (page 3-170) so that other tasks can then intercept 'hot key' codes.

If the caret is inside a writable icon, the Wimp automatically processes the keys listed below, and does not generate an event:

Printable characters	are inserted into the text, if there is room, and the icon is redrawn
Delete, <–|	delete character to left of caret
Copy	delete character to right of caret
<–	move left one character
–>	move right one character
Shift Copy	delete word (forwards)
Shift <–	move left one word (returns &19C if at left of line)
Shift –>	move right one word (returns &19D if at right of line)
Ctrl Copy	delete forwards to end of line
Ctrl <–	move to left end of line
Ctrl –>	move to right end of line

'Printed characters' are those printable ones whose codes are in the ranges &20 - &7E and &80 - &FF.

See the K command on page 3-101 for further information.

Clashes could occur between top-bit-set characters (obtained by pressing Alt plus ASCII code on the keypad) and special key codes. The Wimp avoids any such ambiguities by mapping the special keys to these values:

Key	**Alone**	**+Shift**	**+Ctrl**	**+Ctrl Shift**
Esc	&1B	&1B	&1B	&1B
Print (F0)	&180	&190	&1A0	&1B0
F1 - F9	&181 - 189	&191 - 199	&1A1 - 1A9	&1B1 - 1B9
Tab	&18A	&19A	&1AA	&1BA
Copy	&18B	&19B	&1AB	&1BB

left arrow	&18C	&19C	&1AC	&1BC
right arrow	&18D	&19D	&1AD	&1BD
down arrow	&18E	&19E	&1AE	&1BE
up arrow	&18F	&19F	&1AF	&1BF
Page Down	&19E	&18E	&1BE	&1AE
Page Up	&19F	&18F	&1BF	&1AF
F10 - F12	&1CA - 1CC	&1DA - 1DC	&1EA - 1EC	&1FA - &1FC
Insert	&1CD	&1DD	&1ED	&1FD

These are set up by Wimp_Initialise. Tasks running under the Wimp are not allowed to change any of these settings. Soft key expansions (outside of writable icons) must be performed by the task accessing the key's expansion string using the `Key$n` variables.

Menu_Selection 9

The returned block contains:

R1+0	item in main menu which was selected (starting from 0)
R1+4	item in first submenu which was selected
R1+8	item in second submenu which was selected
	…
	terminated by –1

This event code is returned when the user selects an item from a menu. Selections can be made by the user clicking on an item with any of the mouse buttons. Select and Menu are synonymous; Adjust has a slightly different effect, as discussed below. A press of Return inside a writable menu item also generates this event (though not if it is pressed inside a writable icon inside a menu dialogue box).

The values in the block indicate which item at each menu level was chosen, the first item in each menu being numbered 0. An entry of –1 terminates the list. No handle is used for menus, so the task must remember which menu it last opened Wimp_CreateMenu (page 3-153) with.

If the last item specified has submenus (i.e. was not a 'leaf' of the menu tree) then the command may be ambiguous, in which case the task should ignore it. If the command is clear, but not its parameters, then the task may ignore the command, use default parameters, or use the last parameters set, as is most appropriate.

There is a difference, from the user's point of view, between choosing an item with Select and Adjust. In the former case, the selection will also cancel the menu, causing it to be removed from the screen. In the latter case, the menu should stay on the screen (a persistent menu). The application achieves this as follows. Call Wimp_GetPointerInfo (page 3-140) to read the mouse button state, and save it. After decoding the menu selection and taking the appropriate action, examine the stored button state. If Select was pressed, just return to the polling loop.

If Adjust was down, however, re-encode the menu tree (reflecting any changes that the previous menu selection effected) and call Wimp_CreateMenu with the same menu tree pointer that was used to create the menu in the first place. The next time you call Wimp_Poll, the Wimp will spot the re-opened menu, and recreate it on the screen. It goes down the tree until the end of the tree is reached, or the tree fails to correspond to the previous one, or until a shaded item is reached.

Scroll_Request 10

The returned block contains:

R1+0	window handle
R1+4	visible area minimum x coordinate
R1+8	visible area minimum y coordinate
R1+12	visible area maximum x coordinate
R1+16	visible area maximum y coordinate
R1+20	scroll x offset relative to work area origin
R1+24	scroll y offset relative to work area origin
R1+28	handle to open window behind (–1 means top of the window stack, –2 means bottom)
R1+32	scroll x direction
R1+36	scroll y direction

The scroll directions have the following meanings:

Value	**Meaning**
–2	Page left/down (click in scroll bar outer area)
–1	Left/down (click on scroll arrow)
0	No change
+1	Right/up (click on scroll arrow)
+2	Page right/up (click in scroll bar outer area)

This event code is returned if the user clicks in a scroll area of a window which has one of the ‘Scroll_Request returned’ bits set in its window flags. It returns the old scroll bar offsets and the direction of scrolling requested. The task should work out the new scroll offsets, store them in the scroll offsets (R1+20 and R1+24) of the returned block, and then call Wimp_OpenWindow (page 3-109).

Remember that the coordinates used for scroll offsets are in OS units. Therefore, if you want to make a click on one of the arrows scroll by, say, one pixel, you must scale the –1 or 1 returned in the event block by the appropriate factor for the current mode. For example, in !Edit the text is aligned with the bottom of the window when scrolling down, and subsequently moves down by one text line exactly. When scrolling up, the text is aligned with the top of the window.

Lose_Caret 11

This is returned when the window which owns the input focus has changed. That happens when Wimp_SetCaretPosition (page 3-149) is called, either explicitly, or implicitly by the user clicking on a button type 14 or 15 object. The event isn't generated if the input focus only changes position within the same window.

The event warns the task which had the caret (and which may well be retaining it) that something has changed. It can be used to remove a specialised text-position indicator which does not use the Wimp's caret, or its appearance could be altered to show this is where the caret would be if the window still had the input focus.

R1 points to a standard caret block:

R1+0	window handle that had the input focus (–1 if none)
R1+4	icon handle (–1 if none)
R1+8	x offset of caret (relative to window origin)
R1+12	y offset of caret (relative to window origin)
R1+16	caret height and flags (see Wimp_SetCaretPosition)
R1+20	index of caret into string (or –1 if not in a writable icon)

Gain_Caret 12

This event is returned to the task which now has the caret, subsequent to a Wimp_SetCaretPosition. The block pointed to by R1 is the same as above, except that the window/icon handle is the caret's new owner.

PollWord_NonZero 13

This facility is not available under RISC OS 2.

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

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

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

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

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

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

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

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

Events 14 - 16: not used

Messages

The next three event codes (17 - 19) are concerned with the receipt of user messages. Events of type 0 to 12 are normally sent directly from the Wimp to a task in response to some user action. The User_Message event codes are more general purpose, and are sent from Wimp to task, or from task to task. See the description of Wimp_SendMessage (page 3-193) and the section entitled *Messages* on page 3-228 for more details about the sending of messages and of the various types of User_Message actions which are defined.

One message action that all tasks should act on is Message_Quit, which is broadcast by the Desktop when the user selects the Exit item from the Task manager’s Task display.

User_Message 17

The returned block contains:

R1+0	size of block in bytes (20 - 256 in a multiple of four (i.e. words))
R1+4	task handle of message sender
R1+8	my_ref – the sender's reference for this message
R1+12	your_ref – a previous message's my_ref, or 0 if this isn't a reply
R1+16	message action code
R1+20	message data (dependent on message action)
	…

This event is returned when another task has sent a message to the current task, to one of its windows, or to all tasks using a broadcast message. The action code field defines the meaning of the message, i.e. how the message data should be processed by the receiver.

If the message is not acknowledged (because the receiving task is no longer active, or just ignores it) then no further action is taken by the Wimp.

User_Message_Recorded 18

The block has the same format as that described above under User_Message. The interpretation of the message action is the same, so the way in which the receiving task handles these two types should be identical. However, the way the Wimp responds differs if the message is not acknowledged.

The receiving task can acknowledge the message by calling Wimp_SendMessage with the event code User_Message_Acknowledge (19) and the your_ref field set to the my_ref of the original. This will prevent the sender from receiving its original message back from the Wimp with the event type 19.

Another way to acknowledge a message (and prevent the Wimp returning it to the sender) is to send a reply message using event code User_Message or User_Message_Acknowledge, again with the your_ref field set to the original message's my_ref.

Both types of acknowledgement must take place before the next call to Wimp_Poll.

User_Message_Acknowledge 19

The format of the block is as above. This event type is generated by the Wimp when a message sent with event code User_Message_Recorded was not acknowledged or replied to by the receiver. The message in the block is identical to the one sent by the task in the first place.

Note that in User_Messages 17, 18 and 19 a task should ignore any messages it does not understand: it must not acknowledge messages as a matter of course. See Wimp_SendMessage (page 3-193) for details.

Related SWIs

Wimp_PollIdle (page 3-181)

Related vectors

None

Wimp_RedrawWindow (SWI &400C8)

Starts a redraw of the parts of a window that are not up to date

On entry

R1 = pointer to block

On exit

R0 = 0 for no more to do, non-zero for update according to returned block

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The block contains the following:

R1+0	window handle
R1+4	visible area minimum x coordinate
R1+8	visible area minimum y coordinate
R1+12	visible area maximum x coordinate
R1+16	visible area maximum y coordinate
R1+20	scroll x offset relative to work area origin
R1+24	scroll y offset relative to work area origin
R1+28	current graphics window minimum x coordinate
R1+32	current graphics window minimum y coordinate
R1+36	current graphics window maximum x coordinate
R1+40	current graphics window maximum y coordinate

The window handle at +0 is set on entry, usually from the last call to Wimp_Poll; the rest of the block is filled in by Wimp_RedrawWindow.

Note that this SWI must be called as the first Wimp operation after the Wimp_Poll which returned a Redraw_Window_Request. This means that you cannot, for example, delete or create any other windows between the Wimp_Poll and the Wimp_RedrawWindow. If you need to do any special extra operations in your Wimp_Poll loop, do them just before calling Wimp_Poll, not afterwards.

This call is used to start a redraw of the parts of a window that are not up to date. These consist of a series of non-overlapping rectangles. Wimp_RedrawWindow draws the window outline, issues `VDU 5`, and then exits via Wimp_GetRectangle, which returns the coordinates of the first invalid rectangle (if any) of the work area, and clears it to the window's background colour, unless it's transparent. It also returns a flag saying whether there is anything to redraw.

The first four words are the position of the window's work area on the screen, i.e. they have the same meaning as those words in the Wimp_CreateWindow (page 3-87) and Wimp_OpenWindow (page 3-109) blocks.

The last four words describe an area within the visible work area in screen coordinates, not work area relative, possibly the whole thing if the window is not covered. The graphics clip window is set to the returned rectangle. A task could just redraw its entire work area each time a rectangle is returned. However, it is much more efficient if the task takes note of the graphics clip window coordinates and works out what it needs to draw.

By using these two sets of coordinates in conjunction with the scroll offsets, you can find the work area coordinates to be updated:

work x = screen x – (screen x0–scroll x)
work y = screen y – (screen y1–scroll y)

where:

screen x0 =	[R1+4]
screen y1 =	[R1+16]
scroll x =	[R1+20]
scroll y =	[R1+24]

The code used to redraw the window was outlined in the section entitled *Redrawing windows* on page 3-18. The expressions above in parenthesis are the screen coordinates of the work area origin.

Related SWIs

Wimp_UpdateWindow (page 3-128), Wimp_GetRectangle (page 3-130)

Related vectors

None

Wimp_UpdateWindow (SWI &400C9)

Starts a redraw of the parts of a window that are not up to date

On entry

R1 = pointer to block – see below

On exit

R0 and block as for Wimp_RedrawWindow (page 3-126)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The block contains the following on entry:

R1+0	window handle
R1+4	work area minimum x coordinate (inclusive)
R1+8	work area minimum y coordinate (inclusive)
R1+12	work area maximum x coordinate (exclusive)
R1+16	work area maximum y coordinate (exclusive)

This call is similar to Wimp_RedrawWindow. The differences are:

- not all of the window has to be updated; you specify the rectangle of interest in work area coordinates
- the rectangles to be updated are not cleared by the Wimp first
- this can be called at any time, not just in response to a Redraw_Window_Request event.

The routine exits via Wimp_GetRectangle (page 3-130), which returns the coordinates of the first visible rectangle (if any) within the work area specified on entry.

The code for the task to update the window should follow this scheme:

```
SYS"Wimp_UpdateWindow",,blk TO more
WHILE more
  update the contents of the returned rectangle
  SYS"Wimp_GetRectangle",,blk TO more
ENDWHILE
```

A common reason for calling this is to drag an item across a window. Another is to draw a user-defined text cursor instead of using the system one.

Related SWIs

Wimp_RedrawWindow (page 3-126), Wimp_GetRectangle (page 3-130), Wimp_ForceRedraw (page 3-147)

Related vectors

None

Wimp_GetRectangle (SWI &400CA)

Returns the details of the next rectangle of the work area to be drawn

On entry

R1 = pointer to block

On exit

R0 and block as for Wimp_RedrawWindow (page 3-126)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is used repeatedly following a call of either Wimp_RedrawWindow or Wimp_UpdateWindow. It returns the details of the next rectangle of the work area to be drawn (if any). If the call follows an earlier call to Wimp_RedrawWindow, then the rectangle is also cleared to the background colour of the window. If however it follows a call to Wimp_UpdateWindow then the rectangle's contents are preserved.

The block contains the following on entry:

R1+0	window handle

VDU 5 is asserted at a mode change and in Wimp_RedrawWindow. If you use VDU 4 text in a window (which can only be done when you are sure that the character does not need to be clipped) you should reset to VDU 5 mode before calling Wimp_SetRectangle or Wimp_Poll.

Note that the window handle will be faulted by the Wimp if it differs from the one last used when Wimp_RedrawWindow or Wimp_UpdateWindow was called. This means that a task must draw the whole of a window before performing any other operations.

Related SWIs

Wimp_RedrawWindow (page 3-126), Wimp_UpdateWindow (page 3-128)

Related vectors

None

Wimp_GetWindowState (SWI &400CB)

Returns a summary of the given window's state

On entry

R1 = pointer to block

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call returns a summary of the given window's state.

The block contains the window handle on entry, and the following on exit:

R1+0	window handle (or –2 to indicate the icon bar)
R1+4	visible area minimum x coordinate
R1+8	visible area minimum y coordinate
R1+12	visible area maximum x coordinate
R1+16	visible area maximum y coordinate
R1+20	scroll x offset relative to work area origin
R1+24	scroll y offset relative to work area origin
R1+28	handle of window in front of this one (or –1 if none)
R1+32	window flags – see Wimp_CreateWindow (page 3-87)

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

You can usually find out the window's coordinates without using this call, since Wimp_GetRectangle returns the window coordinates anyway. This call is most useful for reading the window flags, for example to find out if a window is uncovered.

Related SWIs

Wimp_GetWindowInfo (page 3-134)

Related vectors

None

Wimp_GetWindowInfo (SWI &400CC)

Returns complete details of the given window's state

On entry

R1 = pointer to block (in RISC OS 2), else in RISC OS 3:
- bit 0 set ⇒ just return window header (without icons)
- bit 1 reserved (must be 0)
- bits 2 - 31 pointer to buffer to receive data

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call returns complete details of the given window's state, including any icons that were created after the window, using Wimp_CreateIcon.

The block contains the following on entry:

R1+0	window handle (or –2 to indicate the icon bar)

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

The block contains the following on exit:

R1+0	window handle
R1+4	window block – see Wimp_CreateWindow (page 3-87) and Wimp_CreateIcon (page 3-93)

Related SWIs

Wimp_GetWindowState (page 3-132)

Related vectors

None

Wimp_SetIconState (SWI &400CD)

Sets a given icon's state held in its flags word

On entry

R1 = pointer to block

On exit

R0 corrupted
The icon's flags are updated

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the given icon's state held in its flag word as follows:

```
new-state = (old-state AND NOT clear-word) EOR EOR-word
```

The block contains the following:

R1+0	window handle (–1 or –2 for icon bar)
R1+4	icon handle
R1+8	EOR word
R1+12	clear word

The way each bit of the icon flags is affected is controlled by the state of the corresponding bits in the **E**OR word and the **C**lear word:

Value of CE	Effect
00	preserve the bit's status
01	toggle the bit's state
10	clear the bit
11	set the bit

For example, say you wanted to change an icon's button type (bits 12 - 15) to 10 (%1010 binary). You would set the clear-bits to 1 and the EOR bits to the new value:

Clear = %1111000000000000
EOR = %1010000000000000

The screen is automatically updated if necessary, so the call can be used to reflect a change in a text icon's contents. If you change the justification of a text icon using this call, and the icon owns the caret, you should also call Wimp_SetCaretPosition (page 3-149) to make sure that it remains positioned in the text correctly.

Related SWIs

Wimp_GetIconState (page 3-138)

Related vectors

None

Wimp_GetIconState (SWI &400CE)

Returns a given icon's state from its flags word

On entry

R1 = pointer to block

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call returns the given icon's state from its flags word.

On entry the block contains the following:

R1+ 0	window handle
R1+ 4	icon handle

On exit the block contains the following:

R1+0	window handle
R1+4	icon handle
R1+8	32 byte icon block – see Wimp_CreateIcon (page 3-93)

If you want to search for an icon with particular flag settings (for example to find out which icon in a group has been selected), you should use Wimp_WhichIcon (page 3-159).

Related SWIs

Wimp_SetIconState (page 3-136)

Related vectors

None

Wimp_GetPointerInfo (SWI &400CF)

Returns the position of the pointer and the state of the mouse buttons

On entry

R1 = pointer to block

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call returns information about the position of the pointer and the instantaneous state of the mouse buttons. It enables the task to find out where the mouse pointer is independently of the buttons being pressed or released, for example for dragging purposes.

On exit the block contains the following:

R1+0	mouse x
R1+4	mouse y
R1+8	button state
R1+12	window handle (–1 for background, –2 for icon bar)
R1+16	icon handle (see below)

The mouse button state (returned in R1+8 to R1+11) can only have bits 0, 1 and 2 set:

Bit	Meaning if set
0	Right-hand button pressed (Adjust)
1	Middle button pressed (Menu)
2	Lefthand button pressed (Select)

If the mouse is over a user window (window handle ≥0) then the icon handle will be either a valid non-negative value for a user icon, or one of the following system values:

Value	Icon
–1	work area
–2	Back icon
–3	Close icon
–4	Title Bar
–5	Toggle Size icon
–6	scroll up arrow
–7	vertical scroll bar
–8	scroll down arrow
–9	Adjust Size icon
–10	scroll left arrow
–11	horizontal scroll bar
–12	scroll right arrow
–13	the outer window frame

From RISC OS 3 onwards shaded icons in menus are treated differently from normal shaded icons, in that the latter are treated as being ‘invisible’ to the Wimp, i.e. Wimp_GetPointerInfo will never return them. In menus, however, the icons are not invisible, but are not allowed to be selected. This allows the interactive help program to see the icons and to ask for help on them.

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

Related SWIs

None

Related vectors

None

Wimp_DragBox (SWI &400D0)

Initiates a dragging operation

On entry

R1 <= 0 to cancel drag operation, otherwise
R1 = pointer to block

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call initiates a dragging operation. It is typically called as a result of a Mouse_Click event which has reported a drag-type click (i.e. Select or Adjust held down for longer than about 1/5th of a second). A drag spans calls to Wimp_Poll, so the task must maintain information about what is being dragged, etc. Usually the coordinates are not required until the final drag event occurs, at which point the Wimp returns them. Sometimes Wimp_GetPointerInfo should be called in Wimp_Poll null events to track the pointer (especially for type 7 below). A drag is terminated (and reported) when the user releases all of the mouse buttons.

On entry the block contains the following:

R1+ 0	window handle (or –2 to indicate the icon bar) for drag types 1 - 4 only
R1+ 4	drag type
R1+ 8	minimum x coordinate of initial position of drag box
R1+12	minimum y coordinate of initial position of drag box
R1+16	maximum x coordinate of initial position of drag box
R1+20	maximum y coordinate of initial position of drag box
R1+24	minimum x coordinate of parent box (for types 5 - 11 only)
R1+28	minimum y coordinate of parent box (for types 5 - 11 only)
R1+32	maximum x coordinate of parent box (for types 5 - 11 only)
R1+36	maximum y coordinate of parent box (for types 5 - 11 only)
R1+40	R12 value for user routine (for types 8 - 11 only)
R1+44	address of draw box routine (for types 8 - 11 only)
R1+48	address of remove box routine (for types 8 - 11 only)
R1+52	address of move box routine, or <= 0 if there isn't one (for types 8 - 11 only)

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

The coordinates are passed as screen coordinates, i.e. bottom-left inclusive and top-right exclusive.

The drag is confined to the 'parent box' specified, or to an area computed by the Wimp for types 1 - 4 and 12. The action depends on the drag type:

Drag type	**Meaning**
1	drag window position
2	drag window size
3	drag horizontal scroll bar
4	drag vertical scroll bar
5	drag fixed size 'rotating dash' box
6	drag rubber 'rotating dash' box
7	drag point (no Wimp-drawn dragged object)
8	drag fixed size user-drawn box
9	drag rubber user-drawn box
10	as 8 but don't cancel when buttons are released
11	as 9 but don't cancel when buttons are released
12	drag horizontal and vertical scroll bars (not in RISC OS 2)

Types 1 - 4

These are the ‘system’ types since they relate to picking up a window, changing its size and scrolling it respectively. In these cases, the bounding box for pointer movement is worked out automatically by the Wimp. For example, type 2 drags are confined to the defined maximum and minimum sizes of the window.

Bits in the `WimpFlags` CMOS configuration parameter determine the way in which these drags update the screen. There are four bits, 0 - 3, corresponding to drag types 1 - 4. If the bit is clear, then dragging is indicated by a dashed outline box, similar to that used in types 5 and 6 below. An Open_Window_Request event is generated when the mouse button is released to allow the task to update appropriate parts of the dragged window. If the `WimpFlags` bit is set, continuous update is required, and Open_Window_Requests are generated for every mouse move.

These drag types are useful if you want to allow the user to, for example, pick up a window which does not have a Title Bar (and so is usually unmovable). You could detect clicks in a region of within, say, 32 OS units from the top of the visible work area and instigate a drag type 1 when these occur.

Types 5 - 7

These are ‘user’ types, where the task decides what the significance of the dragging will be. In these cases you supply the coordinates of the parent box. The box being dragged is constrained to this area. For types 5 and 6 the initial box position is used to draw a box with a dashed border which cycles round.

For type 5 boxes, the relative positions of the mouse pointer and the box are kept constant, so moving the mouse moves the box too.

For type 6, the relative positions of the bottom right corner of the box and the pointer are kept constant, so moving the mouse will increase or decrease the size of the box. Generally you would arrange the initial box coordinates such that this corner is at or near the pointer position reported in the drag-click event. You can alter the moveable corner to the left by reversing the initial x coordinates, and to the top by reversing the initial y coordinates.

In the case of type 7, where there is no dashed box to be dragged, the initial drag box position is ignored and the mouse coordinates are constrained to the bounding box.

Types 8 - 11

These types give the maximum flexibility for dragging objects around the whole screen. Use drag type 7 and Wimp_UpdateWindow to drag an object within a window. They are, though, somewhat more complex to use than the previously described types.

First the application must provide the addresses of three routines which draw, remove and move the user's drag item (it doesn't have to be a box). If no move routine is supplied ([R1+52] ≤ 0), the Wimp will use the remove and draw routines to perform the operation.

Note that the user code must not be in application space, but in the RMA. This is because the Wimp doesn't know to page the task in when this code is required.

The user code is called under the following conditions:

On entry

SVC mode (so use X-type SWIs and save R14_SVC before hand)
R0 = new minimum x coordinate
R1 = new minimum y coordinate
R2 = new maximum x coordinate
R3 = new maximum y coordinate
R4 = old minimum x coordinate (for move routine only)
R5 = old minimum y coordinate (for move routine only)
R6 = old maximum x coordinate (for move routine only)
R7 = old maximum y coordinate (for move routine only)
R12 = value supplied in Wimp_DragBox call

On exit

R0 - R3 actual box coordinates (normally preserved from entry)

The user routines would draw, remove or just move (i.e. remove and redraw) their drag object according to the coordinates passed. These coordinates are derived by the Wimp from mouse movements.

The graphics window is also set up by the Wimp. The user routines must not change this, or draw outside it.

While these drags are taking place, the Wimp still performs its rotating dashed box code, so the routines can take advantage of this. Programming of the VDU dot-dash pattern is performed by the Wimp, so all the user routines have to do is call the appropriate dot-dash line `PLOT` codes.

The move routine has to deal with two cases: whether the box has moved or not. If the box has moved (i.e. R0 - R3 are not identical to R4 - R7), then the move routine must exclusive-OR once using the old coordinates to remove the box, then EOR again with the new coordinates to redraw it. If the box hasn't changed, the Wimp will have programmed the dot-dash pattern so that a single EOR plot will give the desired shifting effect of the pattern, so this is what the routine should do.

Of course, the foregoing is only applicable to dragged objects which use the dash effect. If you are dragging, say, a sprite, then the move routine only has to do anything when the coordinates have changed, viz restore the background that the sprite overwrote, then save the new background and replot the sprite. When no move has taken place, the routine could do nothing (or change the sprite for an animation effect etc.)

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

Type 12

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

This type is not available in RISC OS 2.

Related SWIs

None

Related vectors

None

Wimp_ForceRedraw (SWI &400D1)

Forces an area of a window or the screen to be redrawn later

On entry

R0 = window handle (–1 means whole screen, –2 indicates the icon bar)
R1 = minimum x coordinate of area to redraw
R2 = minimum y coordinate of area to redraw
R3 = maximum x coordinate of area to redraw
R4 = maximum y coordinate of area to redraw

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call forces an area of a window or the screen to be marked as invalid, and to be redrawn later using Redraw_Window_Request events.

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

If R0 is –1 on entry, then R1 - R4 specify an area of the screen in absolute coordinates. If R0 is not –1, then it indicates a window handle, and R1 - R4 specify an area of the window relative to the window's work area origin.

This call could be used

- to reconstruct the screen if for some reason it has been corrupted
- to reinstate a particular area after, for example, an error box has been drawn over the top of it
- to redraw the screen after redefining one or more of the soft characters, which could affect any part of the screen.

Two strategies are possible when the task is required to change the contents of a window. These are:

- call this routine, which causes the specified area to be redrawn later
- call Wimp_UpdateWindow (page 3-128), followed by the necessary graphic operations (and calls to Wimp_GetRectangle (page 3-130)).

The second method is generally quicker, but involves more code.

Related SWIs

Wimp_RedrawWindow (page 3-126), Wimp_UpdateWindow (page 3-128), Wimp_GetRectangle (page 3-130)

Related vectors

None

Wimp_SetCaretPosition (SWI &400D2)

Sets up the data for a new caret position, and redraws it there

On entry

R0 = window handle (–1 to turn off and disown the caret)
R1 = icon handle (–1 if none)
R2 = x offset of caret (relative to work area origin)
R3 = y offset of caret (relative to work area origin)
R4 = height of caret (if –1, then R2, R3, R4 are calculated from R0,R1,R5)
R5 = index into string (if –1, then R4, R5 are calculated from R0,R1,R2,R3
R2 and R3 are modified to exact position in icon)

On exit

R0 - R5 = preserved

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call removes the caret from its old position, sets up the data for its new position, and redraws it there. Subsequent calls to Wimp_RedrawWindow and Wimp_UpdateWindow will cause the caret to be automatically redrawn by the Wimp, unless it is marked as invisible.

R4 and R5 can only be set to –1 if the icon handle passed in R1 is non-negative.

Some of the values may be calculated:

- If R4 (the height) is –1, the Wimp calculates the x and y coordinates of the caret and its height (R2, R3, R4) from the data in R0, R1 and R5. This is only possible if R1 contains an icon handle.
- Similarly, if R5 (the index) is –1, the Wimp calculates the index into the string and the caret height (R4, R5) from R0 - R3.

In each case, the height of the caret is determined from the bounding box of the font used in the icon (for the system font, a height of 40 OS units is used). The caret's coordinates refer to the pixel at the bottom of the vertical bar. Note that the icon's bounding box and whether it has an outline are also considered.

The font height also contains some flags. Its full description is:

bits 0 - 15	height in OS units (0 - 65535)
bits 16 - 23	colour (if bit 26 is set)

Bit	Meaning when set
24	use VDU 5-type caret, else use anti-aliased caret
25	the caret is invisible
26	use bits 16 - 23 for the colour, else caret is Wimp colour 11
27	bits 16 - 23 are untranslated, else they are a Wimp colour

If bit 27 is set, then bit 26 must be set and the caret is plotted by EORing the logical colour given in bits 16 - 23 onto the screen. For the 256-colour modes, bits 16 - 17 are bits 6 - 7 of the tint, and bits 18 - 23 are the colour.

If bit 27 is clear, then the caret is plotted such that the Wimp colour given (or colour 11) appears when the background is Wimp colour 0 (white). The Wimp achieves this by EORing the actual colour for Wimp colour 0 and the caret colour together, then EORing this onto the screen.

Esoteric note: to ensure that the caret is plotted in a given colour on a non-white background, you must do the following:

- use Wimp_ReadPalette (page 3-189) to obtain the real logical colours associated with your background and caret (byte 0 of the entries)
- EOR these together
- put the result in bits 16 - 23 and set bits 26 and 27.

Related SWIs

Wimp_GetCaretPosition (page 3-151)

Related vectors

None

Wimp_GetCaretPosition (SWI &400D3)

Returns details of the caret's state

On entry

R1 = pointer to block

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call returns details of the caret's state. The block contains the following:

R1+0	window handle where caret is (–1 if none)
R1+4	icon handle (–1 if none)
R1+8	x offset of caret (relative to work area origin)
R1+12	y offset of caret (relative to work area origin)
R1+16	caret height and flags or –1 for not displayed
R1+20	index of caret into string (if in a writable icon)

The height and flags returned at R1+16 are as described under Wimp_SetCaretPosition (page 3-149).

Related SWIs

Wimp_SetCaretPosition (page 3-149)

Related vectors

None

Wimp_CreateMenu (SWI &400D4)

Creates a menu structure

On entry

R1 = –1 means close any active menu, or
R1 = pointer to menu block (or window handle)
R2 = x coordinate of top-left corner of top level menu
R3 = y coordinate of top-left corner of top level menu

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is used to create a menu structure. The top level menu is initially displayed by the Wimp. Having made this call, the task must return to its normal polling loop. While the task calls Wimp_Poll, the Wimp maintains the menu tree, until the user clicks with any of the mouse buttons. If the click was outside the menus, then the Wimp closes all the menus and behaves as if they had not been there. If the mouse is clicked inside a menu, then a Menu_Selection event code is returned from Wimp_Poll, along with a list of selections.

The menu block contains the following:

R1+ 0	menu title (if a null string, then the menu is untitled)
R1+12	menu title foreground and frame colour
R1+13	menu title background colour
R1+14	menu work area foreground colour
R1+15	menu work area background colour
R1+16	width of following menu items
R1+20	height of following menu items
R1+24	vertical gap between items
R1+28	menu items (each 24 bytes):

bytes 0 - 3	menu flags:

Bit	Meaning when set
0	display a tick to the left of the item
1	dotted line following (separates sections)
2	item is writable for text entry
3	generate a message when moving to the submenu
4	allow submenu to be opened even if this item is greyed out (not in RISC OS 2)
7	this is the last item in this menu
8	in the first menu item only, if this bit is set then the title data at R1+0 is the data as for an indirected text icon (see the section entitled *Wimp_CreateIcon (SWI &400C2)* on page 3-93)
all others	not used; must be zero

bytes 4 - 7	submenu pointer (>= &8000) or window handle (1 - &7FFF) (–1 if none)
bytes 8 - 11	menu icon flags – as for a normal icon
bytes 12 - 23	menu icon data (12 bytes) – as for a normal icon

If R1 is a window handle, the Wimp will open that window as the menu otherwise the menu structure must remain intact as long as the tree is open. The Wimp does not take a copy, but uses it directly.

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

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

The above only applies from RISC OS 3 onwards.

Pressing Return while the caret is inside a writable item is equivalent to pressing a mouse button, i.e. it selects that item.

A menu is basically a window whose work area is entirely covered by the menu items. The work area colour bytes at R1+14 and R1+15 are therefore not generally used unless the ‘gap between items’ is non-zero; they are overridden by the items’ icons colours. The window has a Title Bar if the string at R1+0 is non-null, otherwise it is untitled. If the title string is not indirected, its maximum length is the smaller of 12 and (item-width DIV 16); it should be terminated by a control code if its length is less than 12.

The menu will be automatically given a vertical scroll bar if it is taller than the current screen mode.

A menu item is a text icon whose bounding box is derived from width and height given at R1+16 and R1+20. Thus all entries in a menu are the same size. They are arranged vertically and lie horizontally between a ‘tick’ icon on the left and an arrow (submenu indicator) icon on the right, if present.

The menu item flags can alter the appearance of each item, e.g. by telling the Wimp to display the tick, or a separating dashed line beneath it. To shade an item, set bit 22 of the icon flags.

If the submenu pointer for an item is not –1, then it points to a similar data structure describing a submenu. An arrow is displayed to the right of the menu item; if the user moves the mouse pointer over this, then the submenu automatically pops up. Generally, submenu titles are the same as the parent item’s text, or can be a prompt like ‘Name:’.

The submenu pointer can be a window handle instead. Such a window is known as a dialogue box or dbox for short. In this case, the window is opened (as if it were a menu) when the mouse pointer moves over the arrow. The first writable icon in the window is given the input focus. You cannot close a menu window by clicking in it or pressing Return. Instead you should give it an ‘OK’ icon and treat clicks over that as a selection. The menu can then be closed using Wimp_CreateMenu with
R1 = –1.

If you want Return to make a selection, use the key-pressed event.

Cancelling a menu-window can be achieved by clicking outside of the menu structure, or by providing a ‘Cancel’ icon for the user to click on. In the first case, no Close_Window_Request is returned for the window; it is closed automatically by the Wimp.

When a menu window is closed, the caret is automatically given back to wherever it was before the window was opened.

Bit 3 of the menu flags changes the submenu behaviour. If it is set, then moving over the right arrow will cause a MenuWarning message to be generated. The application can respond as it sees fit, usually by calling Wimp_CreateSubMenu (page 3-196) to display the appropriate object. Note that in this case the submenu pointer in the menu structure does not have to be valid, but it is passed to the application in the message block anyway. The submenu pointer is important if Wimp_DecodeMenu will be used later on.

Many of the iconic properties of menu items can be controlled, using the icon flags word and icon data bytes. Below is a list of the aspects of an icon that a menu item may or may not exhibit:

- it can contain text. Indeed it must in order to be useful (bit 0 must be set)
- it can contain a sprite, but see note below
- it can have a border, but this isn't particularly useful
- the text is always centred vertically (bit 4 ignored), but the horizontal formatting bits (3 and 9) are used
- the background should be filled (bit 5 set)
- the text can be anti-aliased
- the item is drawn only by the Wimp (bit 7 ignored)
- the icon can be indirected – useful for long writable item strings
- the button type is always 9 and the ESG is always 0 (bits 12 - 20 ignored); use the menu flags to make an item writable
- the selected bit (21) isn't readable as the icon is 'anonymous'. The task hears about the final selection through the Menu_Selection event
- the shaded bit (22) is useful for disabling certain items. However, such items' submenu arrows can't be followed, so you should only shade leaf items
- the deleted bit (23) is irrelevant
- the colours/font handle byte (bits 24 - 31) should be set as appropriate.

The icon data contains either the actual text (0 to 12 characters, control-code terminated if less than twelve) or the three indirected icon information words. A validation string can naturally be used for writable items.

A menu item can only usefully contain a sprite if it is a sprite-only (no text) indirected icon. This allows for a sprite control block pointer to be given in the middle word of the icon data. Typically this is +1 for a Wimp sprite, or a valid user-area pointer.

If the task can create more than one menu, it must remember which menu is displayed, as the Wimp does not return this when a selection has been made. It must also scan down its data structure to determine which submenus the numbers relate to, before it can decide what action to take. Wimp_DecodeMenu (page 3-158) can help with this.

It is recommended that tasks use a ‘shorthand’ for defining menus, which is translated into the full form required by the Wimp when needed. But menus must be held in semi-permanent data structures once created, since the Wimp accesses them while menus are open.

Note that if a menu selection is made using Adjust, it is conventional for the application to keep the menu structure open afterwards. What happens is that the Wimp marks the menu tree temporarily when a selection is made. The application should call Wimp_GetPointerInfo to see if Adjust is pressed. If so, it should call Wimp_CreateMenu before returning to Wimp_Poll, which causes the tree to be re-opened in the same place.

The menu structure may be modified before re-opening, in which case any changes are noted by the Wimp, for example if menu entries become shaded. If the application does not call Wimp_CreateMenu, then the Wimp will delete the menu tree on the next call to Wimp_Poll, as the tree was marked temporary when the selection was made.

See the section entitled *Menus* on page 3-34 for more information about menus.

Related SWIs

None

Related vectors

None

Wimp_DecodeMenu (SWI &400D5)

Converts a numerical list of menu selections to a string containing their text

On entry

R1 = pointer to menu data structure
R2 = pointer to a list of menu selections
R3 = pointer to a buffer to contain the answer

On exit

R0 corrupted
buffer updated to contain menu item text, separated by '.'s

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call converts a numerical list of menu selections to a string containing the text of each successive menu item, e.g. `Display.Small icons` for a typical Filer menu selection.

Related SWIs

None

Related vectors

None

Wimp_WhichIcon (SWI &400D6)

Searches for icons that match a given flag word

On entry

R0 = window handle (or –2 to indicate the icon bar)
R1 = pointer to block to contain the list of icon handles
R2 = bit mask (bit set means consider this bit)
R3 = bit settings to match

On exit

R0 corrupted
block at R1 updated to contain a list of icon handle words, terminated by –1

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call compares the flag words of all of the icons belonging to the given window with the pattern given in R3. Each icon whose flags match has its handle added to the block pointed to by R1.

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

The mask in R2 is used to determine which bits are to be used in the comparison. The icon's handle is added to the list if (icon-flags `AND` bit-mask) = (bit-settings `AND` bit-mask). For example:

```
SYS "Wimp_WhichIcon",window,buffer,1<<21,1<<21
```

On exit a list of icon handles whose selected bit (21) is set will be in the buffer.

Similarly, to see which is the first icon with ESG number 1 that is selected:

```
SYS "Wimp_WhichIcon",window,buffer,&003F0000,&00210000
```

!buffer now contains the handle of the required icon, or –1 if none is selected.

Related SWIs

Wimp_GetIconState (page 3-138)

Related vectors

None

Wimp_SetExtent (SWI &400D7)

Sets the work area extent of a specified window

On entry

R0 = window handle
R1 = pointer to block

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the work area extent of the specified window, and usually causes the window's scroll bars to be redrawn (to reflect the new total size of window). The work area extent may not be changed so that any part of the visible work area lies outside the extent, so this call cannot change the current size of a window, or cause it to scroll.

On entry, the block contains:

R1+ 0	new work area minimum x
R1+ 4	new work area minimum y
R1+ 8	new work area maximum x
R1+ 12	new work area maximum y

It is usual to make this call when a document has been extended, e.g. by text being inserted into a word-processor.

Under RISC OS 2 you must set the extent to be a whole number of pixels. If not, strange effects can occur, such as the pointer moving beyond its correct bounding box. If you do this, the Wimp automatically readjusts the extent on a mode change.

From RISC OS 3 onwards the Window extent is automatically rounded to be a whole number of pixels (and is re-rounded on a mode change).

Related SWIs

None

Related vectors

None

Wimp_SetPointerShape (SWI &400D8)

Sets the shape and active point of the pointer

On entry

R0 = shape number (0 for pointer off)
R1 = pointer to shape data (–1 for no change)
R2 = width in pixels (must be multiple of 4)
R3 = height in pixels
R4 = active point x offset from top-left in pixels
R5 = active point y offset from top-left in pixels

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the shape and active point of the pointer.

The shape data is a series of bytes giving the pixel colours for the shape. Each row of the shape is given as a whole number of bytes (e.g. 3 bytes for a 12-pixel wide shape). Bytes are given in left to right order. The least significant two bits of each byte give the colour of the leftmost pixel in that group of four (i.e. it looks backwards as you write it down in binary).

In new programs, you should now use the call Wimp_SpriteOp (page 3-198) with R0=36 (SetPointerShape) instead of this one. The following principles still apply though.

This convention should be used when programming the pointer shape under the Wimp:

- shape 1 is the default arrow shape (set-up by *Pointer)
- to use an alternative, define and use shape 2
- when the pointer leaves the window where it was changed, it should be reset to shape 1.

The event codes Pointer_Entering_Window and Pointer_Leaving_Window returned from Wimp_Poll are very useful for deciding when to reprogram the pointer shape.

If you want to use Wimp_SpriteOp for all pointer shape programming, and wish to avoid using *Pointer, you can use the Wimp sprite `ptr_default` to program the standard arrow shape. Note however that `ptr_default` does not have a palette, so you would have to reset the pointer palette too if your pointer shape changed it.

Related SWIs

None

Related vectors

None

Wimp_OpenTemplate (SWI &400D9)

Opens a specified template file

On entry

R1 = pointer to template pathname to open

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This causes the Wimp to open the specified template file, and to read in some header information from the file. Only one template file may be open at a time; this is the one used by Wimp_LoadTemplate (page 3-167) when that SWI is called.

Related SWIs

Wimp_CloseTemplate (page 3-166), Wimp_LoadTemplate (page 3-167)

Related vectors

None

Wimp_CloseTemplate (SWI &400DA)

Closes the currently open template file

On entry

—

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This closes the currently open template file.

Related SWIs

Wimp_OpenTemplate (page 3-165), Wimp_LoadTemplate (page 3-167)

Related vectors

None

Wimp_LoadTemplate (SWI &400DB)

Loads a template

On entry

R1 = pointer to user buffer for template, or ≤ 0 to find the size of the template
R2 = pointer to workspace for indirected icons
R3 = pointer to byte following workspace
R4 = pointer to 256 byte font reference array (–1 for no fonts)
R5 = pointer to (wildcarded) name to match (must be 12 bytes word-aligned)
R6 = position to search from (0 for first call)

On exit

R0 corrupted
R1 preserved, or required size of buffer (if R1 ≤ 0 on entry)
R2 = pointer to remaining workspace, or required size of workspace (if R1 ≤ 0 on entry)
R3, R4 preserved
R5 = pointer to actual name
R6 = position of next entry (0 if no match found)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call loads a template. You must have previously called Wimp_OpenTemplate to open the template.

The space required by the buffer passed in R1 is 88 bytes for the window, 32 bytes for each icon and room for the initial values of all indirected data fields, since all of these things are initially copied into the buffer. The indirected data is then copied by the Wimp into the workspace area pointed to by R2. The font reference array is also updated if fonts are used.

The required sizes of the buffer and workspace area are hard to work out, and so in RISC OS 3 the option was added whereby you can find these values by setting R1 ≤ 0. You should use this option where possible.

Window templates are created by the template creation utility (FormEd). They are stored in a file, and each template has a name associated with it. Because the search name may be wildcarded, it is possible to search for all templates of a given form (e.g. dialog*) by calling Wimp_LoadTemplate with R6=0 the first time, then using the value passed back for subsequent calls. R6 will be returned as 0 on the call after the last template is found. As the wildcarded name is overwritten by the actual one found, it must be re-initialised before every call and must be big enough to have the template name written into it.

The indirected icon workspace pointer is provided so that when the window definition is read into the buffer addressed by R1, its icon fields can be set correctly. An indirected icon's data is read from the file into the workspace addressed by R2, and the icon data pointer fields in the window definition are set appropriately. R2 is updated, and if it becomes greater than R3, a `Window definition won't fit` error is given.

The font reference count array is used to overcome the problem caused with dynamically allocated font handles. When a template file is created, font information such as size, font name etc is stored along with the font handle that was returned for the font in FormEd. When a template is subsequently loaded, the Wimp calls Font_FindFont and replaces references to the original font number with the new handle. It then increments the entry for that handle in the reference array. This array should be initialised to zero before the first call to Wimp_LoadTemplate.

When a window is deleted, for all font handles in the range 1 - 255 you should call Font_LoseFont the number of times given by that font's reference count. This implies that a separate 256 byte array is needed for each template loaded. However, this can be stored a lot more compactly (e.g. using font handle/count byte pairs) once the array has been set up by Wimp_LoadTemplate.

An alternative is to have a single reference count array for all the windows in the task, and only call Font_LoseFont the appropriate number of times for each handle when the task terminates.

Errors

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

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

No error is generated for objects of type ≠ 1: the object is simply loaded into the buffer, and no indirected data processing occurs. This is different from RISC OS 2, which reported an error in these circumstances.

Related SWIs

Wimp_OpenTemplate (page 3-165), Wimp_CloseTemplate (page 3-166)

Related vectors

None

Wimp_ProcessKey (SWI &400DC)

Creates or passes on key presses

On entry

R0 = character code

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call has two uses. The first is to make the Wimp return a Key_Pressed event as though the character code passed in R0 was typed by the user. It is useful in programs where a menu of characters corresponding to those not immediately available from the keyboard is presented to the user, and clicking on one of them causes the code to be entered as if typed.

The second use is to pass on a keypress that a task does not understand, so that other applications (with the ‘hot key’ window flag set) may act on it. The key is passed (via the Key_Pressed event) to each eligible task in turn, from the top of the window stack down. It stops when a task fails to call Wimp_ProcessKey (because it recognises the key), or until the bottom window is reached.

For this to work, it is vital that a task always passes on unrecognised key presses using Wimp_ProcessKey. Conversely, if the program can act on the key stroke, it should not then call Wimp_ProcessKey, as this might result in a single key stroke causing several separate actions.

As a last resort, if no task acts on a function key press, the Wimp will expand the code into the appropriate function key string and insert it into the writable icon that owns the caret, if any.

Related SWIs

None

Related vectors

None

Wimp_CloseDown (SWI &400DD)

Informs the Wimp that a task is about to terminate

On entry

R0 = task handle returned by Wimp_Initialise (only required if R1='TASK')
R1 = 'TASK' (see Wimp_Initialise &400C0)

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call must be made immediately before the task terminates by calling OS_Exit. If this was the only extant task, the Wimp will reset the soft key and mode settings to their original values (i.e. as they were before Wimp_Initialise was first called). Any application memory used by the task will be returned to the Wimp's free pool.

If the task handle is not given, then the Wimp will close down the currently active task, i.e. the one which was the last to have control returned to it from Wimp_Poll. This is sufficient if the task is loaded in the application workspace (as opposed to being a relocatable module).

Module tasks should always pass their handle to Wimp_CloseDown, as there is no guarantee that the module in question is the active one at the time of the call. For example, a task module would be required to close down in its 'die' code, which may be called asynchronously without control passing to the module through Wimp_Poll.

A Wimp_CloseDown will cause the service call WimpCloseDown (&53) to be generated. See the section entitled *Relocatable module tasks* on page 3-60 for details.

Related SWIs

Wimp_Initialise (page 3-85)

Related vectors

None

Wimp_StartTask (SWI &400DE)

Starts a ‘child’ task from within another program

On entry

R0 = pointer to * Command to be executed

On exit

R0 = handle of task started, if it is still alive; 0 otherwise
(not available in RISC OS 2)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is used to start a ‘child’ task from within another program. The text pointed to by R0 on entry can be any * Command which will cause a Wimp program to be executed, e.g. `BASIC -quit myProg`.

The Wimp will create a new ‘domain’ or environment for the task and calls OS_CLI to execute the command. If the new task subsequently calls Wimp_Initialise and then Wimp_Poll, control will return to caller of Wimp_StartTask. Alternatively, control will return when the new task terminates through OS_Exit (which `QUIT` in BASIC calls).

This call is used by the Desktop and the Filer to start new tasks.

Note that you can only call this SWI:

- if you are already a ‘live’ Wimp task, and have gained control from Wimp_Initialise or Wimp_Poll.
- you are in USR mode.

Related SWIs

None

Related vectors

None

Wimp_ReportError (SWI &400DF)

Reports errors

On entry

R0 = pointer to standard error block, see below
R1 = flags, see below
R2 = pointer to application name for error window title (< 20 characters)

On exit

R0 corrupted
R1 = 0 if no key click, 1 if `OK` selected, 2 if `Cancel` selected

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call provides a built-in means for reporting errors that may occur during the running of a program. The error number and its text is pointed to by R0. The control code-terminated string pointed to by R2 is used in the Title Bar of the error window, optionally preceded by the text `Error from` .

The format of a standard error block is:

R0+0	error number
R0+4	zero-terminated error string

The flags in R1 on entry have the following meanings:

Bit	Meaning when set
0	provide an `OK` box
1	provide a `Cancel` box
2	highlight `Cancel` (or `OK` if bit is cleared)
3	if the error is generated while a text-style window is open (e.g. within a call to Wimp_CommandWindow), then don't produce the prompt `Press SPACE or click mouse to continue`, but return immediately
4	don't prefix the application name with `Error from` in the error window's Title Bar
5	if neither box is clicked, return immediately with R1=0 and leave the error window open
6	select one of the boxes according to bits 0 and 1, close the window and return
7	will not produce a 'beep' even if WimpFlags bit 4 is clear (this bit is reserved in RISC OS 2)
8 - 31	reserved; must be 0

If neither bit 0 or 1 is set, an `OK` box is provided anyway. Bits 5 and 6 can be used to regain control while the error window is still open, say to implement timeouts (an example is the disc insert box, which polls the disc drive to see if a disc has been inserted), or use keypresses to stand for clicks on either of the boxes. Note though that the Wimp should not be re-entered while an error window is open, so you should always call Wimp_ReportError with bit 6 of R1 set before you next call Wimp_Poll, if you are using bit 5 in this way.

Wimp_ReportError causes the Service WimpReportError (&57) to be generated. See the section entitled *Relocatable module tasks* on page 3-60 for details.

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

Note that RISC OS 2 will always return R1=1 (i.e. `OK` clicked), even if the `Cancel` box is highlighted.

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

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

Related SWIs

None

Related vectors

None

Wimp_GetWindowOutline (SWI &400E0)

Gets the bounding box for a window

On entry

R1 = pointer to a five-word block

On exit

R0 corrupted
The block is updated

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call gets the bounding box for a window.

On entry, R1+0 contains the window handle; on exit the block is updated thus:

R1+0	window handle (or –2 to indicate the icon bar)
R1+4	minimum x coordinate of window bounding box
R1+8	minimum y coordinate of window bounding box
R1+12	maximum x coordinate of window bounding box
R1+16	maximum y coordinate of window bounding box

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

The Wimp supplies the x0,y0 inclusive, x1, y1 exclusive coordinates of a rectangle which completely covers the specified window, including its border. This call is useful when you want, for example, to set a mouse rectangle to the same size as a window.

Note that this call will only work after a window is opened, not just created.

Related SWIs

None

Related vectors

None

Wimp_PollIdle (SWI &400E1)

Polls the Wimp, sleeping unless certain events have occurred

On entry

R0 = mask (see Wimp_Poll)
R1 = pointer to 256 byte block (used for return data; see Wimp_Poll)
R2 = earliest time for return with Null_Reason_Code event
R3 = pointer to poll word if R0 bit 22 is set (not in RISC OS 2)

On exit

see Wimp_Poll (page 3-112)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call performs the same task as Wimp_Poll. However, the caller also specifies an OS_ReadMonotonicTime-type time on entry. The call will not return before then, unless there is a non-null event to be processed. Effectively the caller can ‘sleep’, not being woken up until the specified time has passed or until it has some action to perform. This gives more processing time to other tasks.

Having performed the appropriate action upon return, the task should add its ‘time-increment’; (e.g. 100 for a one-second granularity clock) to the previous value it passed in R2 and call Wimp_PollIdle again.

Note that if the Wimp is suspended for a while (eg the user goes into the command prompt) and then returns, it is possible for the current time to be much later than the ‘earliest return’ time.

For this reason, it is recommended that (for example) a clock task should cater for this by incorporating the following structure:

```
SYS"OS_ReadMonotonicTime" TO newtime
WHILE (newtime - oldtime) > 0
       oldtime=oldtime+100
ENDWHILE
REM Then pass oldtime to Wimp_PollIdle
```

Related SWIs

Wimp_Poll (page 3-112)

Related vectors

None

Wimp_PlotIcon (SWI &400E2)

Plots an icon in a window during a window redraw or update loop

On entry

R1 = pointer to an icon block (see below)

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call can be used to plot an icon in a window during a window redraw or update loop. The icon doesn't exist as part of the window's definition. Instead, the data to be used to plot the icon is passed explicitly through R1. The format of the block is the same as that used by Wimp_CreateIcon (page 3-93), except that there is no window handle associated with it (this being implicitly the window which is currently being redrawn or updated):

R1+0	minimum x coordinate of icon bounding box
R1+4	minimum y coordinate of icon bounding box
R1+8	maximum x coordinate of icon bounding box
R1+12	maximum y coordinate of icon bounding box
R1+16	icon flags
R1+20	icon data

See Wimp_CreateIcon on page 3-93 for details about these fields.

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

Related SWIs

None

Related vectors

None

Wimp_SetMode (SWI &400E3)

Changes the display mode used by the Wimp

On entry

R0 = mode number

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call changes the display mode used by the Wimp. It should not be used by applications (which should be able to work in any mode), unless absolutely necessary. Its main client is the palette utility, which allows the user to change mode as required.

In addition to changing the mode this call resets the palette according to the number of colours in the new mode, reprograms the mouse pointer appropriately and re-allocates the screen memory to use the minimum required for this mode. In addition, the screen is rebuilt (by asking all tasks to redraw their windows) and tasks are informed of the change through a Wimp_Poll message.

Notes: the new mode is remembered for the next time the Wimp is started, but does not affect the configured Wimp mode, so this will be used after a hard reset or power-up. If there is no active task when Wimp_SetMode is called, the mode change doesn't take place until Wimp_Initialise is next called. If there is insufficient memory for the mode change, it is remembered and no error is generated.

On the next call to Wimp_Poll after a mode change, the Wimp issues Message_ModeChanged and Open_Window_Requests for all open windows. If the new mode is smaller than the previous one, the windows are also forced back onto the screen. This does not happen in RISC OS 2.

Related SWIs

Wimp_SetPalette (page 3-187)

Related vectors

None

Wimp_SetPalette (SWI &400E4)

Sets the palette

On entry

R1 = pointer to 20-word palette block

On exit

R0 corrupted
R1 preserved

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the palette.

The block pointed to by R1 contains the following on entry:

R1+0	Wimp colour 0 RGB value
R1+4	Wimp colour 1 RGB value
R1+8	Wimp colour 2 RGB value
…	…
R1+56	Wimp colour 14 RGB value
R1+60	Wimp colour 15 RGB value
R1+64	border colour RGB value
R1+68	pointer colour 1 RGB value
R1+72	pointer colour 2 RGB value
R1+76	pointer colour 3 RGB value

Each RGB value word has the format &BBGGRR00, i.e. bits 0 - 7 are reserved, and should be 0, bits 8 - 15 are the red value, bits 16 - 23 the green and bits 24 - 31 the blue, as used in a VDU 19,l,16,r,g,b command. The call, whose main user is the palette utility, issues the appropriate palette VDU calls to reflect the new values given in the 20-word block. In modes other than 16-colour ones, a remapping of the Wimp's colour translation table may be required, necessitating a screen redraw. It is up to the user of Wimp_SetPalette to cause this to happen (the palette utility does). Tasks are informed of palette changes through a message event returned by Wimp_Poll.

Related SWIs

Wimp_SetMode (page 3-185), Wimp_ReadPalette (page 3-189)

Related vectors

None

Wimp_ReadPalette (SWI &400E5)

Reads the palette

On entry

R1 = pointer to 20-word palette block

On exit

R0 corrupted
R1 preserved

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call reads the palette. The 20-word block is updated in the format described under Wimp_SetPalette (page 3-187). However, the bottom byte of the first 16 entries contains the logical colour number that is used for that Wimp colour. This is the same as the Wimp colour in 16-colour modes. In 256 colour modes, bits 0 and 1 are bits 6 and 7 of the tint, and bits 2 - 7 are the GCOL colour.

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

Applications can use this call to discover all of the current Wimp palette settings.

Related SWIs

Wimp_SetPalette (page 3-187)

Related vectors

None

Wimp_SetColour (SWI &400E6)

Sets the current graphics foreground or background colour and action

On entry

R0 = colour and GCOL action (see below)

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This calls is used to set the current graphics foreground or background colour and action to one of the 16 standard Wimp colours. As described earlier, these map into ECF patterns in monochrome modes, four grey-level colours in four-colour modes, the available colours in 16-colour modes, and the closest approximation to the Wimp colours in 256-colour modes.

The format of R0 is as follows:

Bits	Meaning
0 - 3	Wimp colour
4 - 6	GCOL action
7	0 for foreground, 1 for background

After the call to Wimp_SetColour, the appropriate GCOL, TINT and (in two-colour modes) ECF commands will have been issued. The Wimp uses ECF pattern 4 for its purposes.

Related SWIs

None

Related vectors

None

Wimp_SendMessage (SWI &400E7)

Sends a message to a task, or broadcasts to all tasks

On entry

R0 = event code (as returned by Wimp_Poll – often 17, 18 or 19)
R1 = pointer to message block
R2 = task handle of destination task, or
window handle (message sent to window's creator), or
–2 (icon bar: message sent to creator of icon given by R3), or
0 (broadcast message, sent to all tasks, including the originator)
R3 = icon handle (only used if R2 = –2)

On exit

R0 corrupted
R2 = task handle of destination task (except for broadcast messages)
the message is queued
the message block is updated (event codes 17 and 18 only)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

All messages within the Wimp environment are generated using this call. The Wimp uses it internally to keep tasks informed about various events through their Wimp_Poll loop.

For a full description of all the message action codes see the section entitled *Messages* on page 3-228.

User tasks can also generate these types of message, with event codes in the range 0 to 12. On entry, R1 should point to a block with the format described under Wimp_Poll (page 3-112). For example, if you send an Open_Window_Request to a task (R0=2), you should point R1 at a Wimp_OpenWindow (page 3-109) block.

More often though, Wimp_SendMessage is used by tasks to send events of type User_Message to one another. These differ from the 'system' types, in that the Wimp performs some special actions, e.g. filling in fields of the message block, and noting whether a reply has been received.

There are three variations, depending on the event code in R0 on entry. The first two, User_Message and User_Message_Recorded (17 and 18), send a message to the destination task(s). The latter expects the message to be acknowledged or replied to, and if it isn't the Wimp returns the message to the sender. (See Wimp_Poll event codes 17, 18 and 19.)

Event code User_Message_Acknowledge (19) is used to acknowledge the receipt of a message without actually generating an event at the destination task. The receiver copies the my_ref field of the message block into the your_ref field and returns the message using the task handle of the sender given in the message block. If you acknowledge a broadcast message, it is not passed on to any other tasks.

The format of a user message block is:

R1+0	length of block, 20 - 256 bytes, a whole number of words
R1+4	not used on entry
R1+8	not used on entry
R1+12	your_ref (0 if this is an original message, not a reply)
R1+16	message action
R1+20	message data (format depends on the message action)
…	

Note that the block length should include any string that appears on the end (e.g. pathnames), including the terminating character, and rounded up to a whole number of words.

On exit the block is updated as follows:

R1+4	task handle of sender
R1+8	my_ref (unique Wimp-generated non-zero positive word)

Thus the receiver of the message will know who sent the message (useful for acknowledgements) and will also have a reference that can be quoted in replies to the sender. Naturally the sender can also use these fields once the Wimp has filled them in.

Note that you can use User_Message_Acknowledge to discover the task handle of a given window/icon by calling Wimp_SendMessage with R0=19, your_ref = 0, and R2/R3 the window/icon handle(s). On exit R2 will contain the task handle of the owner, though no message would actually have been sent.

Related SWIs

Wimp_Poll (page 3-112)

Related vectors

None

Wimp_CreateSubMenu (SWI &400E8)

Creates a submenu

On entry

R1 = pointer to submenu block
R2 = x coordinate of top left of submenu
R3 = y coordinate of top left of submenu

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is made when a message type MenuWarning (&400C0) is received by an application. This message is sent by the Wimp when a submenu is about to be accessed by the pointer moving over the right-pointing arrow of the parent menu.

The contents of R1 - R3 are obtained from the three words at offsets +20 to +28 of the message block. However, the submenu pointer does not have to be the same as that given in this block (which is just a copy of the one given in the parent menu entry when it was created by Wimp_CreateMenu). For example, the application could create a new window, and use its handle instead.

Related SWIs

Wimp_CreateMenu (page 3-153)

Related vectors

None

Wimp_SpriteOp (SWI &400E9)

Performs sprite operations on sprites from the Wimp's pool

On entry

R0 = reason code (in the range 0 - &FF, see OS_SpriteOp (page 1-788))
R1 not used
R2 = pointer to sprite name
R3… OS_SpriteOp parameters

On exit

R0 corrupted
R2… OS_SpriteOp results

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call allows operations on Wimp sprites, without having to specify the Wimp's sprite area pointer. Sprites are always accessed by name (i.e. &100 is added to the reason code given); pointers to actual sprites are not used. Only read-type operations are allowed, except that you may use the reason code MergeSpriteFile (11) to add further sprites to the Wimp area.

The Wimp first tries to access the sprite in the RMA part of its sprite pool. If it is not found there, it tries the ROM sprite area. If this fails, it returns the usual `Sprite not found` message.

Related SWIs

OS_SpriteOp (page 1-788)

Related vectors

None

Wimp_BaseOfSprites (SWI &400EA)

Finds the addresses of the ROM and RAM resident parts of the Wimp's sprite pool

On entry

—

On exit

R0 = base of ROM sprite area
R1 = base of RMA sprite area

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This can be used to find out the actual addresses of the two areas that make up the Wimp sprite pool, for use with OS_SpriteOp. Note that the RMA area may move around, e.g. after a sprite file has been merged with it. In view of this, you should use Wimp_SpriteOp if possible.

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

Related SWIs

None

Related vectors

None

Wimp_BlockCopy (SWI &400EB)

Copies a block of work area space to another position

On entry

R0 = window handle
R1 = source rectangle minimum x coordinate (inclusive)
R2 = source rectangle minimum y coordinate (inclusive)
R3 = source rectangle maximum x coordinate (exclusive)
R4 = source rectangle maximum y coordinate (exclusive)
R5 = destination rectangle minimum x coordinate
R6 = destination rectangle minimum y coordinate

On exit

R0 - R6 = preserved

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call copies a block of work area space to another position. The Wimp does as much on-screen work as it can, using the VDU block copy primitive, and then invalidates any areas which must be updated by the application itself. The call is useful for performing insert/delete operations in editors.

All coordinates are relative to the window's work area origin. Note that if any of the source area contains icons, their on-screen images will be copied, but their bounding boxes will not automatically be moved to the destination rectangle. It is up to the application to move the icons explicitly (by deleting and re-creating then) so that they are redrawn correctly.

If the source area contains an ECF pattern, e.g. representing Wimp colours in a two-colour mode, and the distance between the source and destination is not a multiple of the ECF size (eight pixels vertically and one byte horizontally), then the copied area will be 'out of sync' with the existing pattern.

Note that this call must not be made from inside a Wimp_RedrawWindow or Wimp_UpdateWindow loop.

Related SWIs

None

Related vectors

None

Wimp_SlotSize (SWI &400EC)

Reads or sets the size of the current slot, the next slot, and the Wimp free pool

On entry

R0 = new size of current slot (–1 to read size)
R1 = new size of next slot (–1 to read size)

On exit

R0 = size of current slot (i.e. memory for current task)
R1 = size of next slot (i.e. desirable allocation for next task)
R2 = size of free pool (i.e. free memory)
R4 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

Tasks can use this call to read or set the size of the current slot, i.e. that in which the task is executing, and the next slot (for the next task to start up). It also returns the (possibly altered) size of the Wimp free pool.

If a task wants to alter its memory, it should set R0 to the required amount and R1 to –1.

Next is a number and can be larger than free, in which case next task just gets free. Note that the next slot size does not actually have any effect until the next new task is run. It is simply the amount of the free pool that is allocated to a new task by default.

No tasks should set their current slot size; normally, a new task will call *WimpSlot, which then calls Wimp_SlotSize.

On exit from Wimp_SlotSize, the OS_ChangeEnvironment variables MemoryLimit and ApplicationSpaceSize are updated. Note that it is not possible to change the application space size if this is greater than MemoryLimit. This is the situation when, for example, Twin loads at &80000 and runs another task at &8000, setting that task's memory limit to &80000.

Wimp_SlotSize does not check that the currently active object is within the application workspace, or issue Memory service calls, so it should be used with caution. The same applies to *WimpSlot which uses this SWI.

Possible ways in which this call could be used are:

- the run-time library of a language could provide a system call to set the current slot size using Wimp_SlotSize. An example is BASIC's END=&xxxx construct, which allows a program to adjust its HIMEM limit dynamically.
- a program could use Wimp_SlotSize to give itself a private heap above the area used by the host language's memory allocation routines. This only works if the run-time library routines read the MemoryLimit value once, when the program is started. Edit uses this method to allocate memory for its text files.

Related SWIs

None

Related vectors

None

Wimp_ReadPixTrans (SWI &400ED)

Read pixel translation table for a given sprite

On entry

R0 = &0xx if sprite is in the system area
&1xx if sprite is in a user area and R2 points to the name
&2xx if sprite is in a user area and R2 points to the sprite
R1 = 0 if the sprite is in the system area
1 if the sprite is in the Wimp's sprite area
otherwise a pointer to the user sprite area
R2 = a pointer to the sprite name (R0 = &0xx or &1xx) or
a pointer to the sprite (R0 = &2xx)
R6 = a pointer to a four-word block to receive scale factors, 0 ⇒ do not fill in
R7 = a pointer to a 2, 4 or 16 byte block to receive translation table,
0 ⇒ do not fill in (must be 16 bytes long)

On exit

R0 corrupted
R6 block contains the sprite scale factors
R7 block contains a 2, 4, or 16 byte sprite translation table

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The purpose of this call is to discover, for a given sprite, how the Wimp would plot it if it was in an icon to give it the most consistent appearance independently of the current Wimp mode. The blocks set up at R6 and R7 on exit can be passed directly to the above mentioned sprite plotting calling.

If the sprite is not found in the passed area, it is then searched for in the Wimp sprite pool – except under RISC OS 2.

The size of the table pointed to by R7 depends on the sprite's mode. Under RISC OS 2 the sprite cannot have 256 colours.

The format of the R6 block is:

R6+0	x multiplication factor
R6+4	y multiplication factor
R6+8	x division factor
R6+12	y division factor

All quantities are 32-bits and unsigned.

The format of the R7 block is:

R7+0	colour to store sprite colour 0 as
R7+1	colour to store sprite colour 1 as
…	
R7+14	colour to store sprite colour 14 as
R7+15	colour to store sprite colour 15 as

Scale factors depend on the mode the sprite was defined in and the current Wimp mode. The colour translation table is only valid for sprites defined in 1, 2 or 4-bits per pixel modes. The relationships between the sprite colours and the Wimp colours used to display them are:

Sprite bpp	**Colours used**
1	Colours 0 - 1 –> Wimp colours 0, 7
2	Colours 0 - 3 –> Wimp colours 0, 2, 4, 7
4	Colours 0 - 15 –> Wimp colours 0 - 15
8	Translation table is undefined

So sprites defined with fewer than four bits per pixel have their pixels mapped into the Wimp's greyscale colours.

Use ColourTrans if you want to plot the sprite using the best approximation to its actual colours. This works for sprites in a 256-colour mode as well.

Related SWIs

None

Related vectors

None

Wimp_ClaimFreeMemory (SWI &400EE)

Claims the whole of the Wimp's free memory pool for the calling task

On entry

R0 = 1 to claim, 0 to release
R1 = amount of memory required

On exit

R0 corrupted
R1 = amount of memory available (0 if none/already claimed)
R2 = start address of memory (0 if claim failed because not enough)

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is analogous to OS_ClaimScreenMemory (page 1-388). It allows a task to claim the whole of the Wimp's free memory pool (the 'Free' entry on the Task Manager display) for its own use. There are restrictions however: the memory can only be accessed in processor supervisor (SVC) mode, and while it is claimed, the Wimp can't use the free pool to dynamically increase the size of the RMA etc. For the second reason, tasks should not hang on to the memory for any longer than absolutely necessary. They should also avoid calling code which is likely to have much to do with memory allocation, e.g. code which claims RMA space. In other words, do not call Wimp_Poll while the free pool is claimed.

Related SWIs

OS_ClaimScreenMemory (page 1-388)

Related vectors

None

Wimp_CommandWindow (SWI &400EF)

Opens a text window in which normal VDU 4-type output can be displayed

On entry

R0 = operation type, see below

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call opens a text window in which normal VDU 4-type output can be displayed. It is useful for running old-fashioned, text-based programs from within the Wimp environment. The exact action depends on R0 as follows.

R0 > 1 R0 is treated as a pointer to a text string. This is used as the title for the command window. However, the command window is not opened immediately; it is just marked as ‘pending’. It does not become ‘active’ until the next call to OS_WriteC. When this occurs, the window is opened and the VDU 4 text viewport is set to the same area on the screen.

R0 = 1 The command window status is set to ‘active’. However, no drawing on the screen occurs. This is used by the ShellCLI module so that if Wimp_ReportError is called, the error will be printed textually and not in a window.

R0 = 0 The window is closed and removed from the screen. If any output was generated between the window being opened with R0 > 1 and this call being made, the Wimp prompts with `Press SPACE or click mouse to continue` before re-building the screen.

R0 = –1 The command window is closed without any prompting, regardless of whether it was used or not.

The Wimp uses a command window when starting new tasks. It calls Wimp_CommandWindow with R0 pointing to the command string, and then executes the command. If the task was a Wimp one, it will call Wimp_Initialise, at which point the Wimp will close the command window with R0 = –1. Thus the window will never be activated. However, a text-based program will never call Wimp_Initialise, so the command window will be displayed when the program calls OS_WriteC for the first time.

Certain Filer operations which result in commands such as *Copy being executed also use the command window facility in this way.

Wimp_ReportError (page 3-176) also interacts with command windows. If the window is active, the error text will simply be displayed textually. However, if the command window is pending, it is marked as ‘suspended’ and the error is reported in a window as usual.

Related SWIs

None

Related vectors

None

Wimp_TextColour (SWI &400F0)

Sets the text foreground or background colour

On entry

R0 = colour

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is the text colour equivalent of Wimp_SetColour (page 3-191). It is used to set the text foreground or background colour to one of the 16 standard Wimp colours. As text can't be displayed using ECF patterns, only solid colours are used in the monochrome modes.

R0 on entry has the following form:

Bits	Meaning
0 - 3	Wimp colour (0 - 15)
7	0 for foreground, 1 for background

Wimp_TextColour is used by Wimp_CommandWindow (page 3-210) and on exit from the Wimp. It can be called by applications that wish to display VDU 4-type text on the screen in a special window.

Related SWIs

Wimp_SetColour (page 3-191)

Related vectors

None

Wimp_TransferBlock (SWI &400F1)

Copies a block of memory from one task's address space to another's

On entry

R0 = handle of source task
R1 = pointer to source buffer
R2 = handle of destination task
R3 = pointer to destination buffer
R4 = buffer length

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call copies a block of memory from the source task's address space to the destination task. The buffer addresses and the length are byte aligned, i.e. the buffers don't have to start on a word boundary or be a whole number of words long.

This call is used in the memory data transfer protocol, described in the section entitled *Data transfer protocol* on page 3-247. The Wimp ensures that the addresses given are valid for the task handles, and generates the error `Wimp transfer out of range` if they are not.

Related SWIs

None

Related vectors

None

Wimp_ReadSysInfo (SWI &400F2)

Reads system information from the Wimp

On entry

R0 = information item index

On exit

R0 = information value

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is used to obtain information from the Wimp which is not readily available otherwise. The value in R0 on entry indicates which item of information is required; its value on exit is the appropriate value.

Currently defined values for R0 are:

R0 on entry	On exit
0	R0 = number of active tasks
1	R0 = current Wimp mode
2	R0 = pointer to iconsprites filename suffix for the configured mode (When loading sprite files containing icons, the suffix should be tried; if the file does not exist, try the original filename.)

3 R0 = 0 ⇒ in text output mode (i.e. outside the desktop, or in the ShellCLI, or in a command window)
= 1 ⇒ in the desktop
other values reserved (test for non-zero when looking to see whether in command mode or not)
The Wimp also supports a code variable `Wimp$State`, which can take the following values:

`commands`	Wimp_ReadSysInfo (3) returns 0
`desktop`	Wimp_ReadSysInfo (3) returns 1

other values should be treated as 'not `commands`'.

4 R0 = 0 ⇒ left to right text entry
= 1 ⇒ right to left text entry this returns the state last set by *WimpWriteDir

5 R0 = current task handle (0 if none active)
R1 = version specified by current task to Wimp_Initialise

6 Reserved

7 R0 = current Wimp version * 100

RISC OS 2 does not support values of R0 > 0.

As the call can be used regardless of whether Wimp_Initialise has been called yet, it can be used to see if the program is running from within the desktop environment (R0 > 0 on exit) or simply from a command line (R0 = 0). Note that even if a program is activated from the Task Manager's command line (F12) facility, R0 will be greater than zero.

Related SWIs

None

Related vectors

None

Wimp_SetFontColours (SWI &400F3)

Sets the anti-aliased font colours from the two (standard Wimp) colours specified

On entry

R1 = font background colour
R2 = font foreground colour

On exit

R0 corrupted

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the anti-aliased font colours from the two (standard Wimp) colours specified. It calculates how many intermediate colours can be used, and makes the appropriate Font Manager calls. It takes the display mode into account, so that using this call instead of setting the font colours directly saves the application quite a lot of work.

You should not assume the font colours are as you left them across calls to Wimp_Poll, as another task may have called Wimp_SetFontColours before you regain control. Conversely, you don't have to preserve the colours before you change them, as no-one else will be expecting you to.

This call is less powerful than ColourTrans_SetFontColours (page 3-370), in that it assumes that Wimp colours 0-7 form a grey-scale sequence.

Related SWIs

Wimp_SetColour (page 3-191)

Related vectors

None

Wimp_GetMenuState (SWI &400F4)

Gets the state of a menu, showing which item is selected

On entry

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

On exit

R0 corrupted
The tree is put into the buffer in R1 in the same format as that returned by Wimp_Poll event code 9 (Menu_Select), i.e. a list of selection indices terminated by –1.

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The tree returned will be null:

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

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

This SWI is not available under RISC OS 2.

Related SWIs

None

Related vectors

None

Wimp_RegisterFilter (SWI &400F5)

Used by the Filter Manager to register or deregister a filter with the Wimp

On entry:

R0 = reason code:
- 0 ⇒ register / deregister pre-filter
- 1 ⇒ register / deregister post-filter
- 2 ⇒ register / deregister rectangle copy filter
- 3 ⇒ register / deregister get rectangle filter

R1 = address of filter, or 0 to de-register
R2 = value to be passed in R12 on entry to filter

On exit:

Registers preserved

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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

This SWI is not available under RISC OS 2.

Pre-filters

A pre filter is called whenever a task calls Wimp_Poll:

On Entry:

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

SVC mode, interrupts enabled. The task that called Wimp_Poll is paged in.

On Exit:

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

Post-filters

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

On Entry:

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

SVC mode, interrupts enabled. The task to which the event is to be returned is paged in.

On Exit:

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

R1,R2 must be preserved.

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

Related SWIs

None

Related vectors

None

Wimp_AddMessages (SWI &400F6)

Adds messages to the list of those known by a certain task

On entry

R0 = pointer to word array of messages to add for task

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI allows you to update the list of messages known by a certain task. This routine updates the messages list for the current task.

This call is of use only for tasks that specified a Wimp version number ≥ 300 to Wimp_Initialise.

Related SWIs

None

Related vectors

None

Wimp_RemoveMessages (SWI &400F7)

Removes messages from the list of those known by a certain task

On entry

R0 = pointer to word array of messages to remove from task

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI allows the caller to remove messages from the list specified either on Wimp_Initialise or by Wimp_AddMessages.

This call is of use only for tasks that specified a Wimp version number ≥ 300 to Wimp_Initialise.

Related SWIs

None

Related vectors

None

Wimp_SetColourMapping (SWI &400F8)

Changes the mapping between Wimp colours and physical colours

On entry

R1 = pointer to palette to be used for converting Wimp colours to physical colours
= –1 the default Wimp palette is used
= 0 the palette defined by Wimp_SetPalette is used
else the table is copied away
R2 = pointer to 2 byte array for mapping 1BPP sprites to Wimp colours
R3 = pointer to 4 byte array for mapping 2BPP sprites to Wimp colours
R4 = pointer to 16 byte array for mapping 4BPP sprites to Wimp colours
R5,R6,R7 must be 0

On exit

—

Interrupts

Interrupts are not defined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI is used to change the way in which the Window Manager maps its own Wimp colours to physical colours.

On entry R1 contains a pointer to a 16 word set of physical colours (&BBGGRRxx). When converting a Wimp colour to its physical colour it indirects through this to get the physical colour required. By default this is the same as the palette defined using Wimp_SetPalette.

R2, R3, and R4 point to byte arrays which are used when converting non-paletted sprite colours to their physical colours. Basically, the system uses the values stored within the byte array as an index into the palette being used for ColourTrans calls. Passing 0 indicates no change, –1 indicates the default setting.

Related SWIs

None

Related vectors

None

Messages

Changes applying to applications passing 300 to Wimp_Initialise

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

Message actions

The following is a description of the currently defined message actions. Some of these are system types, others are generated by particular modules (most notably the Wimp). Any other module or application can send its own private messages, as required. A module is allowed to use its SWI chunk number as a base for the message action values. If you require a message action chunk and do not have a SWI chunk allocated, refer to the section entitled *SWI chunk numbers and names* on page 4-552.

System messages

Message_Quit (0)

On receiving this broadcast message a task should tidy up (close files, de-allocate memory etc) and close down by calling Wimp_CloseDown (page 3-172) and OS_Exit. The task doesn't have any choice about closing down at this stage. Any objections (because of unsaved data etc) should be lodged when it gets the Message_PreQuit (8) described below.

Message_DataSave (1) – Message_RAMTransmit (7)

See the section entitled *Data transfer protocol* on page 3-247 for details of these message actions.

Message_PreQuit (8)

This broadcast message gives applications the chance to object to a request to close down; for example, if they have modified data which has not been saved. If the task does not mind terminating, it should ignore this message, and eventually a Message_Quit will be received.

To object to the potential closedown, the task should acknowledge the message by calling Wimp_SendMessage with:

R0 = User_Message_Acknowledge (19)
R1 = as returned by Wimp_Poll
R1+12 = R1+8 (i.e. my_ref copied into your_ref)

Note that if the user subsequently selects OK (i.e. discard the data and quit anyway), the task must restart the closedown sequence by issuing a key-pressed event (Ctrl-Shift-F12) to the task which sent it the PreQuit message:

```
SYS "Wimp_GetCaretPosition",,blk
blk!24=&1FC
SYS "Wimp_SendMessage",8,blk,quitsender
```

where quitsender is read from sender field of original PreQuit message.

The Task Manager uses the Quit and PreQuit messages when the user selects the Exit option from its menu. The way in which this works (in pseudo-BASIC) is as follows:

```
REM in CASE statement for Wimp_Poll event type...
WHEN Menu_Selection : PROCdecodeMenu
IF menuChoice$="Exit" THEN
  REM send the PreQuit and remember my_ref
  SYS "Wimp_SendMessage",User_Message_Recorded,PreQuitBlock,0
  PreQuitRef = PreQuitBlock!8
ENDIF
WHEN User_Message_Acknowledge
  REM got one of our messages back. Is it the PreQuit one?
  IF pollBlock!8 = PreQuitRef THEN
    REM no-one objected to PreQuit so safe to issue quit
    SYS"Wimp_SendMessage",User_Message_Recorded,quitBlock,0
    quitRef=quitBlock!8
  ELSE REM is it the quit one then?
    REM if so, exit the Desktop
    IF pollBlk!16=Message_Quit AND pollBlk!8=quitRef THEN quit
  ENDIF

WHEN User_Message, User_Message_Recorded
  REM if someone else did a quit, then terminate desktop
  IF pollBlk!16=Message_Quit AND pollBlk!8<>quitRef THEN quit
...
```

In English, the Task Manager issues a PreQuit broadcast when the Exit item is selected from its menu. If this is returned by the Wimp (because no other task objected), the Task Manager goes ahead and issues a Quit broadcast. When this comes back unacknowledged, the Task Manager checks the reference and quits if it is correct (as all other tasks would already have done).

The Task Manager must also be able to respond to the key-pressed event (Ctrl-Shift-F12) &1FC.

Tasks should automatically restart the quit procedures as described earlier.

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

Restarting the desktop closedown sequence

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

This facility is not available in RISC OS 2.

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

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

Message_PaletteChange (9)

This broadcast message is issued by the Palette utility. It should not be acknowledged. The utility generates it when the user finishes dragging one of the RGB bars for a given colour, or when a new palette file is loaded.

If a task needs to adapt to a change in the physical colours on the screen, it should respond to this message by changing any of its internal tables (colour maps etc), and then call Wimp_ForceRedraw to ensure that its windows are redrawn with the new colours. Note though that the palette utility automatically forces a redraw of the whole screen if any of the Wimp's standard colours change their logical mapping, so applications don't have to take further action.

This message is not issued when the Wimp mode changes; Message_ModeChange (&400C1) reports this, so tasks interested in colour mapping changes should recognise this message too.

Message_SaveDesktop (10)

See the section entitled *The desktop save protocol* on page 3-243 for details of this.

Message_Shutdown (14)

R1+0	24 (size)
R1+16	Message_Shutdown (10)
R1+20	flags (all reserved)

This message is issued when the computer is being forced to shutdown, say due to power failure on a portable machine. It is broadcast as a result of calling the SWI TaskManager_Shutdown (page 3-315) with bit 3 of R0 set. Applications receiving this should attempt to ensure any unsaved data.

This facility is not available in RISC OS 2.

Filer messages

Message_FilerOpenDir (&400)

A task sends this message to a Filer task. It is a request to open a new directory display. The data part of the message block is as follows:

R1+20	filing system number
R1+24	bit 0 set ⇒ do not canonicalise name before using all other bits reserved
R1+28	full name of directory to view, zero-terminated

The string given at R1+28 must be a full specification of the directory to open including fileserver (if appropriate), disc name, and pathname starting from $, using the same format as the names in Filer windows. Send the message as a broadcast User_Message. If the directory name is invalid (e.g. the filing system is not present), a Wimp_ReportError error will be generated by the Filer.

Note that the Filing System modules (eg. ADFSFiler) do not use a broadcast, but instead discover the Filer's task handle by means of the Service_StartFiler protocol. See the section entitled *Relocatable module tasks* on page 3-60 for further details.

Message_FilerCloseDir (&401)

This message takes the same form as the previous one. All open directory displays whose names start with the name given at R1+28 are closed.

Message_FilerOpenDirAt (&402)

This is similar to the Filer_OpenDir message but allows you to specify the position and mode for the directory viewer. The format of the message block is as follows:

R1+20	filing system number
R1+24	must be 0
R1+28	X position of viewer
R1+32	Y position of viewer
R1+36	width of viewer
R1+40	height of viewer
R1+44	viewmode: bits 0-1 display mode 0 = large icons 1 = small icons 2 = full info 3 – reserved, do not use bits 2-3 sort mode 0 = sort by name 1 = sort by size 2 = sort by type 3 = sort by date bit 4 0 = use default display mode 1 = use display mode in bits 0-1 bit 5 0 = use default sort mode 1 = use sort mode in bits 2-3 all other bits reserved and must be 0
R1+25	full name of directory to view

This message is not available in RISC OS 2.

Filer Action Window

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

The Filer Action Window is not available in RISC OS 2.

To drive Filer_Action you must:

1 Wimp_StartTask with a command of *Filer_Action

2 Send a sequence of messages to the new task describing the activity:

- specify the directory in which the objects that are going to be acted upon exist (using Message_FilerSelectionDirectory);
- specify the objects in the directory (using several Message_FilerAddSelection messages);
- start the action using Message_FilerAction.

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

Controlling the Filer_Action task

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

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

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

FilerAction starts the operation going.

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

Message_FilerSelectionDirectory (&403)

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

This message is not available in RISC OS 2.

Message_FilerAddSelection (&404)

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

This message is not available in RISC OS 2.

Message_FilerAction (&405)

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

Word	Meaning		
0	Operation to be performed:		
	0	Copy	Copy a number of objects from one directory to another
	1	Move (rename)	Move a number of objects from one directory to another by trying a rename first then doing a copy/delete if that fails
	2	Delete	Delete a number of objects in a particular directory
	3	Set access	Set the access of a number of objects to a given value
	4	Set type	Set the file type of a number of objects to a given value
	5	Count	Count the file sizes of the selected objects
	6	Move (by copying and deleting afterwards)	Move a number of objects from one directory by copying them then deleting the source
	7	Copy local (within directory)	Copy a single object to a different name in the same directory
	8	Stamp files	Stamp the selected objects with the time when they get stamped
	9	Find file	Find an object with a given name.
1	Option bits:		

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

Word	Meaning

2 onwards

Information specific to the particular operation:

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

This message is not available in RISC OS 2.

Message_FilerControlAction (&406)

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

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

This message is not available in RISC OS 2.

Message_FilerSelection (&407)

This message is sent by the filer to the application, before it starts sending DataLoad messages when a selection has been dragged from the filer to an application. The data block of the message is as follows:

R1+20	x0 of selection bounding box in screen coordinates
R1+24	y0 of selection bounding box in screen coordinates
R1+28	x1 of selection bounding box in screen coordinates
R1+32	y1 of selection bounding box in screen coordinates
R1+36	width of each selected item
R1+40	height of each selected item
R1+44	view mode for this directory:
	bits 0-1 display mode
	0 = large icons
	1 = small icons
	2 = full info
	3 – reserved, do not use
	bits 2-3 sort mode
	0 = sort by name
	1 = sort by size
	2 = sort by type
	3 = sort by date
R1+48	start column of selection in window
R1+52	start row of selection in window
R1+56	end column of selection in window
R1+60	end row of selection in window

This message is not available in RISC OS 2.

NetFiler message

Message_Notify (&40040)

The NetFiler sends this broadcast message to enable an application to display the text of a *Notify command in some pleasing way. If no-one acknowledges the message, NetFiler simply displays the text in a window using Wimp_ReportError, with the string `Message from station xxx.xxx` in the Title Bar.

Information about the sender, and the text of the notify, are contained in the message block, as follows:

R1+20	sending station number
R1+21	sending station network number
R1+22	LSB of five byte real time on receipt of message
R1+23	second byte of time
R1+24	third byte of time
R1+25	fourth byte of time
R1+26	MSB of five byte real time on receipt of message
R1+27	message text, terminated by a zero byte

So if you want to do something with the notify and prevent the NetFiler from displaying it, copy the my_ref field into the your_ref field and send the message back using Wimp_SendMessage User_Message_Acknowledge (19).

Wimp messages

Message_MenuWarning (&400C0)

The Wimp sends this message when the mouse pointer travels over the right arrow of a menu item to activate a submenu. The menu item must have its ‘generate message’ bit (3) in the menu flags set for this to happen, otherwise the Wimp will just open the submenu item as normal. (The submenu pointer must also be greater than zero in order for this message to be sent.)

In the message block are the values required by Wimp_CreateSubMenu (page 3-196) on entry. The task may use these, or may choose to take some other action (e.g. create a new window and open that as the submenu).

R1+20	submenu pointer from menu item
R1+24	x coordinate of top left of new submenu
R1+28	y coordinate of top left of new submenu
R1+32	main menu selected item number (0 for first)
R1+36	first submenu selected item number
…	
R1+…	–1 to terminate list

After the three words required by Wimp_CreateSubMenu is a description of the current selection state, in the same format that would be returned by the Menu_Selection event. This information, in conjunction with the task’s knowledge of the menu structure, is sufficient to work out the path taken through the menu so far.

Message_ModeChange (&400C1)

Wimp_SetMode (page 3-185) causes this message to be sent as a broadcast. It gives tasks a chance to update their idea of what the current screen mode looks like by reading the appropriate parameters using OS_ReadVduVariables (page 1-730). (Though applications should need to know as little about the display's attributes as possible to facilitate mode independence.)

You should not acknowledge this message.

After sending the message, the Wimp generates an Open_Window_Request event for each window that was active when the mode change occurred. This is because going from a wider to a narrower mode (e.g. 16 to 12) may require the horizontal coordinates of windows to be compressed to fit them all on to the new display. The whole screen area is also marked invalid to force a redraw of each window's contents.

You should take care if, on a mode change, you modify a window in a way that involves deleting it and then recreating with different attributes. This will result in the handle of the window changing just after the Wimp scans the window stack and generates the Open_Window_Request for it, but before it is delivered from Wimp_Poll, and the Wimp will use the wrong handle. In this situation, you should internally mark the window as 'to be recreated' on receipt of the ModeChange message, and then when you receive the Open_Window_Request for that window, carry out the delete/recreate/open action then.

Message_TaskInitialise (&400C2)

This message is broadcast whenever a task calls Wimp_Initialise. It is used by the Task Manager to maintain its list of active tasks. Information in the message block is as follows:

R1+4	new task handle (so it appears that the new task sent the message)
…	
R1+20	CAO (current active object) pointer of new task
R1+24	amount of application memory used by the task
R1+28	task name, as given to Wimp_Initialise, control-char-terminated

Message_TaskCloseDown (&400C3)

This performs a similar task to the one above, keeping the Task Manager (and any other interested parties) informed about the state of a task. It is generated by the Wimp on the task's behalf when it calls Wimp_CloseDown. If a program 'accidentally' calls OS_Exit before calling Wimp_CloseDown, the Wimp will perform the latter action for it. The message block is standard except for

R1+4	dying task's handle

i.e. the Wimp makes it look as though the task sent the message itself.

Message_SlotSize (&400C4)

This broadcast is issued whenever Wimp_SlotSize is called. Again, its primary client is the task manager, enabling that program to keep its display up to date. The message block looks like this:

R1+4	handle of the task which owns the current slot
…	
R1+20	new current slot size
R1+24	new next slot size

As with most broadcast messages, you should not acknowledge this one.

Message_SetSlot (&400C5)

This message has two uses. First it allows the Task Manager to discover if an application can cope with a dynamically varying slot size. Second, it is used by the Task Manager to tell a task to change that size if it can.

The message block contains the following:

R1+20	new current slot size
R1+24	handle of task whose slot should be changed

The receiver should check the handle at R1+24, and the size at R1+20. If the handle is not the task's, it should do nothing (i.e. no acknowledgement).

If the slot size is big enough for the task to carry on running, it should set R0 to this, R1 to –1 and call Wimp_SlotSize (page 3-203). It should then acknowledge the message.

If the slot size is too small for the task to carry on running, it should not call Wimp_SlotSize, but should acknowledge the message if it wants to continue to receive these messages. If ever a Message_SetSlot is not acknowledged, the Task Manager makes that task an undraggable one on its display.

You should be prepared to receive negative values for the slot size (which of course you shouldn't pass to Wimp_SlotSize), so do a proper signed comparison when checking the value in R1+20.

Message_TaskNameRq (&400C6)

This forms the first of a pair of messages that can be used to find the name of a task given the handle. An application should broadcast this message. It will be picked up by the Task Manager, if running. The Task Manager will respond with a TaskNameIs message (see below). The message block should contain the following information:

R1+20	handle of task whose name is required

Message_TaskNameIs (&400C7)

The Task Manager responds to a TaskNameRq message by sending this message. The message block contains the following:

R1+20	handle of task whose name is required
R1+24	task's slot size
R1+28	task's Wimp_Initialise name, control-char-terminated

The principle user of this message-pair is the !Help application in providing help about ROM modules.

In RISC OS 3 you should use the SWI TaskManager_TaskNameFromHandle (see page 3-312) in preference to these messages.

Message_TaskStarted (&400C8)

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

This message is not available under RISC OS 2.

Message_MenusDeleted (&400C9)

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

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

In the case of the former two, the message is only sent after the new menu is created.

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

This message is not available under RISC OS 2.

Application messages

Alarm

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

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

Message_AlarmSet (&500)

Setting an alarm

To set an application alarm, send the following message:

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

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

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

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

Cancelling an alarm

To cancel the alarm, use the following message block:

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

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

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

This message is not available in RISC OS 2.

Message_AlarmGoneOff (&501)

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

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

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

This message is not available in RISC OS 2.

Help

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

Message_HelpRequest (&502)

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

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

Locations 20 onwards are the results of using Wimp_GetPointerInfo.

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

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

Message_HelpReply (&503)

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

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

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

The help text is terminated by a null character.

The desktop save protocol

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

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

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

3 The Task Manager then inserts four *commands:

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

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

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

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

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

5 For each task in its list, the task manager sends a Message_SaveDesktop:

Message_SaveDesktop (10)

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

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

This facility is not available in RISC OS 2.

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

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

A typical example for a C application follows:

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

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

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

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

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

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

Resident modules

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

There is a service call provided for modules which need to save some state to the file, e.g. ColourTrans saves its calibration. For details of this call see the section entitled *Service_WimpSaveDesktop (Service Call &5C)* on page 3-76.

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

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

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

The device claim protocol

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

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

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

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

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

This protocol can be used under RISC OS 2, but will not be used by applications written for it, such as printer drivers prior to version 2.42.

Device Numbers

Currently allocated device numbers are:

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

Example

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

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

The same procedure is followed for the parallel port.

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

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

Message_DeviceClaim (11)

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

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

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

Message_DeviceInUse (12)

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

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

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

Data transfer protocol

The message-passing system is central to the transfer of data around the Wimp system. This covers saving files from applications, loading files into applications, and the direct transfer of data from one application to another. The last use often obviates the need for a 'scrap' (cut and paste) mechanism for intermediate storage; data is sent straight from one program to another, either via memory or a temporary file.

Data transfer code uses an environment variable called `Wimp$Scrap` to obtain the name of the file which should be used for temporary storage. This is set by the file `!Scrap.!Boot`, when a directory display containing the `!Scrap` directory is first displayed. (Under RISC OS 2 this was done by the file `!System.!Boot`, when a directory display containing the `!System` directory is first displayed.) Applications attempting data transfer should check that `Wimp$Scrap` exists. If it doesn't, they should report the error `Wimp$Scrap not defined`.

Four main message types exist to enable programs to support file/data transfer. The protocol which uses them has been designed so that a save to file operation looks very similar to a data transfer to another application. Similarly, a load operation bears much similarity to a transfer from another program. This minimises the amount of code that has to be written to deal with all possibilities.

The messages types are:

1	Message_DataSave
2	Message_DataSaveAck
3	Message_DataLoad
4	Message_DataLoadAck

There are three others which have associated uses: Message_DataOpen, Message_RamFetch and Message_RamTransmit. Before describing the message types in detail, we describe the four data transfer operations.

Note that all messages except for the initiating one should quote the other side's my_ref field in the message's your_ref field, as is usual when replying.

Saving data to a file

This is initiated through a Save entry in a task's menu. This item will have a standard dialogue box, with a 'leaf' name and a file icon which the user can drag to somewhere on the desktop, in this case a directory window. The following happens:

1. The user releases the mouse button, terminating the drag of the file icon; the application receives a User_Drag_Box event.
2. The application calls Wimp_GetPointerInfo (page 3-140) to find out where the icon was dropped, in terms of its coordinates and window/icon handles.
3. The application sends a DataSave message with the file's leafname to the Filer using this information.
4. The Filer replies with a DataSaveAck message, which contains the complete pathname of the file.
5. The application saves the data to that file.
6. The application sends the message DataLoad to the Filer.
7. The Filer replies with the message DataLoadAck.

The last two steps may seem superfluous, but they are important in keeping the application-Filer and application-application protocol the same.

Saving data to another application

This is initiated in the same way as a Filer save. The following happens:

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

You can see now that the saving task doesn't need to know whether it is sending to the Filer or something else. In its initial DataSave message, it just uses the window/icon handles returned by Wimp_GetPointerInfo as the destination task (in R2/R3) and the Wimp does the rest. It must, of course, always use the pathname returned in the DataSaveAck message when saving its data.

Loading data from a file

This is very straightforward. A load is initiated by the Filer when the user drags a file icon into an application window or icon bar icon.

1 The Filer sends the DataLoad message to the application.

2 The application loads the named file and replies with a DataLoadAck message.

The receiving task is told the window and icon handles of the destination. From this it can decide whether to open a new window for the file (the file was dragged to the icon bar) or insert it into an existing window.

Loading data from another application

This is simply the case of saving data to another application, but from the point of view of the receiver:

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

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

3 The external task saves its data to that file.

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

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

6 The application replies with the message DataLoadAck to the external task.

Again, the receiver can decide what to do with the incoming data from the destination window and icon handles.

The messages used in the above descriptions are described below. Messages 1 and 3 are generally sent as User_Message_Recorded, because they expect a reply, and types 2 and 4 are sent as User_Message, as they don't. The message blocks are designed so that a reply can always use the previously received message's block just by altering a couple of fields.

When receiving any message, allow for either type 17 or 18, i.e. don't rely on any sender using one type or the other.

Message_DataSave (1)

The data part of the message block is as follows:

R1+20	destination window handle	
R1+24	destination icon handle	
R1+28	destination x coordinate	(screen coordinates, i.e. not
R1+32	destination y coordinate	relative to the window)
R1+36	estimated size of data in bytes	
R1+40	file type of data	
R1+44	proposed leafname of data, zero-terminated	

The first four words come from Wimp_GetPointerInfo. The rest should be filled in by the saving task. In addition to the usual &xxx file types, the following are defined for use within the data transfer protocol:

&1000	directory
&2000	application directory
&ffffffff	untyped file (i.e. had load/exec address)

Message_DataSaveAck (2)

The message block is as follows:

R1+12	my_ref field of the DataSave message
…	
R1+20	destination window handle
R1+24	destination icon handle
R1+28	destination x coordinate
R1+32	destination y coordinate
R1+36	estimated size of data in bytes; –1 if file is ‘unsafe’
R1+40	file type of data
R1+44	full pathname of data (or `Wimp$Scrap`), zero-terminated

The words at +20 to +32 are preserved from the DataSave message. If the receiver of the file (i.e. the sender of this message) is not the Filer, then it should set the word at +36 to –1. This tells the file’s saver that its data is not ‘secure’, i.e. is not going to end up in a permanent file. In turn the saver will not mark the file as unmodified, and will not use the returned pathname as the document’s window title.

The Filer, on the other hand, will not put –1 in this word, and will insert the file’s full pathname at +44. The saver can mark its data as unmodified (since the last save) and use the name as the document window title.

Message_DataLoad (3)

From the foregoing descriptions you can see that this message is used in two situations, firstly by the Filer when it wants an application to load a file, and secondly by a task doing a save to indicate that it has written the data to `<Wimp$Scrap>`. The message block looks like this:

R1+12	my_ref from DataSaveAck message, or 0 if from Filer
…	
R1+20	destination window handle
R1+24	destination icon handle
R1+28	destination x coordinate
R1+32	destination y coordinate
R1+36	estimated size of data in bytes
R1+40	file type
R1+44	full pathname of file, zero terminated

The receiver of this message should check the file type and load it if possible. After a successful load it should reply with a Message_DataLoadAck.

If the sender of this message does not receive an acknowledgement, it should delete `<Wimp$Scrap>` and generate an error of the form `Data transfer failed: Receiver died`.

In RISC OS 3 when the filer sends a data load to an application it appends the position of the file in the current selection to the end of the message so the format of the block becomes:

R1+44	full pathname of file, zero terminated
R1+*n*	column of file in current selection
R1+*n*+4	row of file in current selection

(where *n* is the length of the full pathname and terminator, plus any padding needed to word align the next entry)

You can check for the existence of these values by comparing the size field of the message with the position of the terminating zero of the pathname.

Message_DataLoadAck (4)

R1+12	my_ref from DataLoad message
…	
R1+20	destination window handle
R1+24	destination icon handle
R1+28	destination x coordinate
R1+32	destination y coordinate

R1+36	estimated size of data in bytes
R1+40	file type
R1+44	full pathname of file, zero terminated

Effectively, the file-loading task just changes the message type to 4 and fills in the your_ref field, then sends back the previous DataLoad message to its originator.

Message_DataSaved (13)

R1+12	reference from DataSave message
R1+16	13

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

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

Memory data transfer

The foregoing descriptions rely on the use of the Wimp scrap file. However, task to task transfers can be made much quicker by transferring the data within memory. The save and load protocols are modified as below to cope with this.

Saving data to another application (memory)

This is the same as previously described in the section entitled *Saving data to another application* on page 3-248 up until the DataSave message. Then:

1 The external task replies with a RAMFetch message.

2 The application sends a RAMTransmit message with data.

3 The external task replies with another RAMFetch message.

4 The last two steps continue until all the data has been sent and received.

Loading data from another application (memory)

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

2 The application replies with a RAMFetch message.

3 If this isn't acknowledged with a RAMTransmit, use the <Wimp$Scrap> file to perform the operation, otherwise…

4 Get and process the data from the RAMTransmit buffer.

5 While the RAMTransmit buffer is full:

Send a RAMFetch for more data
Get and process the data from the RAMTransmit buffer.

So if the first RAMFetch message is not acknowledged (i.e. it gets returned as a User_Message_Acknowledge), the data receiver should revert to the file transfer method. If any of the subsequent RAMFetches are unanswered (by RAMTransmits), the transfer should be aborted, but no error will be generated. This is because the sender will have already reported an error to the user.

The data itself is transferred by the sender calling Wimp_TransferBlock (page 3-214) just before it sends the RAMTransmit message. See the description of that call for details of entry and exit conditions.

The termination condition for the saver generating RAMTransmits and the loader sending RAMFetches is that the buffer is not full. This implies that if the amount of data sent is an exact multiple of the buffer size, there should be a final pair of messages where the number of bytes sent is 0.

Here are the message blocks for the two messages:

Message_RAMFetch (6)

R1+12	my_ref field of DataSave/RAMTransmit message
…	
R1+20	buffer address for Message_RAMTransmit
R1+24	buffer length in bytes

This is sent as a User_Message_Recorded so that a lack of reply to the first one results in the file transfer protocol being used instead, and a lack of reply to subsequent ones allows the transfer to be abandoned. No error should be generated because the other end will have already reported one. A reply to a RAMFetch takes the form of a RAMTransmit from the other task. The receiver should also generate an error if it can't process the received data, e.g. if it runs out of memory. This should also cause it to stop sending RAMFetch messages.

When allocating its buffer, the receiver can use the estimated data size from the DataSave message, but it should be prepared for more data to actually be sent.

Message_RAMTransmit (7)

R1+12	my_ref field of RAMFetch message
…	
R1+20	buffer address from RAMFetch message
R1+24	number of bytes written into the buffer

A data-saving task sends this message in response to a RAMFetch if it can cope with the memory transfer protocol. If the number of bytes transferred into the buffer (using Wimp_TransferBlock) is smaller than the buffer size, then this is the last such message, otherwise there is more to send and the receiver will send another RAMFetch message.

All but the last messages of this type should be sent as User_Message_Recorded types. If there is no acknowledgement, the sender should abort the data transfer and stop sending. It may also give an error message. The last message of this type (which may also be the first if the buffer is big enough) should be sent as a User_Message as there will be no further RAMFetch from the receiver to act as acknowledgement.

The iconise protocol

This protocol is not available in RISC OS 2.

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

If Shift is held down when the Close icon of a window is clicked, the Wimp does not close the window, but instead broadcasts a Message_Iconize.

If no iconiser is loaded nothing happens.

If an iconiser is loaded:

1 It acknowledges the message (stops the broadcast).

2 It sends a Message_WindowInfo to the window.

Old application

If the application is an old RISC OS 2 one, it will ignore the above message.

The iconiser gets acknowledgement back and uses the information in the first Message_Iconize to iconise the window.

New application

If the application is a new RISC OS 3 application it should react as follows:

- If it doesn’t want to help it should ignore the message.

- If it wants to help it should reply with a Message_WindowInfo. The iconiser will then use this information to iconise the window.

  This enables applications such as Edit to give a different icon depending on the file type of the file being edited in the window.
- If the application wants to iconise its own window it should acknowledge the original Window_Info message, and do all the work itself.

Closing a window

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

The iconiser then removes the icon.

When a task exits

The iconiser spots the Message_TaskQuit and remove all the icons for that task.

When a new iconiser starts up

It broadcasts a Message_WindowInfo with a window handle of 0.

An iconiser receiving this message should reopen all iconised windows.

All applications should ignore such a message.

Current iconiser (Pinboard) behaviour

If it does not get a reply to the Message_WindowInfo

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

If it gets a Message_WindowInfo

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

Message_Iconize (&400CA)

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

This message is not available in RISC OS 2.

Message_WindowClosed (&400CB)

R1+20 window handle
R1+24

This message is not available in RISC OS 2.

Message_WindowInf (&400CC)

R1+20 window handle
R1+24 reserved, must be 0
R1+28 sprite name to use, null terminated (MAX = 7 chars + NULL)
sprite name used is `icon_`*`string`*
R1+36 title string to use null terminated (as short as possible truncated to 20 characters)

This message is not available in RISC OS 2.

The Printer protocol

The printer protocol is used to ensure a uniform procedure by which an application may print files of any type, allowing for the files to be printed immediately or queued by other software for printing later. The printer manager Printers, supplied with RISC OS 3, uses this protocol. The description below assumes that the dialogue is being conducted between an application and Printers (to avoid referring repeatedly to 'the destination of the printer protocol').

The protocol for printing a file is

1 The application issues either:

- DataSave (the user has dropped a file onto the Printers icon)
- PrintSave (the user has initiated an application **Print** option).

2 If Printers is loaded, it replies with either:

- PrintError if there is an error (the protocol is now over)
- PrintFile.

This stage is present for compatibility with RISC OS 2 applications.

If Printers is not loaded, the message bounces. In this case, the application should do one of two things:

- If it was graphics printing, go ahead and try to print anyway.
- If it was text printing, complain that the printer manager is required.

3 The application does one of the following:

- ignores PrintFile (this is the normal behaviour under RISC OS 3)
- replies with WillPrint and prints the file (this is the RISC OS 2 behaviour, now deprecated – the application has 'jumped the queue')
- converts the file, stores the output in Printer$Temp, and replies with DataLoad.

In the first case, the protocol continues as described below.

4 Printers responds in one of two ways, depending on whether the destination printer is in use or not, by issuing either:

- PrintTypeOdd (requesting the application to print the file itself immediately)
- DataSaveAck (requesting the application to send the file to Printers for queuing).

In the first case, PrintTypeOdd is **not** broadcast, and does **not** contain valid file type or file name fields. These must not be relied on.

5 If PrinterTypeOdd is issued (the first case above), the application:

- may be able to print the file itself immediately, in which case it replies with PrintTypeKnown and prints the file (the protocol is now over)
- may not be able to print the file itself, in which case it ignores the PrintTypeOdd, and Printers responds with a DataSaveAck, so that it can take a copy of the file.

6 Either way, the application has received a DataSaveAck from Printers. It should save the data it wants printed to the file whose name was supplied in the DataSaveAck message, and reply with DataLoad.

7 Printers will respond with DataLoadAck. The file is in the print queue.

At some future time, the file will rise to the top of the queue (unless the user has removed it manually), and Printers will broadcast a PrintTypeOdd to find an application willing to print the file. (This might be the same application as the one that queued the file, or a different one.) If the PrintTypeOdd is not replied to, Printers will issue the SWI

```
sprintf (s, "@PrintType_%3.3X %s", file_type, file_name);
_swi (Wimp_StartTask, _IN (0), s);
```

This should be a command that will cause the application to print the file immediately.

In response to the PrintSave, the printer manager may reply with PrintError (&80144). If the size of this message is 20, this means you are talking to an old printer manager and it is busy. If the size of the message is not 20, there is an error number at offset 20 and null terminated error text at offset 24.

If the application is doing graphics printing, it should print the file **without calling** Wimp_Poll. Wimp_Poll must **not** be called when using Printers because when Printers regains control it assumes that the current file has been printed and moves on to the next entry in the queue.

The protocol as described above is the one implemented by Printers. Future versions of printer managing software may take the copy by RAM transfer from applications that support it. To be ready for this, the application should be prepared for a RAMFetch to be sent in the place of the DataSaveAck described above. (By ignoring this RAMFetch, they will revert to the file-based protocol.)

The techniques and calls used to actually print are outlined in the section entitled *Printing a document from an application* on page 3-568.

Message_PrintFile (&80140)

This message is broadcast as a recorded delivery upon receipt of a DataSave or PrintSave message. The reason for having this message is two-fold:

- the application doing the DataSave might need to know that, in effect, the user is wanting to print;
- it allows applications to try and improve on !Printer-provided services such as text printing.

The format of the message is:

R1+12	your_ref
R1+16	&80140
R1+20	
…	from DataSave/PrintSave block
R1+44	

This allows any application to try and do better than !Printers can do with the default actions available to it. Such an application has 3 options:

- it can ignore the message, in which case, if no-one else claims it, !Printers will resort to the normal processes (i.e. issue a DataSaveAck);
- it can respond with WillPrint, in which case !Printers takes no further action;
- it can convert the file into another format and store it in the file specified by `Printer$Temp`. It should then reply with a DataLoad with the filetype reflecting the new type.

Message_WillPrint (&80141)

This message is sent by an application in response to a PrintFile broadcast. The application should then proceed to print the file.

Note: It is recommended that you use the PrintTypeOdd protocol in preference to this message.

Message_PrintSave (&80142)

The format of this message is:

R1+12	0
R1+16	&80142
R1+20	
…	as for Message_DataSave
R1+44	

This message allows applications to send files to the printer manager for printing without having to know the task handle, etc, since the message is broadcast. The message simply needs to be broadcast as a recorded delivery, at which point the printer manager will enter the PrintFile dialogue. If the message bounces, the application should complain as the printer manager is not loaded.

Message_PrintInit (&80143)

This is broadcast when a printer manager is starting up. Any active printer managers should quit quietly upon receipt of this message to avoid a clash occurring.

Message_PrintError (&80144)

Under RISC OS 2

This message is sent by RISC OS 2 managers in response to a PrintSave if they are already printing (as they can only queue one file at a time). It is known as Message_PrintBusy under RISC OS 2.

Under !Printers

With !Printers, this message is sent if an error occurs as a result of one of the other messages being used. The format of the block is:

R1+12	your_ref
R1+16	&80144
R1+20	error number
R1+24	error message (null terminated)

To maintain compatibility with RISC OS 2 printer managers, if the message is the original Message_PrintBusy, the size (in R1+0) will be 20.

Error numbers and messages

```
1       Can only print from applications when a printer has been selected
```

This is sent in reply to a PrintSave when there isn't a selected printer.

Message_PrintTypeOdd (&80145)

This message is broadcast if the filetype is not considered known by !Printers. 'Known' is qualified as being the current printer type: text (FFF), obey (FEB) or command (FFE) files, TaskExec (FD6), TaskObey (FD7), Desktop (FEA) and 1st Word Plus (AF8). The format of the message is:

R1+12	0
R1+16	&80145
R1+40	file type of data
R1+44	zero terminated filename

If an application can print this filetype directly, it should respond with PrintTypeKnown. The application can either:

- print the file directly to printer:
- output it to `Printer$Temp`, in which case this must be done before replying with PrintTypeKnown.

Currently assigned printer type files are PoScript (FF5) and Printout (FF4).

Message_PrintTypeKnown (&80146)

This message is sent by an application in response to a PrintTypeOdd.

Message_SetPrinter (&80147)

This message is broadcast by !Printers when the printer settings or selection has changed.

Message_PSPrinterQuery (&8014C)

This message is sent as a recorded delivery by !FontPrint to !Printers when !FontPrint either starts up or receives SetPrinter. The layout of the block is:

R1+12	0
R1+16	&8014C
R1+20	buffer address (or zero)
R1+24	buffer size

If the buffer address is non-zero, !Printers places the following information into the buffer (all Null terminated):

- current printer name,
- current printer type,

- pathname to printer font file.

Regardless of the buffer address, !Printers places the real buffer size into the block and replies with PSPrinterAck.

This message is not available in RISC OS 2.

Message_PSPrinterAck (&8014D)

This is sent by !Printers to !FontPrint in response to PSPrinterQuery. If !FontPrint does not receive this message, it should raise an error to advise the user (e.g. !Printers is required to allow use of !FontPrint).

This message is not available in RISC OS 2.

Message_PSPrinterModified (&8014E)

This is sent by !FontPrint to !Printers when the user clicks on the **Save** button. !Printers then re-reads the font file and resets the printer's font list.

This message is not available in RISC OS 2.

Message_PSPrinterDefaults (&8014F)

This is sent by FontPrint to !Printers when the user clicks on the **Default** button. !Printers then resets the font file, resets the printer's font list and replies with PSPrinterDefaulted.

This message is not available in RISC OS 2.

Message_PSPrinterDefaulted (&80150)

This is sent by !Printers to !FontPrint when the font file has been reset.

This message is not available in RISC OS 2.

Message_PSPrinterNotPS (&80151)

This is sent by !Printers upon receipt of PSPrinterQuery if the currently selected printer is not a PostScript printer.

This message is not available in RISC OS 2.

Message_ResetPrinter (&80152)

This can be sent to !Printers to ensure that the printer settings are correct for the currently selected printer.

This message is not available in RISC OS 2.

Message_PSIsFontPrintRunning (&80153)

If !FontPrint receives this message, it will acknowledge it.

This message is not available in RISC OS 2.

The DataOpen Message

Message_DataOpen (5)

This message is broadcast by the Filer when the user double-clicks on a file. It gives active applications which recognise the file type a chance to load the file in a new window, instead of having the Filer launch a new copy of the program.

The message block looks like this:

R1+20	window handle of directory display containing file
R1+24	unused
R1+28	x offset of file icon that was double clicked
R1+32	y offset of file icon
R1+36	0
R1+40	file type
R1+44	full pathname of file, zero-terminated

The x and y offsets can be used to display a ‘zoom-box’ from the original icon to the new window, to give a dynamic impression of the file being opened.

If the user double-clicks on a directory with Shift held down, this message will be broadcast with the file type set to &1000.

The file type is set to &3000 for untyped files.

The application should respond by loading the file if it can, and acknowledging the message with a Message_LoadDataAck. If no-one loads the file, the Filer will *Run it.

Note that once the resident application has decided to load the file, it should immediately acknowledge the Data Open message. This is so that if the load fails with an error (eg. `Memory full`), the Filer will not then try to *Run the file. This would only result in another error message anyway.

TaskWindow messages

TaskWindow_Input (&808C0)

This message is used to send input data from Parent to Child.

R1+20	size of input data
R1+24	pointer to input data

Input can also be sent via a normal RAM transfer protocol, i.e. send a Message_DataSave, then perform the following two steps until all the data has been sent and received:

1 wait for Message_RAMFetch

2 send back Message_RAMTransmit

See the section entitled *Memory data transfer* on page 3-252 for a full description of this protocol.

TaskWindow_Output (&808C1)

This message is sent to the Parent when one of its children has produced output.

R1+20	size of output data
R1+24...	output data

TaskWindow_Ego (&808C2)

This message is sent to the Parent, to inform him of the Child's task-id.

R1+4	Child's task-id (as filled in by Wimp)
R1+20	Parent's txt-handle (as passed to *TaskWindow or *ShellCLI_Task)

Note that this is the only time the txt-handle is used. It allows the Parent to identify which Child is announcing its task-id.

TaskWindow_Morio (&808C3)

This message is sent to the Parent when the Child exits.

No data (all necessary information is in the wimp message header).

TaskWindow_Morite (&808C4)

This message is sent by the Parent to kill the Child.

No data (all necessary information is in the wimp message header).

TaskWindow_NewTask (&808C5)

This message is broadcast by an external task which requires an application (e.g. Edit) to start up a task window. If the receiving application wishes to deal with this request, it should first acknowledge the Wimp message, then issue a SWI Wimp_StartTask with R1+20… as the command.

R1+20… the command to run

TaskWindow_Suspend (&808C6)

This message is sent by the Parent to suspend a Child.

No data (all necessary information is in the wimp message header).

TaskWindow_Resume (&808C7)

This message is sent by the Parent to resume a suspended Child.

No data (all necessary information is in the wimp message header).

*Commands

*Configure WimpAutoMenuDelay

Sets the configured time before a submenu is automatically opened

Syntax

`*Configure WimpAutoMenuDelay` *`delay`*

Parameters

`delay`	time before a submenu is automatically opened, in $^1/_{10}$ second units

Use

*Configure WimpAutoMenuDelay sets the configured time the pointer must rest over a menu item before its submenu (if any) is automatically opened.

Note that automatic opening of submenus is disabled if bit 7 of the WimpFlags is clear.

This command is not available under RISC OS 2.

Example

```
*Configure WimpAutoMenuDelay 5
```

Related commands

*Configure WimpFlags, *Configure WimpMenuDragDelay

*Configure WimpDoubleClickDelay

Sets the configured time during which a double click is accepted

Syntax

```
*Configure WimpDoubleClickDelay delay
```

Parameters

delay	time during which a double click is accepted, in $^1/_{10}$ second units

Use

*Configure WimpDoubleClickDelay sets the configured time after a single click during which a double click is accepted.

A pending double-click will be immediately cancelled if any of the following occur:

- Wimp_DragBox is called (for example, in response to a drag button event);
- the pointer moves by more than the configured number of OS units;
- the mouse is not clicked again inside the configured amount of time.

This command is not available under RISC OS 2.

Example

```
*Configure WimpDoubleClickDelay 12
```

Related commands

*Configure WimpDoubleClickMove

*Configure WimpDoubleClickMove

Sets the configured distance within which a double click is accepted

Syntax

```
*Configure WimpDoubleClickMove distance
```

Parameters

distance	distance within which a double click is accepted, in OS units

Use

*Configure WimpDoubleClickMove sets the configured distance from the position of a single click within which a double click is accepted.

If the pointer moves this distance or further from the first click, the double click is cancelled.

This command is not available under RISC OS 2.

Example

```
*Configure WimpDoubleClickMove 20
```

Related commands

*Configure WimpDoubleClickDelay

*Configure WimpDragDelay

Sets the configured time after which a drag is started

Syntax

```
*Configure WimpDragDelay delay
```

Parameters

delay	time after which a drag is started, in $^1/_{10}$ second units

Use

*Configure WimpDragDelay sets the configured time after a single click after which a drag is started.

This command is not available under RISC OS 2.

Example

```
*Configure WimpDragDelay 8
```

Related commands

*Configure WimpDragMove

*Configure WimpDragMove

Sets the configured distance the pointer has to move for a drag to be started

Syntax

```
*Configure WimpDragMove distance
```

Parameters

distance	distance the pointer has to move for a drag to be started, in OS units

Use

*Configure WimpDragMove sets the configured distance from the position of a single click that the pointer has to move for a drag to be started.

This command is not available under RISC OS 2.

Example

```
*Configure WimpDragMove 40
```

Related commands

*Configure WimpDragDelay

*Configure WimpFlags

Sets the configured behaviour of windows when dragged, and of error boxes

Syntax

```
*Configure WimpFlags n
```

Parameter

n a value between 0 and 255, as follows:

Bit	Meaning when set
0	window position drags are continuously redrawn
1	window resizing drags are continuously redrawn
2	horizontal scroll drags are continuously redrawn
3	vertical scroll drags are continuously redrawn
4	no beep is generated when an error box appears
5	windows can be dragged partly off screen to right and bottom (not available under RISC OS 2)
6	windows can be dragged partly off screen in all directions (not available under RISC OS 2)
7	open submenus automatically If set and the pointer is kept on a non-leaf menu item for more than the time specified by *Configure WimpAutoMenuDelay then the submenu will be opened automatically by the Wimp (not available under RISC OS 2).

The effect of clearing bits 0 - 3 is that the drag operation is performed using an outline, and the window is redrawn at the end of the drag.

Use

*Configure WimpFlags sets the configured behaviour of windows when dragged, and of error boxes. Generally, all of bits 0 - 3 will be either set or cleared, depending on whether the user requires continuous updates or outline dragging. Bit 4 controls the action of the standard Wimp error reporting window. Bits 5 and 6 control whether the window can move partly off screen (even if bit 6 is clear). Bit 7 controls whether submenus are automatically opened when the pointer rests over their parent entry for longer than the configured WimpAutoMenuDelay.

Examples

```
*Configure WimpFlags 0
*Configure WimpFlags 15
```

Related commands

*Configure WimpAutoMenuDelay, *Status WimpFlags

Related SWIs

Wimp_Poll, Wimp_OpenWindow, Wimp_ReportError

*Configure WimpMenuDragDelay

Sets the configured time before an automatically opened submenu is closed

Syntax

```
*Configure WimpMenuDragDelay delay
```

Parameters

delay	time before an automatically opened submenu is closed, in $^1/_{10}$ second units

Use

*Configure WimpMenuDragDelay sets the configured time before an automatically opened submenu is closed. During this time you can move the pointer over other menu entries without closing the submenu, making it easy to reach the submenu.

Note that automatic opening of submenus is disabled if bit 7 of the WimpFlags is clear.

This facility is not available under RISC OS 2.

Example

```
*Configure WimpMenuDragDelay 7
```

Related commands

*Configure WimpFlags, *Configure WimpMenuDragDelay

*Configure WimpMode

Sets the configured screen mode used

Syntax

`*Configure WimpMode` *`screen_mode`*`|Auto`

Parameter

`screen_mode`	the display mode that the computer should use after a power-on or hard reset, and when entering or leaving the desktop
`Auto`	automatic setting of appropriate mode using monitor lead

Use

*Configure WimpMode sets the configured screen mode used by the machine when it is first switched on, or after a hard reset, and when entering or leaving the desktop. It is identical to the command *Configure Mode; the two commands alter the same value in CMOS RAM.

You can also set a value of Auto (not available in RISC OS 2). More recent Acorn computers can sense the type of monitor lead connected, and hence set an appropriate mode. If no lead can be sensed, either because none is present or because the computer is of an older design, the mode defaults to mode 12.

Under RISC OS 2, this command only sets the configured screen mode used for the Desktop; *Configure Mode sets the configured screen mode used for the command line. If you leave the Desktop and then re-enter it before powering on again or pressing Ctrl Break, the mode used is the one that was last used by the Desktop.

Example

```
*Configure WimpMode 15
```

Related commands

*Configure Mode

Related SWIs

Wimp_SetMode

Related vectors

None

*Desktop

Initialises all desktop facilities, then starts the Desktop

Syntax

```
*Desktop [command|–File filename]
```

Parameters

command	a * Command which will be passed to Wimp_StartTask when the Desktop starts up
filename	a valid pathname specifying a file, each line of which will be passed to Wimp_StartTask when the desktop starts up

Use

*Desktop initialises all desktop facilities, then starts the Desktop. The Desktop provides an environment in which Wimp programs can operate.

*Desktop automatically starts resident Wimp task modules such as the filers, the palette utility and the Task Manager. You can also run an optional * Command or each line of a file of * Commands. This is typically used to load applications such as Edit. Any * Commands using files must specify them by their full pathname.

If you do run a file of * Commands when you start the desktop, its first line should run the file !System!Boot, provided with your computer. This is needed by most desktop applications. If you want to start an application that uses fonts, the next line of the start-up file should run !Fonts.!Boot, again provided with your computer. Applications can then be started on the following lines.

The Desktop may also be configured as the default language, using the command *Configure Language (see page 1-978).

Examples

```
*Desktop
*Desktop !FormEd
*Desktop –File !DeskBoot
```

Related commands

*DeskFS, *Desktop_Filer, *Desktop_ADFSFiler et al.

Related SWIs

Wimp_StartTask

Related vectors

None

*Desktop_…

Commands to start up ROM-resident Desktop utilities

Syntax

`*Desktop_ADFSFiler`, `*Desktop_Configure`, `*Desktop_Draw`, `*Desktop_Edit`, `*Desktop_Filer`, `*Desktop_Free`, `*Desktop_NetFiler`, `*Desktop_Paint`, `*Desktop_Palette`, `*Desktop_Pinboard`, `*Desktop_RAMFSFiler`, `*Desktop_ResourceFiler`, `*Desktop_TaskManager`

Parameters

None

Use

*Desktop_… commands are used by the Desktop to start up ROM-resident Desktop utilities that appear automatically on the icon bar. However, they are for internal use only, and you should not use them; use *Desktop instead. If you do try to use these commands outside the desktop, an error is generated. For example, *Desktop_Palette will give the error message ‘Use *Desktop to start the Palette utility’.

The reason why these commands have to be provided is that it is only possible to start a new Wimp task using a command line.

There is one *Desktop_… command that we’ve documented, because it appears in desktop boot files. This is *Desktop_SetPalette.

Related commands

*Desktop, *Desktop_SetPalette

Related SWIs

Wimp_StartTask

Related vectors

None

*Desktop_SetPalette

Alters the current Wimp palette

Syntax

`*Desktop_SetPalette` *`RGB0 … RGB15 RGBbor RGBptr1 … RGBptr3`*

Parameters

All parameters specify palette entries as 6 hex digits of the form *`BBGGRR`*.

`RGB0 … RGB15`	16 parameters giving the palette values for Wimp colours 0 - 15
`RGBbor`	1 parameter giving the palette value for the border
`RGBptr1 … RGBptr3`	3 parameters giving the palette values for pointer colours 1 - 3

Use

*Desktop_SetPalette alters the current Wimp palette.

Example

```
*Desktop_SetPalette FFFFFF DDDDDD BBBBBB 999999 777777
555555 333333 000000 994400 00EEEE 00CC00 0000DD BBEEEE
008855 00BBFF FFBB00 777777 FFFF00 990000 0000FF
```

Related commands

None

Related SWIs

Wimp_SetPalette (page 3-187)

Related vectors

None

*IconSprites

Merges the sprites in a file with those in the Wimp sprite area

Syntax

`*IconSprites` *`filename`*

Parameters

`filename`	full name of sprite file to load

Use

*IconSprites merges the sprites in a file with those already loaded in the Wimp's shared sprite area. Sprites in this area are used automatically by certain Wimp operations, and because all applications can access them, the need for multiple copies of sprite shapes can be avoided.

Under RISC OS 3 *IconSprites will first try to add a suffix which depends on the properties of the configured Wimp mode, and if this doesn't work will use the original filename as usual.

If the configured Wimp mode is a high resolution mono mode (i.e. bit 4 of the modeflags is set), then it will use the suffix '23'; otherwise the suffix is:

<OS units per pixel (x)><OS units per pixel (y)>'

For example:

Configured Wimp mode	Suffix
23	'23'
20	'22'
12	'24'

This allows applications to provide an alternative set of icons for high resolution mono modes (when using the new Wimp). For example, an application could provide a set of colour sprites in a file called `!Sprites`, and an alternative monochrome set in a file called `!Sprites23`, and then load one set or the other automatically by using `*Iconsprites <Obey$Dir>.Sprites`.

Example

```
*IconSprites <Obey$Dir>.!Sprites
```

Related commands

*Pointer, *SLoad, *SMerge, *SSave, *ToolSprites

Related SWIs

Wimp_SpriteOp

Related vectors

None

*Pointer

Turns the mouse pointer on or off

Syntax

```
*Pointer [0|1]
```

Parameters

`0` or `1` or nothing

Use

*Pointer turns on or off the pointer that appears on screen to reflect the mouse position. If you give either no parameter or a parameter of 1, pointer 1 is set to the default shape held in the Wimp sprite ptr_default (a blue arrow) and the sprite colours are set to their default. The pointer is enabled. If you give a parameter of 0, the pointer is disabled.

Wimp programs that re-program the pointer should use shape 2. Pointer shapes 3 and 4 are used by the Hourglass module.

You can move the pointer with OS_Word 21,5 if the mouse and pointer are unlinked. You can read the pointer position at any time using OS_Word 21,6.

Example

```
*Pointer 0
```
turn off the pointer

Related commands

None

Related SWIs

OS_Word 21 (page 1-710), Wimp_SetPointerShape (page 3-163), Wimp_SpriteOp (page 3-198)

Related vectors

None

*ToolSprites

Merges the sprites in a file with those in the Wimp's pool of border sprites

Syntax

`*ToolSprites` *`filename`*

Parameters

`filename`	full name of sprite file containing tools to load

Use

*ToolSprites merges the sprites in a file with those already loaded in the Wimp's pool of border sprites. Sprites in this area are used by the Wimp to redraw window borders.

If you change the border sprites, you should then force a redraw of the screen by changing mode – even if only to the current mode.

The default border sprites are held in the file Resources:$.Resources.Wimp.Tools, and you may use these as an example. Note that this file does not contain an example of every sprite that the Wimp may use; for further details see the section entitled *RISC OS System Icons* on page 3-25.

Example

```
*ToolSprites <Obey$Dir>.!Sprites
```

Related commands

*IconSprites

Related SWIs

None

Related vectors

None

*WimpMode

Changes the current screen mode used by the Desktop

Syntax

`*WimpMode` *`screen_mode`*

Parameters

`screen_mode`	the display mode that the Desktop should use

Use

*WimpMode changes the current screen mode used by the Desktop.

It does not alter the configured value, which will be used next time the computer is switched on, or after a hard reset, and when entering or leaving the desktop.

Example

```
*WimpMode 20
```

Related commands

*Configure WimpMode

Related SWIs

Wimp_SetMode

Related vectors

None

*WimpPalette

Uses a palette file to set the Wimp's colour palette

Syntax

`*WimpPalette` *`filename`*

Parameters

`filename`	pathname of a file of type &FED (Palette)

Use

*WimpPalette uses a palette file to set the Wimp's colour palette. Typically the file would have been saved using the Desktop's palette utility. If the file is not a Palette file, the error message 'Error in palette file' is generated. If no task is currently active, the palette is simply stored for later use. Otherwise it is enforced immediately.

Palette files can be read in either of two formats:

1. As a list of RGB bytes corresponding to Wimp colours 0 - 15, then the border colour and then the three pointer colours.
2. As a complete VDU sequence, again corresponding to Wimp colours 0 - 15, the border colour and the pointer colours. Typically an entry would be 19,colour,R,G,B.

Type (1) is read for backwards compatibility, but since the palette utility always saves files in format (2), you should use this in preference.

The RunType for Palette files is *WimpPalette %0, so you can also set a new palette from the Desktop simply by double-clicking on the file's icon.

Example

```
*WimpPalette greyScale
```

Related commands

None

Related SWIs

Wimp_SetPalette

Related vectors

None

*WimpSlot

Changes the memory allocation for the current and (optionally) the next Wimp task

Syntax

```
*WimpSlot [–min] minsize[K] [–max maxsize[K]] [-next nextsize[K]]
```

Parameters

minsize	the minimum amount of application space, in bytes or Kilobytes, that the current Wimp application requires
maxsize	the maximum amount of application space, in bytes or Kilobytes, that the current Wimp application requires
nextsize	the size, in bytes or Kilobytes, that will be allocated – if possible – to the next Wimp application

Use

*WimpSlot changes the memory allocation for the current and (optionally) the next Wimp task. It is typically used within Obey files called !Run, which the Filer uses to launch a new Wimp application. *WimpSlot calls Wimp_SlotSize to try to set the application memory slot for the current task to be somewhere between the limits specified in the command.

If there are fewer than minsize bytes free, the error ‘Application needs at least *minsize*K to start up’ is generated.

Otherwise, if the current slot is smaller than minsize, then its size will be increased to minsize. If the current slot is already between minsize and maxsize, then it is unaltered. If a maxsize is specified, and the current slot is larger than maxsize, then its size will be reduced to maxsize.

The slot size that is set by this command will also apply to the application that the *Obey file finally invokes.

The next slot size is automatically saved in a desktop boot file. You can therefore alter the initial default slot either by dragging the **Next** slider in the Task manager’s Task display window before saving a desktop boot file, or by editing the desktop boot file.

Examples

```
*WimpSlot 32K
*WimpSlot –min 150K –max 300K
```

Related commands

*WimpTask

Related SWIs

Wimp_SlotSize

Related vectors

None

*WimpTask

Starts up a new task

Syntax

`*WimpTask` *`command`*

Parameter

`command` * Command which is used to start up the new task

Use

*WimpTask starts up a new task. It simply passes the supplied command to the SWI Wimp_StartTask.

*WimpTask will exit via OS_Exit if you call it from outside a Wimp task.

In RISC OS 2 the command can only be used from within another task.

Example

```
*WimpTask myProg
```

Related commands

*WimpSlot

Related SWIs

Wimp_StartTask

Related vectors

None

*WimpWriteDir

Sets the direction of text entry for writable icons

Syntax

```
*WimpWriteDir 0|1
```

Parameters

`0`	write direction is the default for the current territory
1	write direction is the reverse of the default for the current territory

Use

*WimpWriteDir sets the direction of text entry for writable icons to either the default for the current territory, or the reverse of that.

It also affects the direction in which text inside text icons is printed.

This facility is not available under RISC OS 2.

Example

```
*WimpWriteDir 0
```

Related commands

None

Related SWIs

None

Related vectors

None

54 Pinboard

Introduction and overview

The Pinboard module provides facilities for representing files, applications and directories outside the Filer, by positioning icons either on the icon bar or on the desktop background (the ‘pinboard’ that gives this module its title).

It also provides a * Command to change the desktop background from the default grey to any sprite of your own choice.

The Pinboard module is not available in RISC OS 2.

* Commands

*AddTinyDir

Adds a file, application or directory icon to the icon bar

Syntax

```
*AddTinyDir [object]
```

Parameters

object	a valid pathname specifying a file, application or directory

Use

*AddTinyDir adds a file, application or directory to the icon bar. If no pathname is given, it adds a blank directory icon to the icon bar. You can then later install a file, application or directory on the icon bar by dragging it to the blank icon.

Example

```
*AddTinyDir adfs::MHardy.$.!System
```

Related commands

*Pin, *RemoveTinyDir

Related SWIs

None

Related vectors

None

*BackDrop

Puts a sprite on the desktop background

Syntax

```
*BackDrop [-Centre|-Scale|-Tile] [filename]
```

Parameters

`-centre`	centre sprite on background
`-tile`	tile sprite over background
`-scale`	scale sprite to fill background (the default)
`filename`	a valid pathname, specifying a sprite file

Use

*BackDrop puts the first sprite in the given sprite file on the desktop background. The sprite is scaled to fill the background unless you specify otherwise.

If no filename is specified, the current backdrop's placing is altered.

Example

```
*BackDrop adfs::Disc4.$.Sprites.desert
```

Related commands

None

Related SWIs

None

Related vectors

None

*Pin

Adds a file, application or directory to the desktop pinboard

Syntax

```
*Pin object x y
```

Parameters

object	a valid pathname specifying a file, application or directory
x	the x-coordinate at which to pin the object's icon, given in OS units
y	the y-coordinate at which to pin the object's icon, given in OS units

Use

*Pin adds a file, application or directory to the desktop pinboard, positioning its icon at the given coordinates. The coordinates specify the top-left corner of the icon's bounding box (ie the box drawn around the icon when it is selected for a drag), not of the icon itself. To use a negative coordinate you need to specify it as `0-x` or `0-y`, to avoid the `-` sign being interpreted as the start of a flag. (You may sometimes see this when Pinboard saves its state to a desktop boot file.)

There is no equivalent command to remove the icon; to do so, you must choose **Remove icon** from the Pinboard menu.

Example

```
*Pin adfs::MHardy.$.!System 200 200
```

Related commands

*AddTinyDir

Related SWIs

None

Related vectors

None

*Pinboard

Starts the pinboard

Syntax

```
*Pinboard [-Grid]
```

Parameters

`-Grid`	Turn on grid locking (off by default)

Use

*Pinboard initialises the pinboard, removing any existing pinned icons and backdrop. Grid locking is off by default, but you may turn it on by passing the `-Grid` option to this command, or by choosing **Grid lock** from the Pinboard menu.

Related commands

None

Related SWIs

None

Related vectors

None

*RemoveTinyDir

Removes a file, application or directory icon from the icon bar

Syntax

```
*RemoveTinyDir [object]
```

Parameters

object	a valid pathname specifying a file, application or directory

Use

*RemoveTinyDir removes a file, application or directory icon that was previously placed on the icon bar by a *AddTinyDir command. If no pathname is given, all such icons are removed from the icon bar.

Example

```
*RemoveTinyDir adfs::MHardy.$.!System
```

Related commands

*AddTinyDir, *Pin

Related SWIs

None

Related vectors

None

55 Drag A Sprite

Introduction

The DragASprite module provides SWI calls with which you can make the pointer drag a sprite around the screen. Since not all users will prefer this effect to dragging an outline – whether for aesthetics or performance – there is a bit in the CMOS RAM used to indicate their preference. (See the section entitled *Non-volatile memory (CMOS RAM)* on page 1-361.) You should examine that bit before using this module; if it shows that the user would prefer to drag outlines, oblige them!

To drag a sprite:

1 Prepare a sprite to be dragged (this may be trivial, as the application may have a suitable sprite already to hand).

2 Call the SWI DragASprite_Start (see page 3-298). This takes a copy of your sprite – so you can dispose of your copy whenever you like – and then starts a Wimp drag.

3 When the Wimp sends you an indication that your drag has finished, you should call the SWI DragASprite_Stop (see page 3-300) to release the workspace used for the drag.

SWI calls

DragASprite_Start (SWI &42400)

On entry

R0 = flags
R1 = sprite area holding sprite:

0	system sprite area
1	wimp sprite area
Other	address of sprite area

R2 = pointer to sprite name
R3 = pointer to 16-byte block containing box
R4 = pointer to optional 16-byte block containing bounding box (see flags)

On exit

R0 - 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 starts dragging a sprite. The sprite you supply is copied, so there is no problem if you dispose of your copy of the sprite. If there is insufficient memory available to start the drag, the call reverts to a normal drag of a dotted outline.

The flags given in R0 have the following meanings:

Bits	Meaning
0 - 1	Horizontal location of sprite in box: 00 left 01 centre 10 right
2 - 3	Vertical location of sprite in box: 00 bottom 01 centre 10 top
4 - 5	Drag bounding box is: 00 whole screen 01 display area of window that the pointer's over 10 specified in block pointed to by R4
6	Bounding box applies to: 0 the box 1 the pointer
7	Control of drop-shadow: 0 don't do a drop-shadow 1 make a drop shadow when copying the sprite
8 - 31	Reserved for future use – should be set to 0

The blocks pointed to by R3 and – optionally – R4 have the following format:

Offset	Use	
0	x-low	box
4	y-low	bottom-left (x-low, y-low) is inclusive
8	x-high	top-right (x-high, y-high) is exclusive
12	y-high	

Related SWIs

DragASprite_Stop (page 3-300)

Related vectors

None

DragASprite_Stop (SWI &42401)

Terminates any current drag operation, and releases workspace

On entry

—

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call terminates any current drag operation, and releases any workspace claimed by the DragASprite module to do a drag. You should make this call when your application receives the User_Drag_Box reason code from Wimp_Poll (see page 3-112) during a drag.

Related SWIs

DragASprite_Start (page 3-298)

Related vectors

None

56 The Filter Manager

Introduction and Overview

The Filter Manager provides facilities for you to register filters to be used when a specified task calls Wimp_Poll, or when Wimp_Poll is about to return to that task. These are known – respectively – as pre-filters and post-filters:

- With a pre-filter, you may alter the event mask the task passes to Wimp_Poll
- With a post-filter, you may modify the reason code and data block returned by Wimp_Poll to provide a new event to the task, or to prevent an event form being returned to the task.

Filters need not be applied to a specific task; you can also apply filters to all tasks.

Each filter is a routine that has well-defined entry and exit conditions; it is your responsibility to write the routine.

Service Calls

Service_FilterManagerInstalled (Service Call &87)

Filter Manager starting up

On entry

R1 = &87 (reason code)

On exit

All registers preserved

Use

This service call is issued when the Filter Manager starts up. You may then register new filters using Filter_RegisterPreFilter (page 3-304) and Filter_RegisterPostFilter (page 3-306).

Service_FilterManagerDying (Service Call &88)

Filter Manager dying

On entry

R1 = &88 (reason code)

On exit

All registers preserved

Use

This service call is issued as a broadcast to inform filters that they have been deregistered and that the Filter Manager is about to die.

SWI calls

Filter_RegisterPreFilter (SWI &42640)

Adds a new pre-filter to the list of pre-filters

On entry

R0 = pointer to filter name (null terminated)
R1 = pointer to filter routine
R2 = value to be passed in R12 when filter is called
R3 = task handle to which to apply filter (or 0 for all tasks)

On exit

All registers preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call registers a pre-filter routine (pointed to by R1), which will be called whenever the specified task calls Wimp_Poll.

The entry and exit conditions of the filter routine are:

On entry

R0 = event mask, as passed to Wimp_Poll
R1 = pointer to event block, as passed to Wimp_Poll
R2 = task handle of task that called Wimp_Poll
R12 = value of R2 on entry to this SWI (ie Filter_RegisterPreFilter)

On Exit

It may clear bits in R0 to provide a new event mask

It must preserve all registers other than R0.

The routine should exit using the instruction:

```
MOVS    PC,R14
```

Related SWIs

Filter_RegisterPostFilter (page 3-306), Filter_DeRegisterPreFilter (page 3-308)

Related vectors

None

Filter_RegisterPostFilter (SWI &42641)

Adds a new post-filter to the list of post-filters

On entry

R0 = pointer to filter name (null terminated)
R1 = pointer to filter routine
R2 = value to be passed in R12 when filter is called
R3 = task handle to which to apply filter (or 0 for all tasks)
R4 = event mask (1 bit masks the event out as for Wimp_Poll)

On exit

All registers preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call registers a post-filter routine (pointed to by R1), which will be called whenever the Wimp is about to return from Wimp_Poll to the specified task.

The entry and exit conditions of the filter routine are:

On entry

R0 = event reason code, as returned from Wimp Poll
R1 = pointer to event block, as returned from Wimp Poll
R2 = task handle of task that is being returned to
R12 = value of R2 on entry to this SWI (ie Filter_RegisterPostFilter)

Task is paged in, so you can access its memory

On Exit

The routine may modify the reason code in R0 and the contents of the buffer pointed to by R1 to provide a new event. By setting R0 to –1 on exit it may claim the event, and prevent it from being passed to the task.

It must preserve all registers other than R0.

The routine should exit using the instruction:

```
MOVS    PC,R14
```

Related SWIs

Filter_RegisterPreFilter (page 3-304), Filter_DeRegisterPostFilter (page 3-309)

Related vectors

None

Filter_DeRegisterPreFilter (SWI &42642)

Removes a pre-filter from the list of pre-filters

On Entry

R0 = pointer to filter name (null terminated)
R1 = pointer to filter routine
R2 = value to be passed in R12 when filter is called
R3 = task handle to which to apply filter (or 0 for all tasks)

On exit

All registers preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call removes a pre-filter from the list of pre-filters. All values on entry must be the same as those used to originally register the filter (ie those that were passed to Filter_RegisterPreFilter).

Related SWIs

Filter_RegisterPreFilter (page 3-304)

Related vectors

None

Filter_DeRegisterPostFilter (SWI &42643)

Removes a post-filter from the list of post-filters

On entry

R0 = pointer to filter name (null terminated)
R1 = pointer to filter routine
R2 = value to be passed in R12 when filter is called
R3 = task handle to which to apply filter (or 0 for all tasks)
R4 = event mask (1 bit masks the event out as for Wimp_Poll)

On exit

All registers preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call removes a post-filter from the list of post-filters. All values on entry must be the same as those used to originally register the filter (ie those that were passed to Filter_RegisterPostFilter).

Related SWIs

Filter_RegisterPostFilter (page 3-306)

Related vectors

None

* Commands

*Filters

Lists all currently active pre- and post-Wimp_Poll filters

Syntax

```
*Filters
```

Parameters

None

Use

*Filters lists all currently active pre- and post-Wimp_Poll filters.

Example

```
*Filters

Filters called on entry to Wimp_Poll:
Filter          Task

Penguin         All tasks


Filters called on exit from Wimp_Poll:
Filter          Task            Mask

Penguin         All tasks       00000000
```

Related commands

None

Related SWIs

Filter_RegisterPreFilter (page 3-304), Filter_RegisterPostFilter (page 3-306)

Related vectors

None

57 The TaskManager module

Introduction and Overview

The Task Manager module provides various facilities to ease the management of tasks. These are:

- a SWI to find the name of a task, given its handle
- a SWI to enumerate all the currently active tasks
- a SWI to initiate a desktop shutdown
- a * Command to change the size of various system areas.

The Task Manager module is not available in RISC OS 2.

SWI calls

TaskManager_TaskNameFromHandle (SWI &42680)

Finds the name of a task

On entry

R0 = task handle

On exit

R0 = pointer to task name

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call returns the name of a task, given its task handle. If you wish to keep the name, you must copy it into your own workspace.

Related SWIs

TaskManager_EnumerateTasks (page 3-313)

Related vectors

None

TaskManager_EnumerateTasks (SWI &42681)

Enumerates all the currently active tasks

On entry

R0 = 0 for first call, or value from previous call
R1 = pointer to word aligned buffer
R2 = buffer length (in bytes)

On exit

R0 = value to pass to next call, or < 0 if no more entries
R1 = pointer to first unused word in buffer
R2 = number of unused bytes 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 enumerates all the currently active tasks. On exit the buffer is filled with entries of the form:

Byte	Meaning
0	task handle
4	pointer to task name (should be copied away and not used in place)
8	amount of memory (in K) used by the task
12	flags: Bit 0 1 ⇒ module task, 0 ⇒ application task Bit 1 1 ⇒ slot bar can be dragged, 0 ⇒ slot bar cannot be dragged (Bits 2-31 are reserved, and are currently 0)

Related SWIs

TaskManager_TaskNameFromHandle (page 3-312)

Related vectors

None

TaskManager_Shutdown (SWI &42682)

Initiates a desktop shutdown

On entry

R0 = shutdown 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 initiates a desktop shutdown. The actions performed are controlled by the shutdown flags held in R0:

Bit	Meaning when set
0	don't display restart dialogue (equivalent to **Exit** menu option)
1	don't broadcast Message_PreQuit (see page 3-228)
2	flag in CMOS as portable power-down
3	send a Message_Shutdown (see page 3-231)
4	reject OS_UpCall 1 and 2 (see page 1-183)
5 - 31	reserved (must be zero)

Related SWIs

None

Related vectors

None

* Commands

*ChangeDynamicArea

Changes the size of the font cache, system sprite area and/or RAM disc

Syntax

`*ChangeDynamicArea [-FontSize` *n*`[K]] [-SpriteSize` *n*`[K]] [-RamFsSize` *n*`[K]]`

Parameters

n	Size of the area to be set, in kilobytes

Use

*ChangeDynamicArea changes the size of the font cache, system sprite area and/or RAM disc. It generates an error if it is unable to do so. Its main use is in desktop boot files.

Example

```
*ChangeDynamicArea -SpriteSize 32K -RamFsSize 100K
```

Related commands

None

Related SWIs

OS_ChangeDynamicArea (page 1-384), OS_UpCall 257 (page 1-198)

Related vectors

None

58 TaskWindow

Introduction and Overview

The TaskWindow module is intended to allow programs which do not call SWI Wimp_Poll to be pre-emptively scheduled in the RISC OS desktop. In the following sections *Child* refers to the task created from a call to *TaskWindow and *Parent* refers to the task being used to display the Child's output.

Any screen output produced by the Child is intercepted and sent in Wimp messages to the Parent. These messages are documented on page 3-263.

SWI calls

TaskWindow_TaskInfo (SWI &43380)

Obtains information from the TaskWindow module

On entry

R0 = reason code

On exit

Registers' values depend on value of R0 on entry (see below)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call is used to obtain information from the TaskWindow module which is not readily available otherwise. The reason code in R0 on entry indicates which item of information is required. The registers on exit return the requested information.

Valid reason codes in R0 are:

On entry	On exit
0	R0 is non-zero if the calling task is running in a task window; otherwise it is zero

All other reason codes are reserved.

Related SWIs

Wimp_ReadSysInfo (page 3-216) with R0 = 3 on entry

Related vectors

None

* Commands

*ShellCLI_Task

Runs an application in a window

Syntax

`*ShellCLI_Task` *xxxxxxxx xxxxxxxx*

Parameters

xxxxxxxx	an 8 digit hex number giving the task handle of the parent task
xxxxxxxx	an 8 digit hex number giving a handle which may be used by the parent task to identify the task

Use

*ShellCLI_Task runs an application in a window. This command is intended for use only within desktop applications.

Use of this command is deprecated. Its functionality is subsumed within *TaskWindow.

Related commands

*ShellCLI_TaskQuit (page 3-323), *TaskWindow (page 3-324)

Related SWIs

None

Related vectors

None

*ShellCLI_TaskQuit

Quits the current task window

Syntax

```
*ShellCLI_TaskQuit
```

Parameters

None

Use

*ShellCLI_TaskQuit quits the current task window. This command is intended for use only within desktop applications.

Related commands

*ShellCLI_Task (page 3-322), *TaskWindow (page 3-324)

Related SWIs

None

Related vectors

None

*TaskWindow

Starts a background task, which will obtain a task window if necessary

Syntax

```
*TaskWindow [command] [[-wimpslot] nK] [[-name] taskname] [-ctrl]
            [-display] [-quit] [-task &xxxxxxxx] [-txt &xxxxxxxx]
```

Parameters

command	command to execute as a background task
n	size of memory to allocate to task
taskname	name of task
-ctrl	allow control characters through, depending on setting of **Ignore Ctrl** menu option
-display	open the task window immediately, rather than waiting for a character to be printed
-quit	make that task quit after the command, even if the task window has been opened
-task &*xxxxxxxx*	an 8 digit hex number giving the Wimp task-id of the calling task
-txt &*xxxxxxxx*	an 8 digit hex number giving the handle for the Parent to identify the Child by

Use

*TaskWindow starts a background task, which will obtain a task window if it needs to get input, or to output a character to the screen.

Any fields comprising more than one word must be enclosed in double quotes.

You must call *TaskWindow using *WimpTask (page 3-288) or the SWI Wimp_StartTask (page 3-174), rather than using the command line or the SWI OS_CLI. You can only call Wimp_StartTask or *WimpTask from within an active task.

If –txt and –task are not used, then before starting the task, a TaskWindow_NewTask message is broadcast to find an application (eg Edit) that can provide a window in which to show the task's output. An application task which receives this broadcast, and which wishes to receive output from the task, should acknowledge the message and then SWI Wimp_StartTask the command given in the message block.

Example

```
*TaskWindow "Cat Ram:$" -ctrl -display -quit
```

Related commands

None

Related SWIs

None

Related vectors

None

59 ShellCLI

Introduction

This module provides a single * Command that allows you to invoke a command shell from a Wimp program.

It also has two SWIs for its own internal use. **You must not use them** in your own code.

SWI Calls

Shell_Create (SWI &405C0)

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

Shell_Destroy (SWI &405C1

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

* Commands

*ShellCLI

Invokes a command shell from a Wimp program

Syntax

```
*ShellCLI
```

Parameters

None

Use

*ShellCLI invokes a command shell from a Wimp program, starting it as a Wimp task. It prompts the user with *, and passes each line that the user types to the command line interpreter, OS_CLI (page 1-961). This is repeated until the user enters a blank line, whereupon control is returned to the Wimp program. The Task Manager uses this command to implement its *Command (F12) menu item.

You must call *ShellCLI using *WimpTask (page 3-288) or the SWI Wimp_StartTask (page 3-174), rather than using the command line or the SWI OS_CLI. You can only call Wimp_StartTask or *WimpTask from within an active task.

The command uses the two SWIs Shell_Create and Shell_Destroy; it is the only user of these SWIs.

Example

```
*WimpTask ShellCLI
```

Related commands

None

Related SWIs

Shell_Create (page 3-328), Shell_Destroy (page 3-329)

Related vectors

None

Part 8 – Non-kernel input/output

60 ColourTrans

Introduction

ColourTrans allows a program to select the physical red, green and blue colours that it wishes to use, given a particular output device and palette. ColourTrans then calculates the best colour available to fit the required colour.

Thus, an application doesn't have to be aware of the number of colours available in a given mode.

It can also intelligently handle colour usage with sprites and the font manager, and is the best way to set up colours when printing.

Finally, it supports colour calibration, so that you can make different output devices produce the same colours. (This feature is not supported by RISC OS 2)

Before reading this chapter, you should be familiar with the VDU, sprite and font manager principles.

We also advise that you read the section entitled *Printing a document from an application* on page 3-568. This section gives advice on which ColourTrans calls you should use to set colours when printing. You'll probably find it easiest if you use the same calls for screen output; you should then find that your routines for printer and screen output can share large parts of coding.

Overview

The ColourTrans module is provided on disc in RISC OS 2 as the file System:Modules.Colours, but is in the ROM for later releases of RISC OS. Any application which uses it should ensure it is present using the *RMEnsure command, say from an Obey file. For example:

```
RMEnsure ColourTrans 0.51 RMLoad System:Modules.Colours
RMEnsure ColourTrans 0.51 Error You need ColourTrans 0.51 or later
```

Definition of terms

Here are some terms you should know when using this chapter.

GCOL is like the colour parameter passed to VDU 17. It uses a simple format for 256 colour modes.

Colour number is what is written into screen memory to achieve a given colour in a particular mode.

Palette entry is a word that contains a description of a physical colour in red, green and blue levels. Usually, this term refers to the required colour that is passed to a ColourTrans SWI.

Palette pointer is a pointer to a list of palette entries. The table would have one entry for each logical colour in the requested mode. In 256 colour mode, only 16 entries are needed, as there are only 16 palette registers.

Closest colour is the colour in the palette that most closely matches the palette entry passed. Furthest colour is the one furthest from the colour requested. These terms refer to a least-squares test of closeness.

Finding a colour

There are many SWIs that will find the best fit colour in the palette for a set of parameters. Here is a list of the different kinds of parameters that can return a best fit colour:

- Given palette entry, return nearest or furthest GCOL
- Given palette entry, return nearest or furthest colour number
- Given palette entry, mode and palette pointer, return nearest or furthest GCOL
- Given palette entry, mode and palette pointer, return nearest or furthest colour number

Setting a colour

Some SWIs will set the VDU driver GCOL to the calculated GCOL after finding it.

- Given palette entry, return nearest GCOL, and set that colour
- Given palette entry, return furthest GCOL, and set that colour

Conversion

There is a pair of SWIs to convert GCOLs to and from colour numbers. Note that this only has meaning for 256 colour modes. There are also SWIs to convert between different colour models, such as RGB, CIE, HSV, and CMYK.

Sprites and Fonts

ColourTrans provides full facilities for setting the colours used by sprites and fonts.

Using other palette SWIs

If an application changes the output palette (perhaps by changing the screen colours or by switching output to a sprite), then it has to call a SWI to inform ColourTrans. This is because ColourTrans maintains a cache used for mapping colours. If the palette has independently changed, then it has no way of telling.

If the screen mode has changed there is no need to use this call, since the ColourTrans module detects this itself – but, under RISC OS 2, if output is switched to a sprite (and ColourTrans will be used) then the SWI must also be called.

Wimp

If you are using the Wimp interface, then the ColourTrans calls are fine to use, because they never modify the palette.

Printing

Because ColourTrans allows an application to request an RGB colour rather than a logical colour, it is ideal for use with the printer drivers, where a printer may be able to represent some RGB colours more accurately then the screen.

Colour calibration

There is a major problem in working with colour documents. This is that, if the user selects some colours on the screen, they may well come out as different colours on a printer or other final output device. Colour calibration is a way to get round this problem.

Colour calibration involves calibrating the screen colours with a fixed standard set of colours, and also calibrating the output device colours to the same fixed set of colours. Then, when an application draws to the screen, it does so in standard colours which are converted by the OS to screen colours. If the application draws to the printer it again does so in standard colours, but this time they are converted to printer colours.

So, for the user, calibrating the colours will give constant colour reproduction throughout the system, for the cost of calibrating the devices in the first place.

Colour calibration is not available in RISC OS 2.

Technical Details

Colours

Two different colour systems are used in 256 colour modes. The GCOL form is much easier to use, while the colour number is optimised for the hardware. In all other colour modes, they are identical.

The palette entry used to request a given physical colour is in the same format as that used to set the anti-alias palette in the font manager.

GCOL

The 256 colour modes use a byte that looks like this:

Bit	Meaning
0	Tint bit 0 (red+green+blue bit 0)
1	Tint bit 1 (red+green+blue bit 1)
2	Red bit 2
3	Red bit 3 (high)
4	Green bit 2
5	Green bit 3 (high)
6	Blue bit 2
7	Blue bit 3 (high)

This format is converted into the internal ‘colour number’ format when stored, because that is what the VIDC hardware recognises.

Colour number

The 256 colour mode in the colour number looks like this:

Bit	Meaning
0	Tint bit 0 (red+green+blue bit 0)
1	Tint bit 1 (red+green+blue bit 1)
2	Red bit 2
3	Blue bit 2
4	Red bit 3 (high)
5	Green bit 2
6	Green bit 3 (high)
7	Blue bit 3 (high)

In fact the bottom 4 bits of the colour number are obtained via the palette, but the default palette in 256 colour modes is set up so that the above settings apply, and this is not normally altered.

Palette entry

The palette entry is a word of the form &BBGGRR00. That is, it consists of four bytes, with the palette value for the blue, green and red gun in the top three bytes. Bright white, for instance would be &FFFFFF00, while half intensity cyan would be &77770000. The current graphics hardware only uses the upper nibbles of these colours, but for upwards compatibility the lower nibble should contain a copy of the upper nibble.

Finding a colour

The SWIs that find the best fit have generally self explanatory names. As shown in the overview, they follow a standard pattern. They are as follows:

ColourTrans_ReturnGCOL (page 3-348)
Given palette entry, return nearest GCOL

ColourTrans_ReturnOppGCOL (page 3-357)
Given palette entry, return furthest GCOL

ColourTrans_ReturnColourNumber (page 3-352)
Given palette entry, return nearest colour number

ColourTrans_ReturnOppColourNumber (page 3-361)
Given palette entry, return furthest colour number

ColourTrans_ReturnGCOLForMode (page 3-353)
Given palette entry, mode and palette pointer, return nearest GCOL

ColourTrans_ReturnOppGCOLForMode (page 3-362)
Given palette entry, mode and palette pointer, return furthest GCOL

ColourTrans_ReturnColourNumberForMode (page 3-355)
Given palette entry, mode and palette pointer, return nearest colour number

ColourTrans_ReturnOppColourNumberForMode (page 3-364)
Given palette entry, mode and palette pointer, return furthest colour number

Palette pointers

Where a palette pointer is used, certain conventions apply:

- a palette pointer of –1 means the current palette is used
- a palette pointer of 0 means the default palette for the specified mode.

Modes

Similarly, where modes are used:

- mode –1 means the current mode.

Best fit colour

These calls use a simple algorithm to find the colour in the palette that most closely matches the high resolution colour specified in the palette entry. It calculates the *distance* between the colours, which is a weighted least squares function. If the desired colour is (R_d, B_d, G_d) and a trial colour is (R_t, B_t, G_t), then:

distance = redweight $\times (R_t - R_d)^2$ + greenweight $\times (G_t - G_d)^2$ + blueweight $\times (B_t - B_d)^2$

where redweight = 2, greenweight = 4 and blueweight = 1. These weights are set for the most visually effective solution to this problem. (In RISC OS 2, the weights used were 2, 3 and 1 respectively.)

Setting a colour

ColourTrans_SetGCOL (page 3-350) will act like ColourTrans_ ReturnGCOL, except that it will set the graphics system GCOL to be as close to the colour you requested as it can. Note that ECF patterns will not yet be used in monochrome modes to reflect grey shades, as they are with Wimp_SetColour.

Similarly, ColourTrans_SetOppGCOL (page 3-359) will set the graphics system GCOL with the opposite of the palette entry passed.

Conversion

To convert between the GCOL and colour number format in 256 colour modes, the SWIs ColourTrans_GCOLToColourNumber (page 3-366) and ColourTrans_ColourNumberToGCOL (page 3-367) can be used.

Sprites and Fonts

ColourTrans_SelectTable (page 3-344) will set up a translation table in the buffer. ColourTrans_SelectGCOLTable (page 3-346) will set up a list of GCOLs in the buffer. See the section entitled *Pixel translation table* on page 1-780 for a definition of these tables (although the latter call does not in fact relate to sprites).

ColourTrans_ReturnFontColours (page 3-368) will try and find the best set of logical colours for an anti-alias colour range. ColourTrans_SetFontColours (page 3-370) also does this, but sets the font manager plotting colours as well. It calls Font_SetFontColours, or Font_SetPalette in 256 colour modes – but it works out which logical colours to use beforehand. See the section entitled *Colours* on page 3-413 for details of using colours and anti-aliasing colours; see also the descriptions of the relevant commands later in the same chapter, on page 3-461 and page 3-463.

Using other palette SWIs

If a program has changed the palette, then ColourTrans_InvalidateCache (page 3-372) must be called. This will reset its internal cache. This applies to Font_SetFontColours or Wimp_SetPalette or VDU 19 or anything like that, but not to mode change, since this is detected automatically.

Under RISC OS 2 you must also call this SWI if output has been switched to a sprite, and ColourTrans is to be called while the output is so redirected. You must then call it again after output is directed back to the screen. Later versions of RISC OS automatically do this for you.

Colour calibration

Colour calibration is performed by ColourTrans using a calibration table that maps from device colours to standard colours.

The palette in RISC OS maps logical colours to device colours (also known as physical colours). When you ask RISC OS to select a colour for you, it takes this palette and uses a calibration table to convert the device colours to standard colours, giving a (transient) palette that maps logical colours to standard colours. It then chooses the closest standard colour to the one that you have specified.

Calibration tables

A calibration table is a one-to-one map that fills the device colour space, but does not necessarily fill the standard colour space. In fact, it consists of three separate mappings: one for each component of the device space (red, green and blue on a monitor, for example). Each mapping consists of a series of device component/ standard colour pairs.

The pairs are stored as 32-bit words, in the form &BBGGRRDD, where DD is the amount of the device component (from 0 to 255), and BBGGRR is the standard colour corresponding to that amount. The two other device components are presumed to be zero.

The format of the table is:

Word	Meaning
0	Number of pairs of component 1 (*n1*)
1	Number of pairs of component 2 (*n*2)
2	Number of pairs of component 3 (*n3*)
3	*n1* words giving pairs for component 1
3 + *n1*	*n2* words giving pairs for component 2
3 + *n1* + *n2*	*n3* words giving pairs for component 3

The length of the table is therefore 3 + *n1* + *n2* + *n3* words.

Within each of the three sets of mappings, the words must be sorted in ascending order of device component. To fill the device colour space, there must be entries for device components of 0 and 255, so there must be at least two pairs for each component.

As an example, a minimal calibration table might be:

Word	Meaning
&00000002	2 pairs of red component
&00000002	2 pairs of green component
&00000002	2 pairs of blue component
&02010300	Device colour 000000 corresponds to standard colour 020103
&0203FDFF	Device colour 0000FF corresponds to standard colour 0203FD
&02010300	Device colour 000000 corresponds to standard colour 020103
&03FC02FF	Device colour 00FF00 corresponds to standard colour 03FC02
&02010300	Device colour 000000 corresponds to standard colour 020103
&FF0302FF	Device colour FF0000 corresponds to standard colour FF0302
	(In this column both device and standard colours are given in the format &BBGGRR)

The default mapping for the screen is that device colours and standard colours are the same. This produces the same effect as earlier uncalibrated versions of ColourTrans.

To convert a specific device colour to a standard colour, ColourTrans splits the device colour into its three component parts. Then, for each component, it uses linear interpolation between the two device components ‘surrounding’ the required device component. The standard colours thus obtained for each component are then summed to give the final calibrated standard colour.

Colour calibration is not available in RISC OS 2.

Service Calls

Service_CalibrationChanged (Service Call &5B)

Screen calibration is changed

On entry

R1 = &5B (reason code)

On exit

All registers preserved

This service call should not be claimed

Use

This service is issued by the ColourTrans module when the ColourTrans_SetCalibration SWI has been issued.

It is noticed by the Palette utility in the desktop, which broadcasts a Message_PaletteChange.

This service call is not used by RISC OS 2.

Service_InvalidateCache (Service Call &82)

Broadcast whenever the cache is flushed within ColourTrans

On entry

R1 = &82 (reason code)

On exit

All registers preserved

Use

This service is broadcast whenever the cache is flushed within ColourTrans. You should never claim it.

This service call is not used by RISC OS 2.

SWI Calls

ColourTrans_SelectTable (SWI &40740)

Sets up a translation table in a buffer

On entry

R0 = source mode, or –1 for current mode, or (if ≥ 256) pointer to sprite area
R1 = source palette pointer, or –1 for current palette, or (if R0 ≥ 256) pointer to sprite name/sprite in area pointed to by R0 (as specified by bit 0 of R5)
R2 = destination mode, or –1 for current mode
R3 = destination palette pointer, or –1 for current palette, or 0 for default for the mode
R4 = pointer to buffer, or 0 to return required size of buffer
R5 = flags (used if R0 ≥ 256):
- bit 0 set ⇒ R1 = pointer to sprite; else R1 = pointer to sprite name
- bit 1 set ⇒ use current palette if sprite doesn't have one; else use default
- bit 2 set ⇒ use R6 and R7 to specify transfer function
- bits 24 - 31 give format of table:
  - 0 ⇒ return pixel translation table (see page 1-780)
  - 1 ⇒ return physical palette table
- all other bits reserved (must be zero)

R6 = pointer to workspace for transfer function (if R0 ≥ 256, and bit 2 of R5 is set)
R7 = pointer to transfer function (if R0 ≥ 256, and bit 2 of R5 is set)

On exit

R0 - R3 preserved
R4 = required size of buffer (if R4 = 0 on entry), or preserved
R5 - 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

This call sets up a translation table in a buffer – that is, a set of colour numbers as used by scaled sprite plotting. You may specify the source mode palette either directly, or (except in RISC OS 2) by specifying a sprite. See the section entitled *Pixel translation table* on page 1-780 for details of such tables.

You should use this call rather than any other to set up translation tables for sprites, as it copes correctly with sprites that have a 256 colour palette.

If bit 2 of the flags word in R5 is set, then R6 and R7 are assumed to specify a transfer routine, which is called to preprocess each palette entry before it is converted. The entry point of the routine (as specified in R7) is called with the palette entry in R0, and the workspace pointer (as specified in R6) in R12. The palette entry must be returned in R0, and all other registers preserved.

In RISC OS 2, R0 must be less than 256, and so R5 - R7 are unused. Consequently, to use a sprite as the source you first have to copy its palette information out from its header. Furthermore, you cannot find the required size of the buffer by setting R4 to 0 on entry.

Related SWIs

ColourTrans_GenerateTable (page 3-405)

Related vectors

ColourV

ColourTrans_SelectGCOLTable (SWI &40741)

Sets up a list of GCOLs in a buffer

On entry

R0 = source mode, or –1 for current mode, or (if ≥ 256) pointer to sprite area
R1 = source palette pointer, or –1 for current palette, or (if R0 ≥ 256) pointer to sprite name/sprite in area pointed to by R0 (as specified by bit 0 of R5)
R2 = destination mode, or –1 for current mode
R3 = destination palette pointer, or –1 for current palette, or 0 for default for the mode
R4 = pointer to buffer
R5 = flags (used if R0 ≥ 256):
bit 0 set ⇒ R1 = pointer to sprite; else R1 = pointer to sprite name
bit 1 set ⇒ use current palette if sprite doesn't have one; else use default
all other bits reserved (must be zero)

On exit

R0 - R5 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrant

SWI is not re-entrant

Use

This call, given a source mode and palette (either directly, or – except in RISC OS 2 – from a sprite), a destination mode and palette, and a buffer, sets up a list of GCOLs in the buffer. The values can subsequently be used by passing them to GCOL and Tint.

In RISC OS 2, R0 must be less than 256, and so R5 is unused. Consequently, to use a sprite as the source you first have to copy its palette information out from its header.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ReturnGCOL (SWI &40742)

Gets the closest GCOL for a palette entry

On entry

R0 = palette entry

On exit

R0 = GCOL

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, returns the closest GCOL in the current mode and palette.

It is equivalent to ColourTrans_ReturnGCOLForMode for the given palette entry, with parameters of –1 for both the mode and palette pointer.

The colours are not calibrated in RISC OS 2, but are calibrated in later versions.

Related SWIs

ColourTrans_SetGCOL (page 3-350),
ColourTrans_ReturnColourNumber (page 3-352),
ColourTrans_ReturnGCOLForMode (page 3-353),
ColourTrans_ReturnOppGCOL (page 3-357)

Related vectors

ColourV

ColourTrans_SetGCOL (SWI &40743)

Sets the closest GCOL for a palette entry

On entry

R0 = palette entry
R3 = flags
R4 = GCOL action

On exit

R0 = GCOL
R2 = $\log_2$ of bits-per-pixel for current mode
R3 = initial value AND &80
R4 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, works out the closest GCOL in the current mode and palette, and sets it. The flags in R3 have the following meaning:

Value of R3	Meaning
bit 7 = 1	set background colour
bit 7 = 0	set foreground colour
bit 8 = 1	use ECFs to give a better approximation to the colour
bit 8 = 0	don’t use ECFs

The remaining bits of R3 and the top three bytes of R4 are reserved, and should be set to zero to allow for future expansion. Bit 8 of R3 is ignored in RISC OS 2, which does not support ECF patterns with this call.

Note that if you are using ECF-generating calls, you cannot use the returned GCOL to reselect the pattern; you must instead repeat this call.

The colours are not calibrated in RISC OS 2, but are calibrated in later versions.

Related SWIs

ColourTrans_ReturnGCOL (page 3-348), ColourTrans_SetOppGCOL (page 3-359)

Related vectors

ColourV

ColourTrans_ReturnColourNumber (SWI &40744)

Gets the closest colour for a palette entry

On entry

R0 = palette entry

On exit

R0 = colour number

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, returns the closest colour number in the current mode and palette.

The colours are not calibrated in RISC OS 2, but are calibrated in later versions.

Related SWIs

ColourTrans_ReturnGCOL (page 3-348),
ColourTrans_ReturnColourNumberForMode (page 3-355),
ColourTrans_ReturnOppColourNumber (page 3-361)

Related vectors

ColourV

ColourTrans_ReturnGCOLForMode (SWI &40745)

Gets the closest GCOL for a palette entry

On entry

R0 = palette entry
R1 = destination mode, or –1 for current mode
R2 = palette pointer, or –1 for current palette, or 0 for default for the mode

On exit

R0 = GCOL
R1 = preserved
R2 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, a destination mode and palette, returns the closest GCOL.

The colours are not calibrated in RISC OS 2, but are calibrated in later versions.

Related SWIs

ColourTrans_ReturnGCOL (page 3-348), ColourTrans_SetGCOL (page 3-350), ColourTrans_ReturnColourNumberForMode (page 3-355), ColourTrans_ReturnOppGCOLForMode (page 3-362)

Related vectors

ColourV

ColourTrans_ReturnColourNumberForMode (SWI &40746)

Gets the closest colour for a palette entry

On entry

R0 = palette entry
R1 = destination mode, or –1 for current mode
R2 = palette pointer, or –1 for current palette, or 0 for default for the mode

On exit

R0 = colour number
R1 = preserved
R2 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, a destination mode and palette, returns the closest colour number.

The colours are not calibrated in RISC OS 2, but are calibrated in later versions.

Related SWIs

ColourTrans_ReturnColourNumber (page 3-352),
ColourTrans_ReturnGCOLForMode (page 3-353),
ColourTrans_ReturnOppColourNumberForMode (page 3-364)

Related vectors

ColourV

ColourTrans_ReturnOppGCOL (SWI &40747)

Gets the furthest GCOL for a palette entry

On entry

R0 = palette entry

On exit

R0 = GCOL

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, returns the furthest GCOL in the current mode and palette.

It is equivalent to ColourTrans_ReturnOppGCOLForMode for the given palette entry, with parameters of –1 for both the mode and palette pointer.

The colours are not calibrated in RISC OS 2, but are calibrated in later versions.

Related SWIs

ColourTrans_ReturnGCOL (page 3-348), ColourTrans_SetOppGCOL (page 3-359), ColourTrans_ReturnOppColourNumber (page 3-361), ColourTrans_ReturnOppGCOLForMode (page 3-362)

Related vectors

ColourV

ColourTrans_SetOppGCOL (SWI &40748)

Sets the furthest GCOL for a palette entry

On entry

R0 = palette entry
R3 = 0 for foreground, or 128 for background
R4 = GCOL action

On exit

R0 = GCOL
R2 = $\log_2$ of bits-per-pixel for current mode
R3 = initial value AND &80
R4 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, works out the furthest GCOL in the current mode and palette, and sets it.

The top three bytes of R3 and R4 should be zero, to allow for future expansion.

The colours are not calibrated in RISC OS 2, but are calibrated in later versions.

Related SWIs

ColourTrans_SetGCOL (page 3-350), ColourTrans_ReturnOppGCOL (page 3-357)

Related vectors

ColourV

ColourTrans_ReturnOppColourNumber (SWI &40749)

Gets the furthest colour for a palette entry

On entry

R0 = palette entry

On exit

R0 = colour number

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, returns the furthest colour number in the current mode and palette.

The colours are not calibrated in RISC OS 2, but are calibrated in later versions.

Related SWIs

ColourTrans_ReturnColourNumber (page 3-352),
ColourTrans_ReturnOppGCOL (page 3-357),
ColourTrans_ReturnOppColourNumberForMode (page 3-364)

Related vectors

ColourV

ColourTrans_ReturnOppGCOLForMode (SWI &4074A)

Gets the furthest GCOL for a palette entry

On entry

R0 = palette entry
R1 = destination mode or –1 for current mode
R2 = palette pointer, or –1 for current palette, or 0 for default for the mode

On exit

R0 = GCOL
R1 = preserved
R2 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, a destination mode and palette, returns the furthest GCOL.

The colours are not calibrated in RISC OS 2, but are calibrated in later versions.

Related SWIs

ColourTrans_ReturnGCOLForMode (page 3-353), ColourTrans_ReturnOppGCOL (page 3-357), ColourTrans_SetOppGCOL (page 3-359), ColourTrans_ReturnOppColourNumberFor Mode (page 3-364)

Related vectors

ColourV

ColourTrans_ReturnOppColourNumberForMode (SWI &4074B)

Gets the furthest colour for a palette entry

On entry

R0 = palette entry
R1 = destination mode or –1 for current mode
R2 = palette pointer, or –1 for current palette, or 0 for default for the mode

On exit

R0 = colour number
R1 = preserved
R2 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a palette entry, a destination mode and palette, returns the furthest colour number.

The colours are not calibrated in RISC OS 2, but are calibrated in later versions.

Related SWIs

ColourTrans_ReturnColourNumberForMode (page 3-355),
ColourTrans_ReturnOppColourNumber (page 3-361),
ColourTrans_ReturnOppGCOLForMode (page 3-362)

Related vectors

ColourV

ColourTrans_GCOLToColourNumber (SWI &4074C)

Translates a GCOL to a colour number

On entry

R0 = GCOL

On exit

R0 = colour number

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call changes the value passed from a GCOL to a colour number.

You should only call this SWI for 256 colour modes; the results will be meaningless for any others.

Related SWIs

ColourTrans_ColourNumberToGCOL (page 3-367)

Related vectors

ColourV

ColourTrans_ColourNumberToGCOL (SWI &4074D)

Translates a colour number to a GCOL

On entry

R0 = colour number

On exit

R0 = GCOL

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call changes the value passed from a colour number to a GCOL.

You should only call this SWI for 256 colour modes; the results will be meaningless for any others.

Related SWIs

ColourTrans_GCOLToColourNumber (page 3-366)

Related vectors

ColourV

ColourTrans_ReturnFontColours (SWI &4074E)

Finds the best range of anti-alias colours to match a pair of palette entries

On entry

R0 = font handle, or 0 for the current font
R1 = background palette entry
R2 = foreground palette entry
R3 = maximum foreground colour offset (0 - 14)

On exit

R0 = preserved
R1 = background logical colour (preserved if in 256 colour mode)
R2 = foreground logical colour
R3 = maximum sensible colour offset (up to R3 on entry)

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given background and foreground colours and the number of anti-aliasing colours desired, finds the maximum range of colours that can sensibly be used. So for the given pair of palette entries, it finds the best fit in the current palette, and then inspects the other available colours to deduce the maximum possible amount of anti-aliasing up to the limit in R3.

If anti-aliasing is desirable, you should set R3 = 14 on entry; otherwise set R3 = 0 for monochrome.

The values in R1 - R3 on exit are suitable for passing to Font_SetFontColours. You can also include them in a font string in a control (18) sequence, although we don't recommend this as the printer drivers do not properly support this feature.

Note that in 256 colour modes, you can only set 16 colours before previously returned information becomes invalid. Therefore, if you are using this SWI to obtain information to subsequently pass to the font manager, do not use more than 16 colours.

Also note that in 256 colour modes, the font manager's internal palette will be set, with all 16 entries being cycled through by ColourTrans.

The colours are not calibrated in RISC OS 2, but are calibrated in later versions.

See page 3-461 of the chapter entitled *The Font Manager* for further details of the parameters used in this call.

Related SWIs

ColourTrans_SetFontColours (page 3-370),
Font_SetFontColours (page 3-461)

Related vectors

ColourV

ColourTrans_SetFontColours (SWI &4074F)

Sets the best range of anti-alias colours to match a pair of palette entries

On entry

R0 = font handle, or 0 for the current font
R1 = background palette entry
R2 = foreground palette entry
R3 = maximum foreground colour offset (0 - 14)

On exit

R0 preserved
R1 = background logical colour (preserved if in 256 colour mode)
R2 = foreground logical colour
R3 = maximum sensible colour offset (up to R3 on entry)

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call, given a pair of palette entries, finds the best available range of anti-alias colours in the current palette, and sets the font manager to use these colours. It is the recommended way to set font colours, as the printer drivers properly support this call. A font string control (19) sequence uses this call, and so may also be used when printing.

The colours are not calibrated in RISC OS 2, but are calibrated in later versions.

Related SWIs

ColourTrans_ReturnFontColours (page 3-368)

Related vectors

ColourV

ColourTrans_InvalidateCache (SWI &40750)

Informs ColourTrans that the palette has been changed by some other means

On entry

—

On exit

—

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call must be issued whenever the palette has changed since ColourTrans was last called. This forces ColourTrans to update its cache. Note that colour changes due to a mode change are detected; you only need to use this if another of the palette change operations was used.

Under RISC OS 2 you must also call this SWI if output has been switched to a sprite, and ColourTrans is to be called while the output is so redirected. You must then call it again after output is directed back to the screen. For example, the palette utility on the icon bar calls this SWI when you finish dragging one of the RGB slider bars. Later versions of RISC OS automatically do this for you.

Related SWIs

None

Related vectors

ColourV

ColourTrans_SetCalibration (SWI &40751)

Sets the calibration table for the screen

On entry

R0 = pointer to calibration table

On exit

—

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call copies the calibration table pointed to by R0 into the RMA as the new calibration table for the screen. If the call fails due to lack of room in the RMA then the calibration will be set to the default calibration for the screen, and the ‘No room in RMA’ error will be passed back. Another possible error is ‘Bad calibration table’, given if the device component pairs do not cover the full range 00 to &FF.

This call is not available in RISC OS 2.

Related SWIs

ColourTrans_ReadCalibration (page 3-375)

Related vectors

ColourV

ColourTrans_ReadCalibration (SWI &40752)

Reads the calibration table for the screen

On entry

R0 = 0 to read required size of table, or pointer to buffer

On exit

R0 preserved
R1 = size of table (if R0 = 0 on entry)

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call reads the calibration table for the screen into the buffer pointed to by R0, which should be large enough to contain the complete table. Ideally you should first issue this call with R0=0 to read the size of the table, then allocate space, and then issue this call again to read the table.

This call is not available in RISC OS 2.

Related SWIs

ColourTrans_SetCalibration (page 3-374)

Related vectors

ColourV

ColourTrans_ConvertDeviceColour (SWI &40753)

Converts a device colour to a standard colour

On entry

R1 = 24-bit device colour (&BBGGRR00 for the screen)
R3 = 0 to use the current screen calibration, or pointer to calibration table to use

On exit

R2 = 24-bit standard colour (&BBGGRR00)

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call allows applications to read, say, screen colours, and find the standard colours to which they correspond.

This call is not available in RISC OS 2.

Related SWIs

ColourTrans_ConvertDevicePalette (page 3-377)

Related vectors

ColourV

ColourTrans_ConvertDevicePalette (SWI &40754)

Converts a device palette to standard colours

On entry

R0 = number of colours to convert
R1 = pointer to table of 24-bit device colours
R2 = pointer to table to store standard colours
R3 = 0 to use the current screen calibration, or pointer to calibration table to use

On exit

R0 - R3 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call allows printer drivers to use the same calibration calculation code for their conversions between device and standard colours as the screen does. The printer device palette can be set up and then converted using this call to the standard colours using the printer's calibration table. This call is mainly provided to ease the load on the writers of printer drivers.

This call is not available in RISC OS 2.

Related SWIs

ColourTrans_ConvertDeviceColour (page 3-376)

Related vectors

ColourV

ColourTrans_ConvertRGBToCIE (SWI &40755)

Converts RISC OS RGB colours to industry standard CIE colours

On entry

R0 = red component
R1 = green component
R2 = blue component

On exit

R0 = CIE X tristimulus value
R1 = CIE Y tristimulus value
R2 = CIE Z tristimulus value

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call converts RISC OS RGB colours to industry standard CIE colours, allowing easy interchange with other systems. The CIE standard that is output is the XYZ tristimulus values.

All parameters are passed as fixed point 32 bit numbers, with 16 bits below the point and 16 bits above the point. We suggest that you use numbers in the range 0 - 1, for compatibility with other conversion SWIs such as ColourTrans_ConvertRGBToCMYK.

This call is not available in RISC OS 2.

Related SWIs

ColourTrans_ConvertCIEToRGB (page 3-381)

Related vectors

ColourV

ColourTrans_ConvertCIEToRGB (SWI &40756)

Converts industry standard CIE colours to RISC OS RGB colours

On entry

R0 = CIE X tristimulus value
R1 = CIE Y tristimulus value
R2 = CIE Z tristimulus value

On exit

R0 = red component
R1 = green component
R2 = blue component

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call converts industry standard CIE colours to RISC OS RGB colours, allowing easy interchange with other systems. The CIE standard that is accepted is the XYZ tristimulus values.

All parameters are passed as fixed point 32 bit numbers, with 16 bits below the point and 16 bits above the point. We suggest that you use numbers in the range 0 - 1, for compatibility with other conversion SWIs such as ColourTrans_ConvertCMYKToRGB.

This call is not available in RISC OS 2.

Related SWIs

ColourTrans_ConvertRGBToCIE (page 3-379)

Related vectors

ColourV

ColourTrans_WriteCalibrationToFile (SWI &40757)

Saves the current calibration to a file

On entry

R0 = flags
R1 = file handle of file to save calibration to

On exit

R0 corrupted

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call saves the current calibration to a file. It does so by creating a list of
* Commands which will recreate the current calibration.

If bit 0 of R0 is clear then the calibration will only be saved if it is not the default calibration. If bit 0 of R0 is set then the calibration will be saved even if it is the default calibration.

This call is not available in RISC OS 2.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ConvertRGBToHSV (SWI &40758)

Converts RISC OS RGB colours into corresponding hue, saturation and value

On entry

R0 = red component
R1 = green component
R2 = blue component

On exit

R0 = hue
R1 = saturation
R2 = value

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call converts RISC OS RGB colours into corresponding hue, saturation and value.

All parameters are passed as fixed point 32 bit numbers, with 16 bits below the point and 16 bits above the point. Hue ranges from 0 - 360 with no fractional element, whilst the remaining parameters are in the range 0 - 1 and may have fractional elements.

When dealing with achromatic colours, hue is undefined.

This call is not available in RISC OS 2.

Related SWIs

ColourTrans_ConvertHSVToRGB (page 3-387)

Related vectors

ColourV

ColourTrans_ConvertHSVToRGB (SWI &40759)

Converts hue, saturation and value into corresponding RISC OS RGB colours

On entry

R0 = hue
R1 = saturation
R2 = value

On exit

R0 = red component
R1 = green component
R2 = blue component

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call converts hue, saturation and value into corresponding RISC OS RGB colours.

All parameters are passed as fixed point 32 bit numbers, with 16 bits below the point and 16 bits above the point. Hue ranges from 0 - 360 with no fractional element, whilst the remaining parameters are in the range 0 - 1 and may have fractional elements.

An error is generated if both the hue and saturation are 0; for this reason we recommend that when using this call $0 < \text{hue} \leq 360$.

This call is not available in RISC OS 2.

Related SWIs

ColourTrans_ConvertRGBToHSV (page 3-385)

Related vectors

ColourV

ColourTrans_ConvertRGBToCMYK (SWI &4075A)

Converts RISC OS RGB colours into the CMYK model

On entry

R0 = red component
R1 = green component
R2 = blue component

On exit

R0 = cyan component
R1 = magenta component
R2 = yellow component
R3 = key (black) component

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call converts RISC OS RGB colours into the CMY (cyan/magenta/yellow) model with a K (key – ie black) additive, allowing easy preparation of colour separations.

All parameters are passed as fixed point 32 bit numbers in the range 0 - 1, with 16 bits below the point and 16 bits above the point. The ‘K’ acts as a black additive and is a value equally subtracted or added to the given CMY values.

This call is not available in RISC OS 2.

Related SWIs

ColourTrans_ConvertCMYKToRGB (page 3-391)

Related vectors

ColourV

ColourTrans_ConvertCMYKToRGB (SWI &4075B)

Converts from the CMYK model to RISC OS RGB colours

On entry

R0 = cyan component
R1 = magenta component
R2 = yellow component
R3 = key (black) component

On exit

R0 = red component
R1 = green component
R2 = blue component

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call converts from the CMY (cyan/magenta/yellow) model with a K (key – ie black) additive to RISC OS RGB colours, allowing easy conversion from colour separations.

All parameters are passed as fixed point 32 bit numbers in the range 0 - 1, with 16 bits below the point and 16 bits above the point. The ‘K’ acts as a black additive and is a value equally subtracted or added to the given CMY values.

This call is not available in RISC OS 2.

Related SWIs

ColourTrans_ConvertRGBToCMYK (page 3-389)

Related vectors

ColourV

ColourTrans_ReadPalette (SWI &4075C)

Reads either the screen's palette, or a sprite's palette

On entry

R0 = source mode, or –1 for current mode, or (if ≥ 256) pointer to sprite area
R1 = source palette pointer, or –1 for current palette, or (if R0 ≥ 256) pointer to sprite name/sprite in area pointed to by R0 (as specified by bit 0 of R4)
R2 = pointer to buffer, or 0 to return required size in R3
R3 = size of buffer (if R2 ≠ 0)
R4 = flags (used if R0 ≥ 256):
bit 0 set ⇒ R1 = pointer to sprite; else R1 = pointer to sprite name
bit 1 set ⇒ return flashing colours; else don't
all other bits reserved (must be zero)

On exit

R2 = pointer to next free word in buffer
R3 = remaining size of buffer

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call reads either the screen's palette, or a sprite's palette. It is the recommended way of doing so. It provides a way for applications to enquire about the palette and always read the absolute values, no matter what the hardware is capable of.

All palette entries are returned as **true** 24bit RGB, passing through the calibration if required. In 256 colour modes the palette is returned fully expanded (ie 256 palette entries, rather than the base 16 entries used by VIDC).

This call is not available in RISC OS 2.

Related SWIs

ColourTrans_WritePalette (page 3-395)

Related vectors

ColourV, PaletteV

ColourTrans_WritePalette (SWI &4075D)

Writes to either the screen's palette, or to a sprite's palette

On entry

R0 = –1 to write current mode's palette, or pointer to sprite area
R1 = –1 to write current palette, else ignored (if R0 = –1); or (if R0 ≥ 0) pointer to sprite name/sprite in area pointed to by R0 (as specified by R4)
R2 = pointer to palette to write
R3 reserved (must be zero)
R4 = flags (used if R0 ≥ 0):
bit 0 set ⇒ R1 = pointer to sprite; else R1 = pointer to sprite name
bit 1 set ⇒ flashing colours in table; else not present
all other bits reserved (must be zero)

On exit

—

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call writes to either the screen's palette, or to a sprite's palette.

256 colour palettes are first compacted to the base 16 entries used by VIDC – but only if the compacted palette expands via the tint mechanism to the original palette. Otherwise the full 256 colours are written.

This call is not available in RISC OS 2.

Related SWIs

ColourTrans_ReadPalette (page 3-393)

Related vectors

ColourV, PaletteV

ColourTrans_SetColour (SWI &4075E)

Changes the foreground or background colour to a GCOL number

On entry

R0 = GCOL number
R3 = flags:
    bit 7 set ⇒ set background, else foreground
    bit 9 set ⇒ set text colour
R4 = GCOL action

On exit

All registers preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call changes the foreground or background colour to a GCOL number (as returned from ColourTrans_ReturnGCOL). You should only use it for GCOL numbers returned for the current mode.

If bit 9 of R3 is set on entry, then this call sets the text colours rather than the graphics colours.

This call is not available in RISC OS 2.

Related SWIs

ColourTrans_ReturnGCOL (page 3-348)

Related vectors

ColourV

ColourTrans_MiscOp (SWI &4075F)

This call is for internal use only. It is not available in RISC OS 2.

ColourTrans_WriteLoadingsToFile (SWI &40760)

Writes a * Command to a file that will set the ColourTrans error loadings

On entry

R1 = file handle

On exit

All registers preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call writes a * Command to the specified file that will set the error loadings within the ColourTrans module. This call is mainly provided to support desktop saving of the loadings.

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

Related SWIs

None

Related vectors

ColourV

ColourTrans_SetTextColour (SWI &40761)

Changes the text foreground or background colour to a GCOL number

On entry

R0 = palette entry
R3 = flags word:
bit 7 set ⇒ set background colour; else set foreground colour
all other bits reserved (must be zero)

On exit

R0 = GCOL
R3 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call changes the text foreground or background colour to the GCOL number (as returned from ColourTrans_ReturnGCOL) that is closest to the specified palette entry. You should only use it for GCOL numbers returned for the current mode.

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

Related SWIs

ColourTrans_SetOppTextColour (page 3-403)

Related vectors

ColourV

ColourTrans_SetOppTextColour (SWI &40762)

Changes the text foreground or background colour to a GCOL number

On entry

R0 = palette entry
R3 = flags word:
bit 7 set ⇒ set background colour; else set foreground colour
all other bits reserved (must be zero)

On exit

R0 = GCOL
R3 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call changes the text foreground or background colour to the GCOL number (as returned from ColourTrans_ReturnGCOL) that is furthest from the specified palette entry. You should only use it for GCOL numbers returned for the current mode.

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

Related SWIs

ColourTrans_SetTextColour (page 3-401)

Related vectors

ColourV

ColourTrans_GenerateTable (SWI &40763)

Sets up a translation table in a buffer

On entry

R0 = source mode, or –1 for current mode, or (if ≥ 256) pointer to sprite area
R1 = source palette pointer, or –1 for current palette, or (if R0 ≥ 256) pointer to sprite name/sprite in area pointed to by R0 (as specified by bit 0 of R5)
R2 = destination mode, or –1 for current mode
R3 = destination palette pointer, or –1 for current palette, or 0 for default for the mode
R4 = pointer to buffer, or 0 to return required size of buffer
R5 = flags:
- bit 0 set ⇒ R1 = pointer to sprite; else R1 = pointer to sprite name
- bit 1 set ⇒ use current palette if sprite doesn't have one; else use default
- bit 2 set ⇒ use R6 and R7 to specify transfer function
- bits 24 - 31 give format of table:
  - 0 ⇒ return pixel translation table (see page 1-780)
  - 1 ⇒ return physical palette table
- all other bits reserved (must be zero)

R6 = pointer to workspace for transfer function (if bit 2 of R5 is set)
R7 = pointer to transfer function (if bit 2 of R5 is set)

On exit

R0 - R3 preserved
R4 = required size of buffer (if R4 = 0 on entry), or preserved
R5 - 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

This call is exactly the same as ColourTrans_SelectTable (see page 3-344), except that it assumes that R5 always contains a valid flags word.

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

Related SWIs

ColourTrans_SelectTable (page 3-344)

Related vectors

ColourV

* Commands

*ColourTransLoadings

Sets the red, green and blue weightings used when trying to match colours

Syntax

```
*ColourTransLoadings redweight greenweight blueweight
```

Parameters

redweight	red weighting used when trying to match colours
greenweight	green weighting used when trying to match colours
blueweight	blue weighting used when trying to match colours

Use

*ColourTransLoadings sets the red, green and blue weightings used when trying to match colours (as described in the section entitled *Finding a colour* on page 3-334).

The main purpose of this command is to enable the Task Manager to save the calibration when a desktop save is done. You should not use it yourself.

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

Example

```
*ColourTransLoadings &2 &4 &1
```

Related commands

None

Related SWIs

ColourTrans_WriteLoadingsToFile (page 3-400)

Related vectors

ColourV

*ColourTransMap

Sets up a calibration table from its parameters

Syntax

`*ColourTransMap` *`RRGGBBDD RRGGBBDD RRGGBBDD RRGGBBDD`* *etc*`.`

Parameters

`RRGGBBDD`	8 hex digits, such that &RRGGBBDD is the number to be placed in the calibration table

Use

*ColourTransMap sets up a calibration table from its parameters. The number of parameters passed for each component must have been specified in a previous *ColourTransMapSize command.

The main purpose of this command is to enable the Task Manager to save the calibration when a desktop save is done.

This command is not available in RISC OS 2.

Example

```
*ColourTransMap 01000000 FF0000FF 00020000 00FE00FF etc
```

Related commands

*ColourTransMapSize

Related SWIs

ColourTrans_WriteCalibrationToFile (page 3-383)

Related vectors

ColourV

*ColourTransMapSize

Sets how parameters will be passed in the next *ColourTransMap command

Syntax

`*ColourTransMapSize` *n1 n2 n3*

Parameters

n1	number of parameters to be passed in *ColourTransMap for component 1
n2	number of parameters to be passed in *ColourTransMap for component 2
n3	number of parameters to be passed in *ColourTransMap for component 3

Use

*ColourTransMapSize sets the number of parameters that will be passed in the next *ColourTransMap command for each component. It hence also sets the size of the resultant calibration table, which will be (3 + n1 + n2 + n3) words long. The values n1, n2 and n3 are given in the reverse order to a standard calibration table.

The main purpose of this command is to enable the Task Manager to save the calibration when a desktop save is done.

This command is not available in RISC OS 2.

Example

```
*ColourTransMapSize 8 10 8
```

Related commands

*ColourTransMap

Related SWIs

ColourTrans_WriteCalibrationToFile (page 3-383)

Related vectors

ColourV

61 The Font Manager

Introduction

A *font* is a 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

References in this chapter to the RISC OS 2 Font Manager describe the **outline** Font Manager that is supplied with Release 1.02 of Acorn Desktop Publisher. The RISC OS 2 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 to speed painting
- 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

The Font Manager also uses OS coordinates as a measurement system. 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. The list is cached and so is fast to access – except under RISC OS 2, where it’s consequently a slow process to get the font list unless you cache it yourself, which we recommend.

Another SWI will return a handle for a given font style. A handle is a number 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 a threshold beyond which the Font Manager will discard as much cached information as possible in an endeavour not to let the cache grow. However, if many more fonts are in use than are reasonable for the configured threshold, the Font Manager may be forced to let the cache grow past this point.

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, which considerably slows response. 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.

For screen modes with more than 16 colours, this sets up a ‘pseudo palette’ that indirects into the real palette.

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 control sequences, 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 control sequences, 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 smallest box which is large enough to contain any character in the set. This is called the *font 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 loops 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 (page 3-446) will go from millipoints, while Font_Converttopoints (page 3-447) 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 (page 3-456). You can also set this with Font_SetScaleFactor (page 3-457), although we recommend that you don't do so under the desktop, as other applications may assume the default. If you must alter this value, you should at the very least restore it before polling the Wimp.

Font files

The font files relating to a font are all held in a single directory structure consisting of one or more *font* subdirectories (for different weights and styles/angles) and one or more *encoding* subdirectories. All Acorn font names should conform to:

`fontname.[weight.[style]]`

The weight element can only be omitted if there is no style element either, eg for a Symbol font.

Files held within this structure are:

Filename	Contents
IntMetrics	metrics information for default encoding
IntMetric0	metrics information for encoding /Base0
IntMetric*n*	metrics information for encoding to /Base*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
Outlines 0	outline file for encoding /Base0
Outlines *n*	outline file for encoding to /Base*n*
Messages*n*	mapping of font identifiers to names for country *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 4-463.

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 RISC OS character sets, see *Table D: Character sets* on page 4-569.

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 identifier, regardless of size, whereas each size of font requires a separate slave font. If the data is loaded directly from a bitmap file into the slave font, the master font is not required.

Font names and identifiers

Font identifiers are the names of font subdirectories, and are used for all programmer’s interfaces to the Font Manager, such as SWIs. They are constant across all countries. Font names are the local form of a font identifier for a particular country, and are used for all user interfaces to the Font Manager, such as menus.

Messages files

Font names are obtained by looking in the file *Fontprefix*.Messages*countryno*, using the font identifier as a key. For example, a UK Messages file would be named Messages1. This allows internationalisation by having an extra level of indirection between font identifiers and font names. The file *Fontprefix*.Messages is used as a default if the country-specific file is not present.

The Font Manager only actually scans the font directory if no Messages file can be found. Of course, reading a Messages file is much faster than scanning the font directory.

Messages files allow font paths to become much more effective, since new font directories can be added to the list of known fonts without losing references to other font directories. This and the fact that the Font Manager knows exactly where each font is held makes it possible for a user to put fonts on several floppy discs and still use them effectively. Messages files also allow you to set the default font in a family (eg selecting just 'Trinity' in a font menu can be made to select 'Trinity.Medium', rather than just the first entry in the sub-menu).

Details of the format of Messages files are in *Appendix E: File formats* on page 4-463 – just as for all other font file formats.

Referencing fonts by identifier

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 identifier. 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).

The commands *FontInstall, *FontRemove and *FontLibrary add directories to Font$Path, or remove them. Service_FontsChanged is then issued to notify module-based applications that they should update their list of available fonts by calling Font_ListFonts again. These commands are not available under RISC OS 2, but where

possible, you should use them. (Non module-based applications must call Font_ListFonts each time they require a list of available fonts, as they have no way of knowing when the list has changed.)

RISC OS 2

Under RISC OS 2 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 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 (page 3-428) 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 (page 3-431) must be used.

Handles

Font_ReadDefn (page 3-432) 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 (page 3-448). This setting stays until changed by another call to this function, or while painting, by a character command to change the handle.

Font_CurrentFont (page 3-449) 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 Font Manager will endeavour not to use more than this, but may have to should there be many more fonts in use than are reasonable for the configured FontMax.

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 (page 3-426) 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 to save the font cache in a known state. You can then use *LoadFontCache to reload it later, but there are restrictions when doing so:

- The cache must not contain any claimed fonts (ie ones that are in use).
- The format of the loaded cache must be understood by the Font Manager loading the cache. In practice this generally means that the cache must have been saved by the same version of the Font Manager as is loading it.

Using saved font caches 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 (page 3-461). This sets the background colour, the foreground colour and the range of colours in between.

Physical colours

Font_SetPalette (page 3-463) 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.

256 colour screen modes

There can be a maximum of 16 colours used. For screen modes having more than 16 colours, the above calls instead set up a ‘pseudo palette’ that provides for up to 16 indirect references to colours in the real palette. Such a ‘pseudo palette’ must be defined before trying to paint fonts.

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 (page 3-218), or ColourTrans_SetFontColours (page 3-370), or a control sequence 19,… in a string passed to Font_Paint (page 3-437); these all use colours that are already in the palette.

Thresholds

The setting of intermediate levels uses threshold tables. These can be read with Font_ReadThresholds (page 3-465) or set with Font_SetThresholds (page 3-468). They use a lookup table that is described in Font_ReadThresholds.

Painting

Font_Paint (page 3-437) 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 control 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)
21	comment string that is not displayed
25	change underline position and thickness
26	change font handle
27	set new 4-entry transformation matrix (not in RISC OS 2)
28	set new 6-entry transformation matrix (not in RISC OS 2)

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 (page 3-451). 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 in between two characters, 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 (page 3-444). 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 identifier 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.

Under RISC OS 3 and later, the identifier can contain a transformation matrix specified in the same way as the font identifier passed to Font_FindFont (page 3-428). This allows simple generation of oblique fonts. For an example, see the RISC OS 3 file Resources:$.Fonts.Corpus.Oblique.Outlines0.

Handling mode changes

For efficiency, the Font Manager caches the value in millipoints of the last y coordinate to which it painted, and reuses that value if it paints to the same y coordinate next time. However, this cached value does not take account of screen eigen values, and both RISC OS 2 and RISC OS 3 (version 3.00) fail to notice a mode change and update the cached y position to take account of the new eigen values. Consequently the Font Manager will paint to the wrong y position on the screen.

To work around this, on receipt of a Service_ModeChange (page 1-638) your application should call Font_Paint (page 3-437) to paint a null string at coordinates (–1, –1), which is off the screen. This will invalidate the cached value, so a subsequent paint to the same y coordinate as before the mode change will then work correctly. For example you would make this call in BASIC:

```
SYS "Font_Paint",,"",&10,-1,-1
```

Service Calls

Service_FontsChanged (Service Call &6E)

New Font$Path detected

On entry

R1 = &6E (reason code)

On exit

All registers preserved

Use

This is issued by the Font Manager to notify any module-based applications that they should call Font_ListFonts to update the list of available fonts.

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 identifier (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 identifier, 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 font identifier 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. 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 32-bit fixed point, with the integer part in the top 16 bits; the last two numbers are offsets, in $1/1000$th of an em. Each number – including the last one – must be followed by a space.

Spaces are significant in the above syntaxes; you must include them only where shown.

The *font identifier* is the name of the font directory without the Font$Path prefix, and is invariant in any territory. These are used in all programmer's interfaces to the Font Manager, such as SWIs. The *font name* is the name of the font (ie the one displayed to the user) in the given territory. These are used in all user interfaces to the Font Manager, such as menus.

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`*`territory encname`*`\F`*`fontid`*`\f`*`territory fontname`*

Since *fontid*\f*territory fontname* is also accepted by Font_FindFont, when prepending '\E*encid*\e*territory 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 (page 3-431)

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 (page 3-428)

Related vectors

None

Font_ReadDefn (SWI &40083)

Read details about a font

On entry

R0 = font handle
R1 = pointer to buffer to hold font identifier, or 0 to return required size of buffer – if R3 = 'FULL' on entry
R3 = &4C4C5546 ('FULL') to return full information about encoding and matrix

On exit

R0, R1 preserved
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

or, if R1 = 0 and R3 = 'FULL' on entry:

R0, R1 preserved
R2 = required buffer size to hold full information
R3 - R7 corrupted

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.

By setting R3 to ‘FULL’, you can get the full font identifier, including such information as its transformation matrix and encoding. You can also find the required size of buffer to hold this information by setting R1 to 0 on entry. These features are not available in RISC OS 2.

Related SWIs

None

Related vectors

None

Font_ReadInfo (SWI &40084)

Get the font bounding box

On entry

R0 = font handle

On exit

R0 preserved
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 every character in the font. This is called the font bounding box.

You should use the SWI Font_CharBBox (see page 3-454) in preference to this one.

Related SWIs

Font_CharBBox (page 3-454), Font_StringBBox (page 3-471)

Related vectors

None

Font_StringWidth (SWI &40085)

Calculate how wide a string would be in the current font

On entry

R1 = pointer to string
R2 = maximum x offset before termination in millipoints
R3 = maximum y offset before termination in millipoints
R4 = character code of ‘split’ character (–1 for none); eg 32 for space
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 in the current font.

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 control 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 caret position in a string if you know the coordinates (although Font_FindCaret is better for this)
- finding the caret 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 (page 3-492) in preference to this one – except under RISC OS 2, where it is not available.

Related SWIs

Font_FutureFont (page 3-451), Font_ScanString (page 3-492)

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 (bits 0, 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 ignores the values of R0 and of R5 - R7, and behaves as though bits 2,3 and 5 - 31 inclusive of R2 are clear.

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 (in which case bit 5 must be clear), the text is justified between the start coordinates (given in R3, R4) and the last position of the graphics cursor (see below).

  In fact, the graphics cursor 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 (in which case bit 4 must be clear), the text is justified by adding additional spacing between words and letters. These additional offsets are specified in a coordinate block pointed to by R5.

  You can achieve left justification by simply setting these two values to zero.

If both bits 0 and 5 of R2 are clear then the string isn't justified.

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 rubout box is defined by two points previously visited by the graphics cursor (see below).
- If bit 5 of R2 is set, then two coordinate pairs held 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.

Setting coordinates using the graphics cursor

To set the justification coordinates using the graphics cursor, you must have previously called a VDU 25 move command. Likewise, to set the rubout coordinates using the graphics cursor you must also 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 specify 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.

Using the coordinate block

The coordinate block pointed to by R5 contains eight words: these give additional spacing to use to achieve justification, and coordinates for the rubout box. The values are in millipoints (since bit 4 of R2 must be 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)

Transformation matrices

If bit 6 of R2 is set (in which case bit 4 must be clear), 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 millipoints (since bit 4 of R2 must be 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); 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 ‘sufficient null bytes for subsequent values to be word-aligned’. The equation for this is:

number of null bytes = (address of *align* + 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 – since bit 4 of R2 must be clear – are in millipoints. 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

Normally the string is painted up to its terminator. However, you can paint a substring by setting bit 7 of R2, and specifying the length of the substring in R7.

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 used only 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 3-370). Again, RISC OS 2 does not support this control sequence; but it does still provide ColourTrans_SetFontColours, which you should use in preference to 17,… or 18,… control sequences.

After the call, the current colours are updated to the last values set by this control sequences.

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 control sequences.

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 (page 3-461) 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. This is because 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 *fontidentifier.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

Related SWIs

Font_StringWidth (page 3-435), Font_ScanString (page 3-492)

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 - 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 ‘loops’ 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. You can get these colours by calling ColourTrans_ReturnColourNumber.

Related SWIs

ColourTrans_ReturnColourNumber (page 3-352)

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 (page 3-447), Font_ReadScaleFactor (page 3-456), Font_SetScaleFactor (page 3-457)

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 (page 3-446), Font_ReadScaleFactor (page 3-456), Font_SetScaleFactor (page 3-457)

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 3-437). Where possible, you should do so in preference to using this SWI.

Related SWIs

Font_Paint (page 3-437), Font_CurrentFont (page 3-449), Font_SetFontColours (page 3-461)

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_Paint (page 3-437), Font_SetFont (page 3-448), Font_SetFontColours (page 3-461), Font_SetPalette (page 3-463), Font_SetThresholds (page 3-468)

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 (page 3-435), Font_Paint (page 3-437)

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 (page 3-492) in preference to this one – except under RISC OS 2, where it is not available.

Related SWIs

Font_StringWidth (page 3-435), Font_FindCaretJ (page 3-469), Font_ScanString (page 3-492)

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

R0 preserved
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 (page 3-434), Font_StringBBox (page 3-471)

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 (page 3-446), Font_Converttopoints (page 3-447), Font_SetScaleFactor (page 3-457)

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, 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 (page 3-446), Font_Converttopoints (page 3-447), Font_ReadScaleFactor (page 3-456)

Related vectors

None

Font_ListFonts (SWI &40091)

Scan for fonts, returning their identifiers 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-compatible mode (see below)
- bit 16 set ⇒ return font identifier in buffer pointed to by R1 (or required size of buffer for next identifier if R1 = 0)
- bit 17 set ⇒ return local font name in buffer pointed to by R4 (or required size of buffer for next name if R4 = 0)
- 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 if R1 and R4 = 0)
- 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 length of data placed in buffer
R4 preserved
R5 = required size of buffer pointed to by R4 (if R4 = 0 on entry); else length of data placed 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 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 3-153).

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 3

The Font Manager in the RISC OS 3 ROMs (ie Font Manager 3.07 or earlier) has a bug in its handling of indirected menu titles. To work around this, you must use MessageTrans to decode the ‘FontList’ token in the Fonts resource file; if its length is more than 12 characters you must set the ‘indirected menu title’ bit of the first menu item, and otherwise you must clear it.

Notes on RISC OS 2

In the ‘RISC OS 2-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, 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 - 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 (page 3-448), Font_CurrentFont (page 3-449), Font_SetPalette (page 3-463)

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 R4 and R5

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 (page 3-218) or ColourTrans_SetFontColours (page 3-370) 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 pages 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, 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 R4 and R5 as true 24-bit palette values (where white is &FFFFFF00, rather than &F0F0F000). Otherwise, for compatibility, palette values are processed as follows:

```
R4 = (R4 AND &F0F0F000) OR ((R4 AND &F0F0F000) >> 4)
R5 = (R5 AND &F0F0F000) OR ((R5 AND &F0F0F000) >> 4)
```

Thus the bottom nibbles of each gun are set to be copies of the top nibbles. Furthermore, this call now uses PaletteV 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 PaletteV is not intercepted, it calls OS_Word 12 to do so.

Related SWIs

Font_SetFontColours (page 3-461)

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_SetFontColours (page 3-461), Font_SetPalette (page 3-463), Font_SetThresholds (page 3-468)

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.

Related SWIs

Font_SetFontColours (page 3-461), Font_SetPalette (page 3-463), Font_ReadThresholds (page 3-465)

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 (page 3-492) in preference to this one – except under RISC OS 2, where it is not available.

Related SWIs

Font_FindCaret (page 3-452), Font_ScanString (page 3-492)

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 (page 3-492) in preference to this one – except under RISC OS 2, where it is not available.

Related SWIs

Font_ReadInfo (page 3-434), Font_CharBBox (page 3-454), Font_ScanString (page 3-492)

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_SetFontColours (page 3-461), Font_SetPalette (page 3-463), Font_SetThresholds (page 3-468)

Related vectors

None

Font_MakeBitmap (SWI &40099)

Make a font bitmap file

On entry

R1 = font handle, or pointer to font identifier
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.*fontidentifier*.*encoding*.

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

Related SWIs

Font_SetFontMax (page 3-478)

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 identifier as it would have been constructed from Font$Path and the font identifier. This means that each of the elements of Font$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
...                                    replace old file with new one
SYS "Font_UnCacheFile",,"filename",1
```

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 (in points, or in pixels × 72 × 16 under RISC OS 2)
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 (page 3-480)

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 (in points, or in pixels × 72 × 16 under RISC OS 2)
R6, R7 may be corrupted

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 (page 3-478)

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

R0 preserved
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.

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.

Any characters which are only available as bitmaps will either generate an error (if bit 4 of R0 is set), or not be output.

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)

Reads the full metrics information held in a font's IntMetrics file

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 full 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 you should ignore them.

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 (axis-aligned) 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.

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. (You can also paint kerned text using Font_Paint (page 3-437), which may be an easier option.)

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.

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 Messages*n* 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 fontname`*
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

Font_ListFonts (page 3-458)

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 5 must be set)
- bit 19 set ⇒ return matrix applying at end of string in buffer pointed to by R6 (bit 6 must be set)
- bit 20 ⇒ return number of split characters in R7 (bit 7 must be set)
- 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

R0 preserved
- R1 = pointer to point in string of caret position – if bit 17 of R2 is set; else to split point, or end of string if splitting not required

R2 preserved
R3, R4 = x, y coordinate offset to caret position – if bit 17 of R2 is set; else to split point, or end of string if splitting not required

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, if at all.

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 3-437. Below we merely describe the changes and additions relative to that SWI.

This call is not available in RISC OS 2.

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 (page 3-435), Font_FindCaret (page 3-452), Font_FindCaretJ (page 3-469), Font_StringBBox (page 3-471)

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.

To set font colours you should either use ColourTrans_SetFontColours (see page 3-370) or Font_Paint control sequence 19 (see page 3-440).

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.

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.

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.

Related SWIs

None

Related vectors

None

Font_FindField (SWI &400A6)

Returns a pointer to a specified field within a font identifier

On entry

R1 = pointer to font identifier
R2 = character code of qualifier required

On exit

R1= pointer to value following qualifier in string (if field present); else preserved
R2 = 0 if field not present; 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, given a font string and a qualifier that specifies a field within that string, returns a pointer to the specified field.

The ‘F’ field is space-terminated, while all others are control-character terminated.

This call is not available in RISC OS 2.

Related SWIs

Font_ApplyFields (page 3-504)

Related vectors

None

Font_ApplyFields (SWI &400A7)

Merges a new set of fields with those already in a given font identifier

On entry

R0 = pointer to original font identifier
R1 = pointer to set of fields to be added (in format of a font identifier)
R2 = pointer to output buffer, or 0 to get required new size of buffer
R3 = size of output buffer

On exit

R0 - R2 preserved
R3 = remaining size of buffer, or incremented by the length of the output string (excluding its null terminator)

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 merges a new set of fields with those already in a given font identifier, replacing existing fields and adding new ones. You can also delete existing fields by specifying a null field to replace it.

This operation is performed in two passes:

1 Copy fields in [R0] from [R1] if present, else [R0]

2 Copy fields in [R1] from [R1] if not present in [R0]

This call is not available in RISC OS 2.

Related SWIs

Font_FindField (page 3-503)

Related vectors

None

Font_LookupFont (SWI &400A8)

Returns information about a particular font

On entry

R0 = font handle, or 0 for current handle
R1 = 0
R2 = 0

On exit

R0, R1 preserved
R2 = characteristics of font:
bit 0 set ⇒ font is old ‘x90y45’ bitmap format
bit 1 set ⇒ font is in ROM
bit 8 set ⇒ font is monochrome only, irrespective of value of FontMax2
bit 9 set ⇒ font is filled with non-zero rule, rather than even-odd
all other bits reserved and should be ignored
R3-R7 corrupted

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 information about a particular font. On exit R2 contains a set of flags describing how the font is rendering, and other characteristics.

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

Related SWIs

None

Related vectors

None

*Commands

*Configure FontMax

Sets the configured maximum desirable 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 desirable 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.

The Font Manager has to keep permanently in its cache some information on each font in use. Consequently, if many more fonts are in use than are reasonable for the configured FontMax, the Font Manager may be forced to let the cache grow past this point.

Example

```
*Configure FontMax 256K
```

Related commands

*Configure FontSize

Related SWIs

Font_CacheAddr (page 3-426), Font_SetFontMax (page 3-478), Font_ReadFontMax (page 3-480)

Related vectors

None

*Configure FontMax1

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

Syntax

```
*Configure FontMax1 max_pointsize
```

Parameters

max_height	maximum height of font at which to scale from a bitmap font; units are points, except under RISC OS 2, which uses pixel height (see below)

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 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 points, except under RISC OS 2 which uses pixel height:

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

The pixel height corresponds to different point sizes on different resolution output devices.

Example

```
*Configure FontMax1 25
```

Related commands

*Configure FontMax2

Related SWIs

Font_SetFontMax (page 3-478), Font_ReadFontMax (page 3-480)

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 height of font at which to scale from outlines to anti-aliased bitmaps; units are points, except under RISC OS 2, which uses pixel height (see below)

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 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 points, except under RISC OS 2 which uses pixel height:

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

The pixel height corresponds to different point sizes on different resolution output devices.

Example

```
*Configure FontMax2 20
```

Related commands

*Configure FontMax1, *Configure FontMax3

Related SWIs

Font_SetFontMax (page 3-478), Font_ReadFontMax (page 3-480)

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 height of font at which to retain bitmaps in the cache; units are points, except under RISC OS 2, which uses pixel height (see below)

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 use1 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 points, except under RISC OS 2 which uses pixel height:

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

The pixel height corresponds to different point sizes on different resolution output devices.

Example

```
*Configure FontMax3 35
```

Related commands

*Configure FontMax2

Related SWIs

Font_SetFontMax (page 3-478), Font_ReadFontMax (page 3-480)

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 width of font at which to use horizontal subpixel anti-aliasing; units are points, except under RISC OS 2, which uses pixel width (see below)

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 points, except under RISC OS 2 which uses pixel width:

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

The pixel width corresponds to different point sizes on different resolution output devices.

Example

```
*Configure FontMax4 0
```

Related commands

*Configure FontMax5

Related SWIs

Font_SetFontMax (page 3-478), Font_ReadFontMax (page 3-480)

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 height is set in points, except under RISC OS 2 which uses pixel height:

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

The pixel height corresponds to different point sizes on different resolution output devices.

Example

```
*Configure FontMax4 0
```

Related commands

*Configure FontMax5

Related SWIs

Font_SetFontMax (page 3-478), Font_ReadFontMax (page 3-480)

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 (page 3-426)

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.          The last '.' is essential
Corpus.Medium
Portrhouse.Standard
Trinity.Medium
```

Related commands

None

Related SWIs

Font_FindFont (page 3-428), Font_ListFonts (page 3-458)

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.

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.

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, its identifier is given, together with 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  Cache memory
    ----                   ----               ---------    ---  ------------
  1.Homerton.Medium        12 point            90x45         3      4 Kbytes
  2.Homerton.Medium        master ROM outlines               1    696 bytes

Cache size:  32 Kbytes
      free:  24 Kbytes
```

Related commands

None

Related SWIs

Font_ListFonts (page 3-458)

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.

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 that was previously saved using *SaveFontCache back into the font cache.

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.

Example

```
*LoadFontCache scsi::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.

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.

62 SuperSample module

Introduction and Overview

The SuperSample module provides SWIs for the use of the Font Manager. You must not use them in your own code.

This module is not available under RISC OS 2.

SWI calls

Super_Sample90 (SWI 40D80)

This SWI is for internal use only. You must not use it in your own code.

Super_Sample45 (SWI 40D81)

This SWI is for internal use only. You must not use it in your own code.

63 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 the path:

- 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.

User units

The coordinates used in a path can be in any units that you wish to use. These are converted 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 user units differ too much 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 3-565. 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; for example plotting with a scale factor of 1 – which is &1000 Transform units – and with a translation of (64, 32) – which are respectively &4000 and &2000 internal Draw units – would use the values [&1000, 0, 0, &1000, &4000, &2000].

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 each passed as a 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 3-565. 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 (if required)
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 converted 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 3-538.

The fill style is as on page 3-540, 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 3-539, 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 (ie storage requirement) does not change during the call, such as when doing a transformation only.
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 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 3-544, 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 3-538.

The fill style is as specified on page 3-540 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 3-544.

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 (see below)
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 (if required)
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 3-538.

The fill style is as specified on page 3-540 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 3-541, 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 3-544, 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 (see above). Use Draw_ProcessPath to get around this problem by processing it before stroking.

Related SWIs

Draw_StrokePath (page 3-552)

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 3-538.

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 (page 3-550)

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 3-538.

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 (page 3-552)

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 3-538.

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 (page 3-552)

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(80000) : 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.

Part 9 – Printing

64 Printer Drivers

Introduction

Printing from applications

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 printer, a PostScript printer, and a plotter.

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.

To simplify printer driving further, output to the printer is done using 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, using the printer's resolution and set of features to the best. Thus applications need not know about printer specific operation, but this does not result in lack of fine control of the printer.

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

The structure of the printing system

Printer drivers are written to support a general class of printers, such as PostScript printers. Under RISC OS 3 you can have more than one printer driver installed at the same time, and it is easy to switch between them. Support for this is supplied by a sharer module, and also by the new Printer manager application, which also provides facilities that allows users to control the unique attributes of each type of printer.

The structure of the printing system need not concern application writers; you do not need to know which part of the printing system is handling your calls. Further details are given in the section entitled *The structure of the printing system* on page 3-598, should you be interested.

Overview

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

Millipoints

Many of the printer driver SWIs deal with an internal measurement system, using millipoints. This is $^1/_{1000}$ of a point, or $^1/_{72000}$ 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 $^1/_{180}$ of an inch, so each OS unit is $^2/_5$ 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 (ie 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.

Printing a document from an application

Overview of printing

This section describes how to print from an application.

Initiating printing under the Wimp

If your application is running under the Wimp, it must initiate printing using the Wimp message protocol described in the section entitled *The Printer protocol* on page 3-256.

Get information on the printer you're going to use

Your application should not make any assumptions about the printer that it is going to use; RISC OS supports many types of printers, and your users could be using any of them. Similarly, you shouldn't make assumptions about the printer driver you'll be using.

Instead you should use the SWI PDriver_Info (see page 3-611) to find out any information you need to know about the printer and printer driver that you're using. You should do so each time you start printing, rather than when your application is loaded. This is because the user may change either the way in which they are using a printer or the printer that they are using during the time your application is printing. The information this call returns includes:

- the type of printer driver in use
- the version number of that printer driver
- the x and y resolution of the printer in use
- the name of the printer in use
- the halftone resolution of the printer (if any).

It also returns a features word giving a bit mask showing:

- the printer driver's colour and shading capabilities
- the printer driver's plotting capabilities, such as its ability to handle filled shapes, thick lines, overwriting and transformed sprites
- the printer driver's support for optional features such as screen dumps, arbitrary transformations, insertion of illustrations, and font declaration and handling.

You may wish to – or indeed have to – change the behaviour of your application based on this information. Note that many colour limitations you might be worried about can be overcome by the printer drivers' own halftoning.

There are two other informational SWIs that you may find useful:

- PDriver_CheckFeatures (see page 3-617) provides a quick way of checking if specific features are available by comparing a bit mask of features you desire to be present against the printer driver's own features word.
- PDriver_PageSize (page 3-618) returns the size of the paper in use and its printable area.

Open the printer: file for output

To start a print job, you should first open 'printer:' as an output file. This device independent name is used so that the Printer application can control the actual destination of printed output using the OS_Byte 5 call (for details of this call, see page 1-522).

You may – if you wish – 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.

Start a new print job

The next stage of printing is to start a new print job by passing the file handle of the output file you just opened (see above) to PDriver_SelectJob (see page 3-622).

This call suspends the current print job, if there is one, and sets up a new job that uses the file handle that you passed for its output.

Declare the fonts your document uses

You should then declare any fonts that your document uses, assuming the printer driver you're using supports this feature. (You can find this out using PDriver_Info; see the section above entitled *Get information on the printer you're going to use*). Certain printer drivers need this information before printing begins; for example, the PostScript driver needs it to generate more efficient output, to perform font downloading, and to conform with structuring rules for PostScript documents.

To declare the fonts, you should call PDriver_DeclareFont (see page 3-648) for each distinct font that your document uses. The definition of what is a 'distinct font' is strict, and is given in the documentation of this SWI. Having declared each font, you must make one further call of this SWI, passing special values to indicate the end of the list of fonts. Even if your document does not use any fonts you should still make this 'end of list' call; the printer driver then knows that your application is aware of this call, and uses no fonts.

Specify the rectangles to be printed on a page

You're now ready to print a page. The first step is to specify as rectangles those area(s) of your document that you wish to have printed on the page.

You must call PDriver_GiveRectangle (see page 3-632) for each rectangle, specifying the location and size of the rectangle within the document, an ID for the rectangle (allocated by you), a transformation to apply to it before printing, and its location and background colour on the printed page.

Typically you will just specify a single rectangle consisting of a whole page of your document. An example of the use of multiple rectangles would be for printing 'thumbnails' (ie printing multiple pages of your document on a single page).

Print the rectangles specified by the printer driver

To actually start printing you call PDriver_DrawPage (see page 3-635). This returns the first rectangle for your application to plot. This may be all of a rectangle you specified for printing, or it may only be a strip of that rectangle. You should plot the rectangle using calls that normally output to the screen; the printer driver intercepts these calls, and converts them to printed output. For general information see the section entitled *Trapping of output calls* on page 3-573, and the section entitled *Guidelines on output calls to use* on page 3-574. For a more detailed description of how the printer drivers handle each output call, see the section entitled *The output calls in detail* on page 3-575.

Having plotted the first rectangle, you should find any other rectangles to plot by repeatedly calling PDriver_GetRectangle (see page 3-637), plotting each rectangle as it is returned. Eventually the call will return a special value indicating that there are no more rectangles to print.

You must not make any assumptions about the returned rectangles. The printer driver is free to request any rectangle from the area(s) that you specified be printed; it may do so in any order it pleases, as many times as it pleases. A dot matrix driver, for instance, may get the output a strip at a time to conserve workspace, and may make multiple passes over a strip (particularly if it uses a multi-coloured ribbon); whereas a PostScript driver can process an entire page in one go.

Similarities with the Window Manager

You may have noticed that this procedure is very similar to the process used by the Wimp to redraw windows: Wimp_RedrawWindow initiates the redraw, returning the first rectangle to draw, and Wimp_GetRectangle returns all subsequent windows to redraw. You should find that with care you can share a lot or all of the code to redraw windows and to print.

Draw any other pages

For each page, you must repeat the procedure in the above two sections (*Specify the rectangles to be printed on a page*, and *Print the rectangles specified by the printer driver*).

End the print job

To end the print job, you must call PDriverEndJob (see page 3-626).

Close the output file

Finally, you must close the output file you've been using (typically 'printer:').

Error handling whilst printing

If you get an error from any printing SWI or from any other call whilst printing pages, you must do one of the following:

- Correct the cause of the error and continue
- **Immediately** call PDriver_AbortJob to end the current job **before** any error message or other output occurs.

  Because the printer driver intercepts output calls this may itself cause an error; thus you may get an infinite loop where an error causes an error, which causes an error, which…

This implies that all calls inside the main print loop **must** be error-returning SWIs (ie have their 'X' bit set; see the chapter entitled *An introduction to SWIs* on page 1-23, and the chapter entitled *Generating and handling errors* on page 1-41).

There are also changes made to error handling within printing to ensure that Escape conditions and errors are not ignored for an undue length of time; see the section entitled *Error handling changes* on page 3-595.

Multitasking whilst printing

Multitasking (ie calling Wimp_Poll) whilst printing has an obvious advantage:

- You can use other applications whilst printing takes place.

However, there are problems associated with it:

- Multitasking is incompatible with the queueing mechanism used by the Printers application. If you call Wimp_Poll whilst printing, the Printers application will assume that you have finished; if there is another job in the queue it will continue with that, which will obviously cause an immediate error. Future versions of RISC OS may address this problem.

- It is difficult to arrange your printing such that you call Wimp_Poll sufficiently often to confer a useful degree of multitasking on the rest of the system, since you cannot predict for all classes of printer the size and complexity of rectangles that you will be asked to print.

  To do so properly, you need to use a timer.
- You must select the previous print job before polling the Wimp, and reselect your own job on return.

In practice these problems are likely to outweigh any advantages conferred by multitasking. If you have any doubts as to whether or not you should multitask, we recommend that you don't.

Code skeleton

This example code skeleton may help you to understand the above:

```
Use Printers message protocol if running under the Wimp

PDriver_Info
REM check what features are available (eg PDriver_DeclareFont)

OPEN printer:

PDriver_SelectJob

IF driver supports PDriver_DeclareFont THEN
        WHILE fonts to be declared
                PDriver_DeclareFont font
        ENDWHILE
        PDriver_DeclareFont end of font list
ENDIF

FOR each page to print
        REPEAT
                PDriver_GiveRectangle
        UNTIL all rectangles declared
        REM typically only one rectangle given, specifying whole page

        PDriver_DrawPage
        WHILE more rectangles to print
                plot returned rectangle using supported output calls
                PDriver_GetRectangle
        ENDWHILE
ENDFOR

PDriver_EndJob

CLOSE printer:
```

Other calls for controlling a print job

PDriver_CurrentJob (page 3-624) will tell you the file handle for the currently active print job.

PDriver_EnumerateJobs (page 3-640) allows you to scan through all the print jobs that the printer driver currently knows about.

PDriver_CancelJob (page 3-638) 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 (page 3-630) will abort all print jobs known to the printer drivers. Normally, you should never have to use this command. It may be useful during development of an application as an emergency recovery measure.

Trapping of output calls

Software vectors

When printing occurs, the printer driver intercepts software vectors through which pass calls that may output to the screen. These are:

- ByteV
- ColourV
- DrawV
- SpriteV
- WrchV.

It treats the intercepted calls in different ways:

1 It appropriately processes the call itself; typically this produces printed output, or alters the printer driver's own record of the graphic state for the current print job.

2 It faults the call as one that is inappropriate to call in the context of a print job.

3 It ignores the call as one that is irrelevant to the print job, and has no wider meaning to the rest of RISC OS.

4 It passes the call on to RISC OS, as it is one that is relevant to other parts of the system.

A few calls are both processed by the printer driver and passed on to RISC OS.

Font manager SWIs

Furthermore, when printing starts the printer driver issues a service call which alters the font manager's SWI handling:

- It processes certain SWIs itself, as normal.
- It passes certain SWIs to the printer driver using an internal mechanism. The printer driver may then:
  1 process the call
  2 ignore the call.
  The font manager does not process such SWIs itself.

Guidelines on output calls to use

This section outlines which calls you should use for output to the printer drivers.

General advice

Only use overwriting; do not use logical operations such as AND, OR, EOR and NOT, as many types of printer (eg PostScript printers, plotters) cannot support them. Avoid using ECFs.

Setting colours

You should set colours for all printing using appropriate calls to the ColourTrans module, rather than by other calls (such as Font_SetFontColours). The ColourTrans calls are independent of the current screen mode and palette, and ensure that the colours rendered by the printer are the best approximations it is able to produce.

All other calls that set colours take the colour to be printed, choose the closest colour available from the current screen palette, and then ask the printer to render that colour. So the printer produces its best approximation to the screen palette's best approximation. Using these other calls:

- makes the printer driver output dependent on the current screen mode and palette
- artificially limits the printer driver to the number of colours displayed on the screen, which can be particularly embarrassing if (say) a user were to try to print in colour whilst in a 2 colour mode.

We therefore recommend that you do not set colours with these other calls.

Graphic object output

You should use calls to the Draw module to print object-oriented graphics such as rectangles. This is preferable to using VDU and PLOT sequences.

Painting fonts

You should paint fonts using font manager SWIs rather than VDU sequences. Set colours using ColourTrans calls, or control (19) sequences within the string to be painted – as these in fact use ColourTrans. Do not set colours using font manager calls, or other (non-19) control sequences.

Sprite output

Use OS_SpriteOp calls to output sprites. You should set colours for the sprite using a translation table, set up using ColourTrans calls during the print job.

Character output

You should use the OS_Write… SWIs, OS_NewLine and OS_PrettyPrint for character output. Do not use OS_PrintChar.

You should avoid OS_Byte calls and VDU sequences wherever possible – in particular where an alternative method is available and recommended. For example, you should use font SWIs for font painting rather than the VDU sequences that do so.

The output calls in detail

The following sections contain tables giving more detail on how the printer driver treats calls passing through the software vectors it claims, and how it interacts with the font manager. Some of the tables are followed by extra information on the more complex calls; unless otherwise stated, you should not take this to mean that such calls are recommended over and above all other supported ones.

ByteV

The printer drivers pass on the vast majority of calls made through ByteV; they are interpreted as usual by the ROM's OS_Byte routine. The printer drivers claim and process only the following calls:

Call processed	Meaning	Notes
OS_Byte 135	Read character at text cursor and screen mode	Use only to read 'screen' mode; returned character may be invalid
OS_Byte 163	Read/write general graphics information	Use only to set dot-dash length (ie R1 = 242, R2 ≤ 64)
OS_Byte 218	Read/write bytes in VDU queue	No restrictions

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, but some are processed by the printer driver:

ColourTrans SWI	Meaning	Printer driver's treatment
Colour Number To GCOL	Translate a colour number to a GCOL	Passed on
Convert CIE To RGB	Convert industry standard CIE colours to RISC OS RGB colours	Passed on
Convert CMYK To RGB	Convert from the CMYK model to RISC OS RGB colours	Passed on
Convert Device Colour	Convert a device colour to a standard colour	Passed on
Convert Device Palette	Convert a device palette to standard colours	Passed on
Convert HSV To RGB	Convert hue, saturation and value into corresponding RISC OS RGB colours	Passed on
Convert RGB To CIE	Convert RISC OS RGB colours to industry standard CIE colours	Passed on
Convert RGB To CMYK	Convert RISC OS RGB colours into the CMYK model	Passed on
Convert RGB To HSV	Convert RISC OS RGB colours into corresponding hue, saturation and value	Passed on
GCOL To Colour Number	Translate a GCOL to a colour number	Passed on
Generate Table	Set up a translation table in a buffer	Processed if R2 = –1 (ie table is for current mode); passed on otherwise
Invalidate Cache	Inform ColourTrans that the palette has been changed by some other means	Passed on
Misc Op	For internal use only	Passed on

ColourTrans SWI	Meaning	Printer driver's treatment
Select GCOL Table	Set up a list of GCOLs in a buffer	Passed on
Select Table	Set up a translation table in a buffer	Processed if R2 = –1 (ie table is for current mode); passed on otherwise
Set Calibration	Set the calibration table for the screen	Passed on
Set Colour	Change the foreground or background colour to a GCOL number	Passed on
Set Font Colours	Set the best range of anti-alias colours to match a pair of palette entries	Processed
Set GCOL	Set the closest GCOL for a palette entry	Processed
Set Opp GCOL	Set the furthest GCOL for a palette entry	Processed
Set Opp Text Colour	Change the text foreground or background colour to a GCOL number	Passed on
Set Text Colour	Change the text foreground or background colour to a GCOL number	Passed on
Read Calibration	Read the calibration table for the screen	Passed on
Read Palette	Read either the screen's palette, or a sprite's palette	Passed on
Return Colour Number	Get the closest colour for a palette entry	Processed
Return Colour Number For Mode	Get the closest colour for a palette entry	Processed if R1 = –1 (ie colour is for current mode); passed on otherwise
Return Font Colours	Find the best range of anti-alias colours to match a pair of palette entries	Passed on
Return GCOL	Get the closest GCOL for a palette entry	Passed on

ColourTrans SWI	Meaning	Printer driver's treatment
Return GCOL For Mode	Get the closest GCOL for a palette entry	Passed on
Return Opp Colour Number	Get the furthest colour for a palette entry	Processed
Return Opp Colour Number For Mode	Get the furthest colour for a palette entry	Processed if R1 = –1 (ie colour is for current mode); passed on otherwise
Return Opp GCOL	Get the furthest GCOL for a palette entry	Passed on
Return Opp GCOL For Mode	Get the furthest GCOL for a palette entry	Passed on
Write Calibration To File	Save the current calibration to a file	Passed on
Write Loadings To File	Write a * Command to a file that will set the ColourTrans error loadings	Passed on
Write Palette	Write to either the screen's palette, or to a sprite's palette	Passed on

ColourTrans_ReturnColourNumber
ColourTrans_ReturnColourNumberForMode with R1 = –1

Both these calls are treated in the same way by the printer drivers. They 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_ReturnOppColourNumber
ColourTrans_ReturnOppColourNumberForMode with R1 = –1

These calls behave exactly as though ColourTrans_ReturnColourNumber had been called with R0 containing the furthest possible RGB combination from the one actually specified.

This results in a subtle difference between the ‘opposite’ colours returned by the printer driver, and those normally returned by the ColourTrans module. The printer driver 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_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 (see above). The resulting values are placed in the table.

ColourTrans_SetFontColours

This call sets the printer driver’s version of the font colours, to as accurate a representation of the desired RGB values as the printer can manage. Along with control (19) sequences within the string to be painted – which themselves use this call – it is the recommended way to set font colours.

As with other ColourTrans calls, the returned values are obtained by calling the ColourTrans module; in this case before the printer driver’s own colours are actually set. Just as with the above calls, you should not subsequently use these values to set printing colours.

ColourTrans_SetGCOL

This call sets the printer driver's version of the foreground or background colour, as appropriate. It is the recommended way to do so.

The *gcol_action* passed in R4 is interpreted as follows:

- If *gcol_action* MOD 8 ≠ 0, subsequent plots and sprite plots will not do anything.
- If *gcol_action* = 0, the RGB value in R0 is remembered by the printer driver and used for subsequent plots. Plotting is done by overwriting with the closest approximation to this RGB value that the printer can render. Subsequent sprite plotting will be done without using the sprite's mask.
- If *gcol_action* = 8, subsequent plots will be treated the same as R4 = 0 above, except that sprite plots will be done using the sprite's mask, if any.
- If *gcol_action* > 8, and *gcol_action* MOD 8 = 0, an unspecified solid colour will be used.

This call never uses ECFs; the flag which sets whether or not to use ECFs (bit 8 of R3) is ignored.

After this has been done, the call is effectively converted into a call to ColourTrans_ReturnGCOL, and is passed down to the ColourTrans module in order to set the information returned. Note that the returned GCOL is therefore the closest GCOL available from the current screen palette, which may considerably differ from the passed RGB value. You should therefore not subsequently use the returned value to set colours for printing.

ColourTrans_SetOppGCOL

This behaves like ColourTrans_SetGCOL above, except that the RGB value the printer driver remembers is the furthest possible RGB value from the one actually specified in R0, and the returned values are given by converting this call into a call to ColourTrans_ReturnOppGCOL. Again, you should not subsequently use the returned value to set printing colours, as they are dependent on the current screen palette.

ColourTrans_SetTextColour
ColourTrans_SelectOppTextColour

You should not use these calls to set text colours when printing, as the printer drivers ignore text colours. You should instead use the font manager to print coloured text; if necessary, you can use the outline System font introduced in RISC OS 3.

DrawV

Printer drivers intercept Draw SWIs via the DrawV vector. Those calls that normally plot to the screen are intercepted and processed by the printer driver to generate printer output. There are a number of restrictions on these calls, mainly due to the limitations of PostScript. Fortunately most of the operations that are disallowed are not particularly useful.

All other calls to DrawV are passed on to the Draw module and treated in the same way as usual.

Floating point calls

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.

Treatment of Draw SWIs

The table below summarises the printer driver's treatment of each integer Draw SWI. It is followed by more detailed notes of the restriction on each of the calls processed by the printer driver:

Draw SWI	Meaning	Printer driver's treatment
Fill	Process and output a path, filling the interior portion	Processed, but with restrictions; see below for notes
Flatten Path	Convert an input path into a flattened output path	Passed on
Process Path	Multi-purpose main Draw SWI	Faulted if R7 = 1 or 2; processed otherwise, but with restrictions; see below for notes
Stroke	Process and output a path	Processed, but with restrictions; see below for notes
Stroke Path	Process a path, writing output to a path	Passed on
Transform Path	Convert an input path	Passed on

Draw_ProcessPath

This call 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 some printer drivers, such as the PostScript one.

If you are using this call to calculate bounding boxes, using the R7 = &80000000+ address option, then the parameters such as the matrix, flatness and line thickness must exactly correspond with those in the intended call. If they differ, then rounding errors, flattening errors and the like may cause clipping.

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.

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, all the restrictions mentioned above for Draw_Fill also apply to this call. Further restrictions are:

- Printer drivers 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.

Font manager SWIs

The printer driver interacts with the font manager via a service call and the SWI PDriver_FontSWI in such a way that when it is active, calls to some SWIs are passed to the printer driver, which may then process the SWI or ignore it. The table below shows how each SWI is handled. Note that some of the SWIs listed as being processed by the font manager may cause a Lose Font operation, which is passed to the printer driver:

Font SWI	Meaning	Processing
Cache Addr	Get the version number, font cache size and amount used	Processed by font manager as usual
Caret	Define text cursor	Passed to printer driver, but ignored by it
Char BBox	Get the bounding box of a character	Processed by font manager as usual
Convert to OS	Convert internal coordinates to OS coordinates	Processed by font manager as usual
Convert to Points	Convert OS coordinates to internal coordinates	Processed by font manager as usual
Current Font	Get current font handle and colours	Processed by font manager as usual
Current RGB	Read the settings of colours after calling Font_Paint	Processed by font manager as usual
Decode Menu	Decode a selection made from a font menu	Processed by font manager as usual
Find Caret	Find where the caret is in the string	Processed by font manager as usual
Find Caret J	Find where the caret is in a justified string	Processed by font manager as usual
Find Font	Get the handle for a font	Passed to printer driver and processed by it

Font SWI	Meaning	Processing
Future Font	Check font characteristics after Font_StringWidth	Processed by font manager as usual
Future RGB	Read the settings of colours after calling various Font_… SWIs	Processed by font manager as usual
List Fonts	Scan for fonts, returning their names one at a time; or build a menu of fonts	Processed by font manager as usual
Lose Font	Finish use of a font	Passed to printer driver and processed by it
Make Bitmap	Make a font bitmap file	Processed by font manager as usual
Paint	Output a string	Passed to printer driver and processed by it
Read Colour Table	Read the anti-alias colour table	Processed by font manager as usual
Read Defn	Read details about a font	Processed by font manager as usual
Read Encoding Filename	Return the filename of the encoding file used for a given font handle	Processed by font manager as usual
Read Font Max	Read the FontMax values	Processed by font manager as usual
Read Font Metrics	Reads the metrics information held in a font's IntMetrics file	Processed by font manager as usual
Read Font Prefix	Find the directory prefix for a given font handle	Processed by font manager as usual
Read Info	Get the font bounding box	Processed by font manager as usual
Read Scale Factor	Read the internal to OS conversion factor	Processed by font manager as usual
Read Thresholds	Read the list of threshold values for printing	Processed by font manager as usual
Scan String	Return information on a string	Processed by font manager as usual
Set Colour Table	For internal use by the ColourTrans module only	Passed to printer driver, but ignored by it
Set Font	Select the font to be subsequently used	Passed to printer driver and processed by it

Font SWI	Meaning	Processing
Set Font Colours	Change the current colours and (optionally) the current font	Passed to printer driver and processed by it; however, you should use ColourTrans_ SetFontColours to set font colours
Set Font Max	Set the FontMax values	Processed by font manager as usual
Set Palette	Define the anti-alias palette	Passed to printer driver, but ignored by it
Set Scale Factor	Set the internal to OS conversion factor	Processed by font manager as usual
Set Thresholds	Define the list of threshold values for printing	Processed by font manager as usual
String BBox	Get the bounding box of a string	Processed by font manager as usual
String Width	Calculate how wide a string would be	Processed by font manager as usual
Switch Output To Buffer	Switch output to a buffer, creating a Draw file structure	Processed by font manager as usual
UnCache File	Delete uncached font information, or recache it	Processed by font manager as usual

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, set font colours to absolute RGB values using ColourTrans_SetFontColours or control (19) sequences within the string to be painted. 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.

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 processed by the printer driver to generate printer output.

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 using the screen mode in effect at the start of the print job. If it is printed scaled, the scaling factors are applied to this size. It is done this way in the expectation that the application is scaling the sprite for what it believes is the current screen mode.

Colours

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.
- If a translation table is supplied with the call, the printer driver assumes that the table contains code values allocated by ColourTrans_SelectTable with R2 = –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.

The latter method is strongly recommended over the former. As usual when printing, if you don't use ColourTrans calls, you will get unpredictable results that are dependent on the current screen palette.

Treatment of SpriteOp reason codes

The table below shows the printer driver's treatment of each SpriteOp reason code:

Reason code	Meaning	Printer driver's treatment
2	Screen save	Faulted: unknown what 'screen' may be
3	Screen load	Faulted: unknown what 'screen' may be
8	Read area control block	Passed on

Reason code	Meaning	Printer driver's treatment
9	Initialise sprite area	Passed on
10	Load sprite file	Passed on
11	Merge sprite file	Passed on
12	Save sprite file	Passed on
13	Return name	Passed on
14	Get sprite	Faulted: unknown what 'screen' may be
15	Create sprite	Passed on
16	Get sprite from user coordinates	Faulted: unknown what 'screen' may be
24	Select sprite	Passed on for user sprite (ie when &100 or &200 added to reason code); faulted for system sprite (ie when reason code is 24); see note below
25	Delete sprite	Passed on
26	Rename sprite	Passed on
27	Copy sprite	Passed on
28	Put sprite	Processed
29	Create mask	Passed on
30	Remove mask	Passed on
31	Insert row	Passed on
32	Delete row	Passed on
33	Flip about x axis	Passed on
34	Put sprite at user coordinates	Processed
35	Append sprite	Passed on
36	Set pointer shape	Passed on
37	Create/remove palette	Passed on
40	Read sprite information	Passed on
41	Read pixel colour	Passed on
42	Write pixel colour	Passed on

Reason code	Meaning	Printer driver's treatment
43	Read pixel mask	Passed on
44	Write pixel mask	Passed on
45	Insert column	Passed on
46	Delete column	Passed on
47	Flip about y axis	Passed on
48	Plot sprite mask	Processed
49	Plot mask at user coordinates	Processed
50	Plot mask scaled	Processed
51	Paint character scaled	Faulted; you should instead pass the character to WrchV using OS_WriteC (or a derivative)
52	Put sprite scaled	Processed
53	Put sprite grey scaled	Processed, but normally treated as reason code 52, because grey-level anti-aliasing can be unpredictable and is hard to support
54	Remove lefthand wastage	Passed on
55	Plot mask transformed	Processed
56	Put sprite transformed	Processed
57	Insert/delete rows	Passed on
58	Insert/delete columns	Passed on
60	Switch output to sprite	Passed on; applications can still create sprites whilst printing
61	Switch output to mask	Passed on; applications can still create masks whilst printing
62	Read save area size	Passed on

OS_SpriteOp 24

A call to SpriteV with reason code 24 is passed through to the operating system if it is for a user sprite (ie when &100 or &200 is added to the reason code), as the call is simply asking the operating system for the address of the sprite concerned. If the call is

for a system sprite (ie nothing has been added to the reason code), it is faulted, because it is asking for a sprite to be selected for use with the VDU 25,232-239 sprite plotting sequences, which are themselves not supported by printer drivers.

WrchV

The printer driver queues all characters sent through WrchV in the same way as the VDU drivers do, processing complete character sequences as they appear.

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. Consequently, you should not select a print job if there is an incomplete sequence in the VDU queue. Also, the output stream specification set by OS_Byte 3 should be in its standard state – as though set by OS_Byte 3,0. Finally, the printing application should not use any output calls whilst partway through sending a VDU sequence, as the two may clash.

Internal graphics state of printer driver

When plotting starts in a rectangle supplied by a printer driver, the printer driver behaves as though the VDU system were in the following state:

- VDU drivers are enabled.
- VDU 5 state is 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, with a possible slight difference caused by rounding errors when converting the coordinates to OS units. However, 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.
- 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 has a nominal size on the paper of $^1/_{180}$ inch, depending on the transformation supplied with this rectangle.
- A pixel has a nominal size on the paper of $^1/_{90}$ inch square (ie 2 OS units square); thus all PLOT line, PLOT point and PLOT outline calls produce lines that are approximately $^1/_{90}$ inch wide.

- No text coordinate system is defined.

This is designed to be as similar as possible to the state set up by the window manager when redrawing.

The printer driver maintains its own state, and calls that it processes alter this state rather than that of the screen. If WrchV is called but the printer driver does not currently want a rectangle printed, it will keep track of the state – for example, the current foreground and background colours – but will not produce any printer output.

Rounding

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.

Treatment of character sequences

The table below shows the number of extra bytes needed to complete each character sequence, and whether the printer driver claims and processes the sequence, claims and faults it, claims and ignores it, or passes it on to the VDU drivers. It gives further information for most sequences, or a reference to a longer note following the table:

Character sequence	Extra bytes	Meaning	Printer driver's treatment
0	0	Do nothing	Ignored
1	1	Send character to printer only	Faulted: would probably disrupt printing
2	0	Enable printer	Faulted: would probably disrupt printing
3	0	Disable printer	Ignored: would probably disrupt printing
4	0	Write text at text cursor	Faulted: text printing always uses graphics cursor
5	0	Write text at graphics cursor	Ignored: text printing always uses graphics cursor anyway
6	0	Enable processing of character sequences	Processed: reverses effect of character 21
7	0	Generate bell sound	Passed on
8	0	Move cursor back one character	Processed: always moves graphics cursor
9	0	Move cursor on one character	Processed: always moves graphics cursor

Character sequence	Extra bytes	Meaning	Printer driver's treatment
10	0	Move cursor down one line	Processed: always moves graphics cursor
11	0	Move cursor up one line	Processed: always moves graphics cursor
12	0	Clear window	Processed: always clears graphics window
13	0	Move cursor to start of current line	Processed: always moves graphics cursor
14	0	Turn on page mode	Ignored: meaningless when printing
15	0	Turn off page mode	Ignored: meaningless when printing
16	0	Clear graphics window	Processed
17	1	Define text colour	Ignored: text colour is unused in graphics printing
18	2	Define graphics colour	Processed: see note below
19	5	Define logical colour	Passed on: affects screen hardware
20	0	Restore default logical colours	Passed on: affects screen hardware
21	0	Disable VDU drivers	Processed: printer driver parses but does not process subsequent character sequences, until it receives character 6
22	1	Select screen mode	Faulted: cannot change the 'mode' of a printed page
23,0	8	Set the interlace or the cursor appearance	Passed on: affects screen hardware
23,1	8	Control the appearance of the cursor	Passed on: affects screen hardware
23,2-5	8	Define ECF pattern and colours	Passed on: affects global resources
23,6	8	Set dot-dash line style	Ignored: use Draw SWIs for dotted lines
23,7	8	Scroll text window or screen	Faulted: text printing always uses graphics window
23,8	8	Clear a block of the text window	Faulted: text printing always uses graphics window
23,9	8	Set flash time for first flashing colour	Passed on: affects screen hardware

Character sequence	Extra bytes	Meaning	Printer driver's treatment
23,10	8	Set flash time for second flashing colour	Passed on: affects screen hardware
23,11	8	Set default patterns	Passed on: affects global resources
23,12-15	8	Define ECF patterns and colours	Passed on: affects global resources
23,16	8	Control movement of cursor	Processed: bit 6 of the flags is ignored, and treated as if set
23,17,0-1	7	Set tint for text colours	Ignored: text printing always uses graphics colours
23,17,2-3	7	Set tint for graphics colours	Processed
23,17,4	7	Choose colour patterns used	Passed on: affects global resources
23,17,5	7	Exchange text foreground and background colours	Ignored: text printing always uses graphics colours
23,17,6	7	Set ECF origin	Passed on: affects global resources
23,17,7	7	Set character size/spacing	Processed: uses screen pixel size for the screen mode that was in effect when the print job was started
23,18-24	8	Reserved	Faulted: reserved, so meaning not known by printer driver
23,25-26	8	Obsolete font calls provided for compatibility	Faulted: obsolete
23,27	8	Obsolete sprite call provided for compatibility	Faulted: obsolete
23,28-31	8	Reserved	Faulted: reserved, so meaning not known by printer driver
23,32-255	8	Redefine printable characters	Passed on: affects global resource
24	8	Define graphics window	Processed: window must lie entirely within rectangle currently being printed, or unpredictable results will occur

Character sequence	Extra bytes	Meaning	Printer driver's treatment
25,0-15	4	Plot solid line	Processed
25,16-31	4	Plot dotted line	Processed: plots solid line (use Draw_Stroke to get dotted lines)
25,32-47	4	Plot solid line	Processed
25,48-63	4	Plot dotted line	Processed: plots solid line (use Draw_Stroke to get dotted lines)
25,64-71	4	Plot point	Processed
25,72-79	4	Horizontal line fill	Faulted: cannot be implemented on some printers
25,80-87	4	Triangle fill	Processed
25,88-95	4	Horizontal line fill	Faulted: cannot be implemented on some printers
25,96-103	4	Rectangle fill	Processed
25,104-111	4	Horizontal line fill	Faulted: cannot be implemented on some printers
25,112-119	4	Parallelogram fill	Processed
25,120-127	4	Horizontal line fill	Faulted: cannot be implemented on some printers
25,128-143	4	Flood fill	Faulted: cannot be implemented on some printers
25,144-151	4	Circle outline	Processed
25,152-159	4	Circle fill	Processed
25,160-167	4	Circular arc	Processed
25,168-175	4	Segment	Processed
25,176-183	4	Sector	Processed
25,184	4	Move relative	Processed: equivalent to 25,0
25,185-187	4	Relative rectangle move/copy	Faulted: dependent on current picture contents
25,188	4	Move absolute	Processed: equivalent to 25,4
25,198-191	4	Absolute rectangle move/copy	Faulted: dependent on current picture contents

Character sequence	Extra bytes	Meaning	Printer driver's treatment
25,192-199	4	Ellipse outline	Processed
25,200-207	4	Ellipse fill	Processed
25,208-215	4	Font plot	Faulted
25,216-231	4	Reserved	Faulted: reserved, so meaning not known by printer driver
25,232-239	4	Sprite plot	Faulted
25,240-255	4	Reserved	Faulted: reserved, so meaning not known by printer driver
26	0	Restore default windows	Processed: resets graphics window to maximum size (ie rectangle being printed)
27	0	Do nothing	Ignored
28	4	Define text window	Ignored: text printing always uses graphics window
29	4	Define graphics origin	Processed
30	0	Home cursor	Processed: always moves graphics cursor
31	2	Move cursor	Processed: always moves graphics cursor
32-126	0	Output printable character	Processed: outputs character in system font (use Font SWIs for font printing)
127	0	Delete	Processed
128-255	0	Output printable character	Processed: outputs character in system font (use Font SWIs for font printing)

VDU 18 and GCOLs

You should only use the GCOL sequence (VDU 18,*gcol_action*,*colour*) if absolutely necessary, and you should be aware of the fact that the printer driver has a simplified interpretation of the parameters, as follows:

- As usual the background colour is affected if *colour* ≥ 128, and the foreground colour if *colour* < 128.
- The *gcol_action* is treated in the same way as for ColourTrans_SetGCOL; see page 3-580.

We strongly recommend that applications should use ColourTrans calls 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.

Miscellaneous SWIs

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.

Processor flags

Note that processor flags may have different values on exit from some calls when a printer driver is active to those that they would otherwise have.

Error handling changes

This section describes a couple of somewhat unusual features of the printer drivers' error handling that an application author should be aware of. Before reading this, you should have read the section entitled *Error handling whilst printing* on page 3-571.

Escape handling

Firstly, 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.

Escape generation is turned on permanently for these SWIs:

- PDriver_SelectJob for a new job
- PDriver_EndJob.

When the printer driver is intercepting plotting calls (ie there is an active print job, and plotting output is directed either to the screen or to a sprite internal to the printer driver, and the Wimp is not reporting an error – as defined by ServiceCall_WimpReportError) escape generation is also enabled for these calls:

- PDriver_DrawPage
- PDriver_GetRectangle
- OS_WriteC and its derivatives – ie all SWIs that call WrchV
- All ColourTrans SWIs – except ColourTrans_ColourNumberToGCOL, ColourTrans_GCOLToColourNumber, ColourTrans_InvalidateCache, ColourTrans_MiscOp, ColourTrans_SetOppTextColour and ColourTrans_SetTextColour.
- Draw_Fill
- Draw_Stroke
- Font_SetFontColours
- Font_SetPalette
- Font_Paint
- OS_SpriteOp with reason codes 28 (put sprite), 34 (put sprite at user coordinates), 48 (plot mask), 49 (plot mask at user coordinates), 50 (plot mask scaled) 52 (put sprite scaled), 53 (put sprite grey scaled), 55 (plot mask transformed), or 56 (put sprite transformed).

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 that the application is not expecting from being ignored. For example, quite a lot of code assumes incorrectly that OS_WriteC cannot produce an error; generating a persistent error ensures that it cannot reasonably get ignored forever.

Persistent errors are created at all times for these SWIs:

- PDriver_EndJob
- PDriver_GiveRectangle
- PDriver_DrawPage
- PDriver_GetRectangle

When the printer driver is intercepting plotting calls, the following SWIs also generate persistent errors:

- OS_WriteC and its derivatives – ie all SWIs that call WrchV
- All ColourTrans SWIs – except ColourTrans_ColourNumberToGCOL, ColourTrans_GCOLToColourNumber, ColourTrans_InvalidateCache, ColourTrans_MiscOp, ColourTrans_SetOppTextColour and ColourTrans_SetTextColour.
- Draw_Fill
- Draw_Stroke
- Draw_ProcessPath with R7=1
- Font_SetFontColours
- Font_SetPalette
- Font_Paint
- OS_SpriteOp with reason codes 28 (put sprite), 34 (put sprite at user coordinates), 48 (plot mask), 49 (plot mask at user coordinates), 50 (plot mask scaled) 52 (put sprite scaled), 53 (put sprite grey scaled), 55 (plot mask transformed), or 56 (put sprite transformed)

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.

Technical Details

The structure of the printing system

The Printers application

The Printers application is the user interface to the printing system. It is split into two parts.

The ‘front end’

The ‘front end’ of the Printers application contains those parts of the code that are device independent. This includes support for such things as the text printing queue, reading printer definition files.

The ‘back ends’

The Printers application has several ‘back ends’, each of which contains code that is specific to a particular class of device. RISC OS 3 provides back ends for PostScript printers, LaserJet printers, and dot-matrix printers.

A back end implements such things as handling the printer configuration window for that class of printer, passing the information from that window to the printer drivers, and printing fancy text files. New back ends can be added to support new classes of printers, faxes, and so on.

Printer definition files

Printer definition files are supplied with the other RISC OS applications in a directory named Printers. Inside this directory there is a Top_Left file used to calibrate the position of output on Epson and IBM compatible dot matrix printers, a subdirectory for each supported printer manufacturer (eg Epson, Star, Apple, etc), and a Read_Me file giving extra help.

Inside each of the subdirectories there are some printer definition files (eg FX-80) and a further Read_Me file. These Read_Me files give some technical detail on what is in each definition file, paying attention to tricky areas to guide you should you either want to modify the files, or want to choose the most appropriate starting point to provide support for a new printer. They also give some guidance on which files to try for ‘compatible’ printers.

You will find further help in the chapter entitled *Printer definition files* on page 3-709.

The PrintEdit application

The PrintEdit application is used to edit dot matrix and LaserJet printer definition files. You can use this application to provide support for a new printer by editing the printer definition file of an already supported printer that has similar behaviour.

The FontPrint application

The FontPrint application is used to edit the list of supported fonts in the Printer application's configuration for the current PostScript printer. With it you can specify mappings between RISC OS font names and the printer's native font names, and the encodings that those fonts use. You can also specify whether a font is resident in the printer, or to be supplied by downloading it from RISC OS.

The PDriver module

The PDriver module – also known as the PDriver sharer module – allows you to have multiple resident printer drivers, and hence easily use different devices such as a dot-matrix printer, a PostScript printer and a Fax card during the same session. The module is responsible for tracking which printer drivers are loaded, and which of these is the 'current' printer driver. It is also responsible for handling printer jobs, and tracking which printer driver owns which jobs.

When the PDriver sharer module starts up, it issues the service call Service_PDriverStarting (see page 3-608). Any printer drivers resident at that time should declare themselves to the PDriver sharer module by calling PDriver_DeclareDriver (see page 3-650).

The PDriver module's SWI handling

The PDriver sharer module receives all printer driver SWIs. It processes some of these SWIs by itself, and some in conjunction with a printer driver, but passes on the majority to the current printer driver (set using PDriver_SelectDriver – see page 3-653).

Most of the SWIs that the PDriver sharer module processes in whole or in part by itself relate to job handling:

- The PDriver sharer module processes the SWIs PDriver_CurrentJob (page 3-624) and PDriver_EnumerateJobs (page 3-640). These just require some inspection of its internal job management structures, and no interaction with the real drivers.

- The PDriver sharer module needs the cooperation of the printer drivers to process the PDriver_SelectJob (page 3-622), PDriver_SelectIllustration (page 3-644), PDriver_AbortJob (page 3-628), PDriver_EndJob (page 3-626) and PDriver_Reset (page 3-630) job handling SWIs.

  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.

  Ending and Aborting are easily handled: they just clear the internal data for the specified job, and then pass through to the real driver. Resetting is similarly easy; it just requires that **all** drivers be reset.

- The PDriver sharer module does not process PDriver_CancelJob (page 3-638) nor PDriver_CancelJobWithError (page 3-642). 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 PDriver sharer module itself.

The printer drivers

Each printer driver handles a particular class of output device. They are mainly responsible for producing the device-dependent output necessary to print a page. They receive SWI calls via a handler registered with the PDriver sharer module when they declare themselves using PDriver_DeclareDriver; see page 3-650 for details of this mechanism.

RISC OS 3 provides printer drivers for PostScript printers (PDriverPS), and for bit image devices such as dot-matrix printers (PDriverDP).

PDriverDP

PDriverDP differs from PDriverPS in that it is further subdivided, to cope with the wide range of available bit image devices.

The PDriverDP module itself handles the device-independent parts of printing, rendering print jobs into bit image strips. These strips are then passed to a PDumper module (or printer dumper).

Printer dumpers

Printer dumpers provide the actual device driving for a particular type of bit image printer, outputting the bit image in an appropriate manner. In RISC OS 3 (version 3.00) they are also responsible for performing colour matching, error diffusion and halftoning; the same code is duplicated in each printer dumper.

RISC OS 3 provides printer dumpers for LaserJets (PDumperLJ), ImageWriters (PDumperIW) and Integrex/Epson printers (PDumperDM). PDumperDM is effectively two printer dumpers in one; they are combined to save ROM space.

Note that there is **not** a one to one correspondence between printer dumpers and back ends for the Printers application. For example, the dot-matrix back end caters for both PDumperIW and PDumperDM.

For more information on printer dumpers, see the chapter entitled *Printer Dumpers* on page 3-673.

PDumperSupport

In RISC OS 3 (version 3.10) the colour matching, error diffusion and halftoning has been separated from the printer dumpers and is provided by an additional module called PDumperSupport. It reads in printer dumper palette files to do so; these are held in the Printers application.

The PDumperSupport module saves ROM space, and allows the code to be used as a resource by third party printer dumpers, as it provides a SWI interface. Alternatively, third parties may choose not to use this module, but instead to perform colour matching in their own printer dumpers. Another option is to replace the PDumperSupport module to modify the colour matching performed by the Acorn printer dumpers.

Summary

The diagram below summarises the printing system in RISC OS 3 (version 3.10):

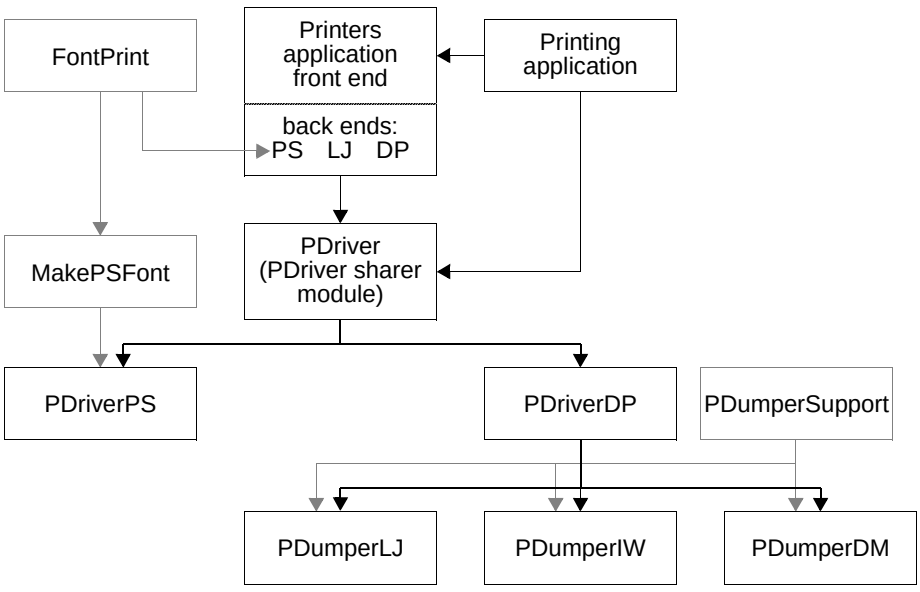


Figure 64.1 Structure of printing system in RISC OS 3 (version 3.10)

The RISC OS 2 printing system

The RISC OS 2 printing system was much simpler in structure, but at the expense of considerable duplication of code. Each printer driver had its own printing application. Only one such application could be loaded at once, hence there was no PDriver sharer module. Its job handling capabilities were supplied by each printer driver.

There were also no printer dumper modules, nor the PDumperSupport module. Each class of dot-matrix printer had its own printer driver. There was no support for ImageWriters.

If you were using a PostScript printer the structure was simply this:

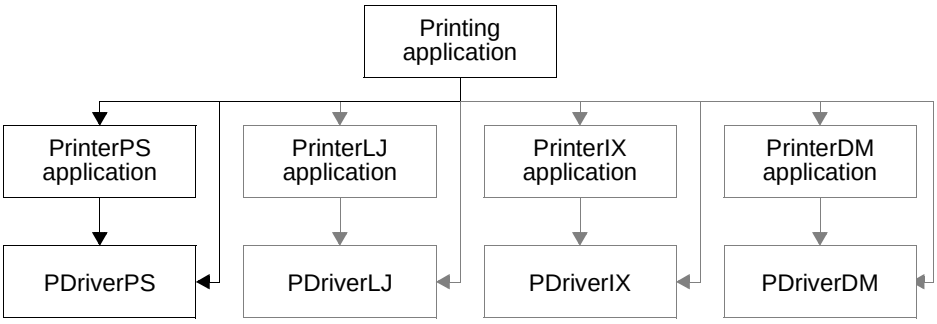


Figure 64.2 Structure of printing system in RISC OS 2

(The other parts of the printing system – shown above in grey – could not have been loaded at the same time as the PostScript system.)

Extending the printing system

There are a variety of ways in which you can extend the printing system, replacing one or more of its parts. Acorn is prepared to supply source code for the current system to developers wishing to do this; we strongly recommend that you follow this route for maximum compatibility with the existing system.

Adding printer definition files

You can add support for a new printer that closely matches an existing one by creating a new printer definition file (or more likely modifying an existing one) using the PrintEdit application. This may be for as trivial a reason as changing the name to match a particular printer or to make things more obvious to the user, or it may involve changing sequences to make things work better (or work at all) with a particular printer. Obviously you don’t need source code to do this.

Adding a printer dumper

You need to add a new printer dumper if the existing ones don't understand the output format of the printer you wish to support, or if you want to optimise things for your printers (such as printing on 20 out of 24 pins, or providing new colour matching).

A new printer dumper is virtually guaranteed to need a new printer definition file. Sometimes the PrintEdit application will be capable of creating this (eg 20 out of 24 pins); sometimes you will need to create them by hand, to modify PrintEdit, or to write your own tool. You might also need to supply a new palette file for your dumper.

Adding a back end to the Printers application

You will also need a new back end to the Printers application if your printer differs in major ways from those already supported: for example to add a direct drive laser printer, or a fax card. While these printers are still bit image – and thus can use a printer dumper – they need new text printing code, and (since they can't use the parallel port, for example) they need new printer connection management.

The LaserJet is a good example of this: it uses a printer dumper, yet it has its own back end since its text requirements differ from those of other dot-matrix printers.

Adding a printer driver

To add a radically different type of printer you will need to write a new printer driver. For example, a pen plotter or turtle graphics printer might need a new driver.

Adding a palette file and/or a new PDumperSupport module

You may wish to do this for an existing printer to modify the colour balance, whilst still using the Acorn drivers and printer dumpers.

Printer driver numbers

Printer drivers are identified by numbers, which are used as parameters to many of the PDriver SWIs. Currently assigned printer driver numbers are:

Value	Meaning
0	PostScript
1	Epson FX80 or compatible
2	HP LaserJet or compatible
3	Integrex ColourJet
4	FAX modem
5	Direct drive laser printer
6	Caspel graphics language
7	PDumper interface
99	Ace Computing Epson JX/Star LC10 driver or PaintJet driver

Using PostScript fonts

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 declare them using PDriver_DeclareFont; see the section entitled *Declare the fonts your document uses* on page 3-569, and the documentation of the SWI on page 3-648.

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, and/or by adding extra characters such as composite accented characters.
- Downloading an existing Type 1 PostScript version of the font on the fly.
- Generating and downloading a Type 3 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 by FontPrint 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.

Font names

A standard Adobe implementation of PostScript – such as that used on the Apple LaserWriter – has 35 fonts built in. Font names have been preallocated for RISC OS fonts that have the same metrics and general appearance as those fonts, and map onto them. This allows Acorn to produce a version of !PrinterPS that already knows the correct font name mappings. These names are:

RISC OS name	PostScript name
Churchill.Medium.Italic	ZapfChancery-MediumItalic
Clare.Medium	AvantGarde-Book
Clare.Medium.Oblique	AvantGarde-BookOblique
Clare.Demi	AvantGarde-Demi
Clare.Demi.Oblique	AvantGarde-DemiOblique
Corpus.Medium	Courier
Corpus.Medium.Oblique	Courier-Oblique
Corpus.Bold	Courier-Bold
Corpus.Bold.Oblique	Courier-BoldOblique
Homerton.Medium	Helvetica

Homerton.Medium.Oblique	Helvetica-Oblique
Homerton.Bold	Helvetica-Bold
Homerton.Bold.Oblique	Helvetica-BoldOblique
NewHall.Medium	NewCenturySchlbk-Roman
NewHall.Medium.Italic	NewCenturySchlbk-Italic
NewHall.Bold	NewCenturySchlbk-Bold
NewHall.Bold.Italic	NewCenturySchlbk-BoldItalic
Pembroke.Medium	Palatino-Roman
Pembroke.Medium.Italic	Palatino-Italic
Pembroke.Bold	Palatino-Bold
Pembroke.Bold.Italic	Palatino-BoldItalic
Robinson.Light	Bookman-Light
Robinson.Light.Italic	Bookman-LightItalic
Robinson.Demi	Bookman-Demi
Robinson.Demi.Italic	Bookman-DemiItalic
Selwyn	ZapfDingbats
Sidney	Symbol
Trinity.Medium	Times-Roman
Trinity.Medium.Italic	Times-Italic
Trinity.Bold	Times-Bold
Trinity.Bold.Italic	Times-BoldItalic

You can use T1ToFont to convert AFM (Adobe Font Metrics) files into IntMetrics files, and hence ensure that the correct metrics are used.

Service Calls

Service_Print (Service Call &41)

For internal use only

You must not use it in your own code.

Service_PDriverStarting (Service Call &65)

PDriver sharer module started

On entry

R1 = &65 (reason code)

On exit

All registers preserved

Use

This service call is issued when the PDriver sharer module starts up. Any printer drivers resident at that time should declare themselves to the PDriver sharer module by calling PDriver_DeclareDriver (see page 3-650).

Service_PDriverGetMessages (Service Call &78)

Get common messages file

On entry

R1 = &78 (reason code)

On exit

Not claimed

R0 - R8 must be preserved

Call claimed

R1 = 0 (implies service claimed)
R3 = pointer to 20 byte block for open messages file

Use

This service call is issued by a PDriver module that is about to open the common message file for printer drivers, held in Resources:$.Resources.PDrivers.Messages. It is provided so that the module can find if another PDriver module has already opened the file, and if so get its MessageTrans block:

- If the service call is claimed R3 will point to a 20 byte block. The first 16 bytes of this are a MessageTrans block referring to the file, and the remaining word is a usage count. The PDriver module should increment this usage count and use the MessageTrans block to access the file. When the module has finished using the file it should decrement the usage count, and if the count is ≤ 0 should then call MessageTrans_CloseFile (page 3-759) followed by OS_Module 7 (page 1-238) to free the 20 byte block.
- If the service call is not claimed the PDriver module should instead allocate 20 bytes using OS_Module 6 (page 1-237), and then use MessageTrans_OpenFile (page 3-752) to open Resources:$.Resources.PDrivers.Messages, placing the MessageTrans block in the first 16 bytes of the claimed buffer, and setting the usage count in the last word to 1.

A PDriver module receiving this service call that is using the common messages file should set R3 to point to the MessageTrans block and claim the service call by setting R1 to zero.

Service_PDriverChanged (Service Call &7F)

Currently selected printer driver has changed

On entry

R1 = &7F (reason code)
R2 = printer driver number of new driver (see page 3-604)

On exit

All registers are preserved

Use

This service call is issued when the PDriver sharer module has changed the currently selected printer driver. R2 contains the printer driver number being selected; see page 3-604 for a list of these.

This may be of use, for example, to a spooler module that needs to monitor which printer driver is currently selected.

SWI Calls

PDriver_Info (SWI &80140)

Get information on the printer driver

On entry

—

On exit

R0 = version number and type:
- bits 0 - 15 printer driver's version number × 100
- bits 16 - 31 printer driver number (see page 3-604)

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 = pointer to printer name, null terminated, maximum 20 characters long
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 (which is zero, unless it has been changed using PDriver_SetInfo)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This calls 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.

The values can be changed by the SWI PDriver_SetInfo, typically as a result of the user changing the printer configuration using the Printers application. If this call is made while a print job is selected, the values returned are those for that job (ie those in force when the job was first selected using PDriver_SelectJob). If this call is made 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 bottom 16 bits of R0 have the version number of the printer driver × 100: eg Version 3.21 would be 321 (&0141). The top 16 bits contain the printer driver number of the currently selected driver; see page 3-604 for a list of these.

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. For a complete description of the values bits 0 - 2 may have, see page 3-614:

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.

Bit(s)	Value	Meaning

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	it does not support transformed sprite plotting.
	1	it supports transformed sprite plotting.
12	0	it cannot handle new Font manager features.
	1	it 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 supplied to PDriver_DrawPage other than scalings, translations, rotations by multiples of 90 degrees and combinations thereof.
	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 PDriver_MiscOp call.
	1	it does support the PDriver_MiscOp call.
28	0	it does not support the PDriver_SetDriver call.
	1	it does support the PDriver_SetDriver call.
29	0	it does not support the PDriver_DeclareFont call.
	1	it does support the PDriver_DeclareFont call.

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 discrete 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 pointed to by R4 is always null terminated, 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.

The value in R7 – a printer driver specific number identifying the configured printer – is for internal use only.

Related SWIs

PDriver_SetInfo (page 3-615), PDriver_CheckFeatures (page 3-617)

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 = features word:
bit 0 set ⇒ colour, else monochrome
all other bits reserved (must be zero)
R4 = pointer to new printer name, null terminated, maximum 20 characters long
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 - 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 application on the desktop to configure a printer driver so that it is set up for a specific printer within the general class of printers the driver supports. 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.

Only bit 0 of the features word passed in R3 is used; all other bits are ignored.

The printer name in R4 is ignored by RISC OS 2.

The value in R7 – a printer driver specific number identifying the configured printer – is for internal use only.

This SWI must never be called by user applications.

Related SWIs

PDriver_Info (page 3-611)

Related vectors

None

PDriver_CheckFeatures (SWI &80142)

Check the features of a printer

On entry

R0 = mask of bits to check in features word
R1 = desired value of features word

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 it returns normally 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 (page 3-611)

Related vectors

None

PDriver_PageSize (SWI &80143)

Find how large the paper and print area is

On entry

—

On exit

R1 = x size of paper (including margins), in millipoints
R2 = y size of paper (including margins), in millipoints
R3 = left edge of printable area of paper, in millipoints from paper's left edge
R4 = bottom edge of printable area of paper, in millipoints from paper's bottom edge
R5 = right edge of printable area of paper, in millipoints from paper's left edge
R6 = top edge of printable area of paper, in millipoints from paper's bottom edge

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 size of the paper in use and its printable area. An application can use this information to decide how to place the data to be printed on the page.

The values can be changed by the SWI PDriver_SetPageSize, typically as a result of the user changing the printer configuration using the Printers application. If this call is made while a print job is selected, the values returned are those for that job (ie those in force when the job was first selected using PDriver_SelectJob). If this call is made 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 (page 3-620)

Related vectors

None

PDriver_SetPageSize (SWI &80144)

Set how large the paper and print area is

On entry

R1 = x size of paper (including margins), in millipoints
R2 = y size of paper (including margins), in millipoints
R3 = left edge of printable area of paper, in millipoints from paper's left edge
R4 = bottom edge of printable area of paper, in millipoints from paper's bottom edge
R5 = right edge of printable area of paper, in millipoints from paper's left edge
R6 = top edge of printable area of paper, in millipoints from paper's bottom edge

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

This call is used by the Printers application to set – for a particular driver – the paper size and printable area associated with subsequent print jobs. It must never be called by user applications.

All units are in millipoints, and R3 - R6 are relative to the bottom left corner of the page.

Related SWIs

PDriver_PageSize (page 3-618)

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 suspend current print job
R1 = pointer to a title string for the job, or zero if none

On exit

R0 = file handle for print job that was previously active, or zero if none
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 makes a given print job the current one. The job is identified by the handle of the file used for output from the job (eg printer:), which must be open for output.

The current print job (if any) is suspended, and a print job with the given file handle is selected. If a print job with this file handle already exists, it is resumed; otherwise a new print job is started. The printer driver starts to intercept plotting calls if it is not already doing so.

A file handle of zero has special meaning; the current print job (if any) is suspended, and the printer driver ceases to intercept plotting calls.

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 (page 3-624), PDriver_EndJob (page 3-626), PDriver_AbortJob (page 3-628), PDriver_Reset (page 3-630), PDriver_EnumerateJobs (page 3-640), PDriver_SelectIllustration (page 3-644)

Related vectors

None

PDriver_CurrentJob (SWI &80146)

Get the file handle of the current job

On entry

—

On exit

R0 = file handle for current job, or 0 if none

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call gets the file handle of the current job, returning it in R0. A value of zero is returned if no print job is active.

Related SWIs

PDriver_SelectJob (page 3-622), PDriver_EndJob (page 3-626), PDriver_AbortJob (page 3-628), PDriver_Reset (page 3-630), PDriver_EnumerateJobs (page 3-640), PDriver_SelectIllustration (page 3-644)

Related vectors

None

PDriver_FontSWI (SWI &80147)

This call is part of the internal interface between the font system and printer drivers. Applications must not call it.

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 call 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 (page 3-622), PDriver_CurrentJob (page 3-624), PDriver_AbortJob (page 3-628), PDriver_Reset (page 3-630), PDriver_CancelJob (page 3-638), PDriver_CancelJobWithError (page 3-642), PDriver_SelectIllustration (page 3-644)

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 call should be used to end a print job abnormally. It should be called immediately you get an error while printing, before you try to display the error message. 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 (page 3-622), PDriver_CurrentJob (page 3-624), PDriver_EndJob (page 3-626), PDriver_Reset (page 3-630), PDriver_CancelJob (page 3-638), PDriver_CancelJobWithError (page 3-642), PDriver_SelectIllustration (page 3-644)

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 all printer drivers, leaving the printer drivers 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 (page 3-622), PDriver_CurrentJob (page 3-624), PDriver_EndJob (page 3-626), PDriver_AbortJob (page 3-628), PDriver_SelectIllustration (page 3-644)

Related vectors

None

PDriver_GiveRectangle (SWI &8014B)

Specify a rectangle to be printed

On entry

R0 = rectangle identification word (specified by application)
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 (in millipoints)
R4 = background colour for this rectangle, in the form &BBGGRR00.

On exit

R0 - R4 preserved

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 4 word block pointed to by R1 contains the following:

Word	Contents
0	low x coordinate of rectangle to print, in OS units (inclusive)
1	low y coordinate of rectangle to print, in OS units (inclusive)
2	high x coordinate of rectangle to print, in OS units (exclusive)
3	high y coordinate of rectangle to print, in OS units (exclusive)

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:

$x' = (x \times R2!0 + y \times R2!8)/2^{16}$
$y' = (x \times R2!4 + y \times R2!12)/2^{16}$

(The rectangle and the transformation are very similar to Draw module rectangles and transformation matrices.)

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. We recommend that you initially try a margin of 1 point (2½ OS units), increasing this if results are not satisfactory.

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_DrawPage (page 3-635), PDriver_GetRectangle (page 3-637)

Related vectors

None

PDriver_DrawPage (SWI &8014C)

Called to draw the page after all rectangles specified

On entry

R0 = number of copies to print
R1 = pointer to 4 word block, to receive the rectangle to print (in OS units)
R2 = page sequence number within the document, or 0
R3 = pointer to a page number string, or 0

On exit

R0 = non-zero if more rectangles to be printed, zero if finished
R1 preserved
R2 = identification word for rectangle containing rectangle to print – if R0 ≠ 0
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 (if any) that the printer driver wants plotted in the area.

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 is non-zero on exit, 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.

Your application should stop trying to plot the current page if R0 = 0, and continue otherwise.

If R0 ≠ 0, it in fact gives the number of copies still to be printed, but this 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 4 word block pointed to by R1 contains the following on exit:

Word	Contents
0	low x coordinate of rectangle to print, in OS units (inclusive)
1	low y coordinate of rectangle to print, in OS units (inclusive)
2	high x coordinate of rectangle to print, in OS units (exclusive)
3	high y coordinate of rectangle to print, in OS units (exclusive)

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. For 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.

Related SWIs

PDriver_GiveRectangle (page 3-632), PDriver_GetRectangle (page 3-637)

Related vectors

None

PDriver_GetRectangle (SWI &8014D)

Get the next print rectangle

On entry

R1 = pointer to 4 word block, to receive the rectangle to print (in OS units)

On exit

R0 = non-zero if more rectangles to be printed, zero if finished
R1 preserved
R2 = identification word for rectangle containing rectangle to print – if R0 ≠ 0

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. See page 3-635 for further details

Related SWIs

PDriver_GiveRectangle (page 3-632), PDriver_DrawPage (page 3-635)

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, which must be done before giving any error message. See the section entitled *Error handling changes* on page 3-595.

Related SWIs

PDriver_EndJob (page 3-626), PDriver_AbortJob (page 3-628), PDriver_CancelJobWithError (page 3-642)

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 of R3 is set on exit from PDriver_Info), this call makes the printer driver output a screen dump to the file handle supplied in R0. The file concerned should already be open for output.

If the SWI is not supported, an error is returned.

Note that currently none of the Acorn printer drivers support this SWI.

Related SWIs

None

Related vectors

None

PDriver_EnumerateJobs (SWI &80150)

List existing print jobs

On entry

R0 = zero to enumerate first print job, or handle returned from previous call

On exit

R0 = next print job handle, or zero if no more

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call enumerates all the existing print jobs within the system, returning their job handles. The order in which they appear is undefined. To enumerate the complete list you should set R0 to zero and repeatedly call this SWI until R0 is returned as zero.

Related SWIs

PDriver_CurrentJob (page 3-624)

Related vectors

None

PDriver_SetPrinter (SWI &80151)

This call is used to set options specific to a particular printer driver. It is a private interface between the RISC OS 2 printing applications and the corresponding printer drivers. You must not use it.

This SWI has now been superseded by the SWI PDriver_SetDriver (page 3-667).

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

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 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, which must be done before giving any error message. See the section entitled *Error handling changes* on page 3-595.

This call is not available in RISC OS 2, unless version 2.00 or above of the printer driver module has been soft-loaded.

Related SWIs

PDriver_EndJob (page 3-626), PDriver_AbortJob (page 3-628), PDriver_CancelJob (page 3-638)

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
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 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. For example, the Draw application uses this to print.

This call is not available in RISC OS 2, unless version 2.00 or above of the printer driver module has been soft-loaded.

Related SWIs

PDriver_SelectJob (page 3-622), PDriver_CurrentJob (page 3-624), PDriver_EndJob (page 3-626), PDriver_AbortJob (page 3-628), PDriver_Reset (page 3-630), PDriver_CancelJob (page 3-638), PDriver_CancelJobWithError (page 3-642)

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

R0 - R7 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 26 of R3 is set on exit from PDriver_Info), it inserts an external file containing an illustration, such as an Encapsulated PostScript file, 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 call is not available in RISC OS 2, unless version 2.00 or above of the printer driver module has been soft-loaded.

Related SWIs

PDriver_SelectIllustration (page 3-644)

Related vectors

None

PDriver_DeclareFont (SWI &80155)

Declares the fonts that will be used in a document

On entry

R0 = handle of font to be declared, or zero
R1 = pointer to name of font to be declared, or zero
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
all other bits reserved (must be zero)

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 declares the fonts that will be used in a document, either by name or by handle. Certain printer drivers need this information before printing begins; for example, the PostScript driver needs it to perform font downloading, and to conform with structuring rules for PostScript documents. You should declare fonts after you have called PDriver_SelectJob (page 3-622) or PDriver_SelectIllustration (page 3-644) to start the print job.

Before calling PDriver_DeclareFont you must check if the printer driver you are using supports it by calling PDriver_Info (page 3-611) and examining bit 29 of R3 on return. If it is set you should declare each distinct font that your document uses by repeated calls

of PDriver_DeclareFont. For the purposes of this call, a font is ‘distinct’ if it differs in its name, encoding or matrix fields (\F, \E or \M; for details of font name syntax see the chapter entitled *The Font Manager* on page 3-411). You must not declare other variations in the font such as size, colour, etc. You may declare the font by its handle (passed in R0), or by a pointer to its name (R0 = 0, R1 = pointer to name). Any font name you pass must be exactly the same as is passed to Font_FindFont (see page 3-428), including any encoding and matrix information.

After you have declared all the fonts, you must make one further call with both R0 and R1 set to zero to signify the end of the list. If your document does not use any fonts you should still make this ‘end of list’ call; the printer driver then knows that your application is aware of this call, and will generate more efficient output.

The flags word gives other information about the font.

- Setting bit 0 stops a non-resident font being downloaded, in which case it will be substituted with a resident font, usually Courier (although this is driver specific). An example of appropriate use of this facility would be to set up a draft print option, so the correct font is used unless it can only be obtained at the expense of a slow download.
- Bit 1 is used to specify if kerning is applied to the font at any point in the job, so the PostScript printer driver knows whether or not it needs to download the font’s kerning information.

Once you have declared the fonts your application may then go on to make any PDriver_DrawPage request (page 3-635).

If this SWI is not called at all, the results are printer driver dependent. PDriverDP does not care in the least whether you call this SWI or not. On the other hand PDriverPS does care, and will perform default actions configured by the user, dependent on which fonts are already in the printer and which fonts need to be downloaded.

This call is not available under RISC OS 2.

Related SWIs

None

Related vectors

None

PDriver_DeclareDriver (SWI &80156)

Registers a printer driver with the PDriver sharer module

On entry

R0 = pointer to reason code handler for driver
R1 = pointer to driver's private word (to be passed in R12 when calling driver)
R2 = printer driver number (see page 3-604)

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 registers a printer driver with the PDriver sharer module. A driver should make this call when it is started, or when it receives Service_PDriverStarting (page 3-608). The driver can then be selected using PDriver_SelectDriver (page 3-653). Duplicate printer drivers are not allowed, and an error is generated if the driver is already registered.

You must register any new printer driver numbers with Acorn; see the section entitled *Printer driver and printer dumper numbers* on page 4-556.

The driver passes pointers to a reason code handler and to a private word (typically the driver's private workspace pointer). The driver's reason code handler provides entry points used by the sharer to implement PDriver_… SWIs. The sharer fully implements these SWIs itself:

PDriver_DeclareDriver
PDriver_RemoveDriver
PDriver_SelectDriver
PDriver_EnumerateDrivers

For all other SWIs, the sharer subtracts the PDriver SWI chunk base (&80140) from the SWI number to derive a reason code, and then calls the appropriate driver's reason code handler with the following register usage:

On entry

R11 = reason code (SWI number – &80140)
R12 = pointer to private word
R14 = return address

Other register usage as documented for corresponding SWI

On exit

V clear	register usage as documented for corresponding SWI
V set	R0 = pointer to error block

The handler should implement the functionality of the SWI, as documented.

This call is not available under RISC OS 2.

Related SWIs

PDriver_RemoveDriver (page 3-652)

Related vectors

None

PDriver_RemoveDriver (SWI &80157)

Deregisters a printer driver with the PDriver sharer module

On entry

R0 = printer driver number (see page 3-604)

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 deregisters a printer driver with the PDriver sharer module. This cancels all jobs associated with the driver. Doing so can get some applications confused – and possibly crash them, if they have pending jobs and believe the driver to still be present – so we strongly recommend that a driver checks that it has no pending jobs before calling this SWI.

This call is not available under RISC OS 2.

Related SWIs

PDriver_DeclareDriver (page 3-650)

Related vectors

None

PDriver_SelectDriver (SWI &80158)

Selects the specified driver

On entry

R0 = printer driver number (page 3-604), or –1 to set no current active driver,
or –2 to read current driver

On exit

R0 = previous driver number, or –1 if none

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 specified driver, returning an error if the driver has not been registered. This call is not designed for use by applications' authors, and should only be used by the Printer application.

If you **must** use this call, your code should store the previous driver number (returned in R0), and attempt to reselect it when finished.

This call is not available under RISC OS 2.

Related SWIs

PDriver_DeclareDriver (page 3-650), PDriver_RemoveDriver (page 3-652), PDriver_EnumerateDrivers (page 3-655)

Related vectors

None

PDriver_EnumerateDrivers (SWI &80159)

Enumerates all drivers within the system.

On entry

R0 = zero to enumerate first driver, or handle returned from previous call

On exit

R0 = handle to enumerate next driver, or zero if no more
R1 = printer driver number (page 3-604)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call enumerates all the drivers within the system, returning their printer driver numbers (a list of which is on page 3-604). To enumerate the complete list you should set R0 to zero and repeatedly call this SWI until R0 is returned as zero.

This call is not available under RISC OS 2.

Related SWIs

PDriver_SelectDriver (page 3-653)

Related vectors

None

PDriver_MiscOp (SWI &8015A)

Processes miscellaneous printer driver operations

On entry

R0 = reason code
Other registers are reason code dependent

On exit

Reason code dependent

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call processes miscellaneous printer driver operations. The action depends on the reason code passed in R0:

R0	Action	Page
0	Adds a font name to a list of those known to the current printer	3-658
1	Removes font name(s) from a list of those known to the current printer	3-660
2	Enumerates the font name(s) in a list of those known to the current printer	3-661
&80000000	Registers a printer dumper with PDriverDP	3-663

R0	Action	Page
&80000001	Deregisters a printer dumper with PDriverDP	3-664
&80000100 - &80000FFF	An extension mechanism to provide direct control over a printer dumper	3-665

Reason codes with bit 31 clear are applicable to all drivers, whereas those with bit 31 set are driver specific.

This call is not available under RISC OS 2.

Related SWIs

PDriver_MiscOpForDriver (page 3-666)

Related vectors

None

PDriver_MiscOp 0 (SWI &8015A)

Adds a font name to a list of those known to the current printer

On entry

R0 = 0 (reason code)
R1 = pointer to RISC OS font name (control-character terminated)
R2 = pointer to printer's native font name (control-character terminated),
or 0 if none
R3 = flag word for printer dependent code:
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 - 31 reserved (must be zero)
R4 = flag word for font addition:
bit 0 set ⇒ overwrite existing entries
bits 1 - 31 reserved (must be zero)

On exit

R0 - R4 preserved

Use

This call adds a font name to a list of those known to the current printer. It is used by the Printers application, and need not be called by other applications.

If no job is selected, the font name gets added to a global list which describes the fonts known to the printer. If a job is selected, the font name instead gets added to a local list – associated with the job – which describes the fonts and their mappings within that job. Each record is stored as a separate block within the RMA. When PDriver_SelectJob is called to start a job, the local list is initialised by copying the blocks in the current global list.

The RISC OS font name pointed to by R1 should ideally 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.

R2 contains a pointer to the printer's native font name to be associated with the RISC OS font name. This is case sensitive, and is used by the printer dependent code as required. You may pass a null name if necessary; for example direct drive laser printer drivers don't have native font names.

R3 is a flag word to be used by the printer dependent code; see specific printer documentation for further details.

R4 contains a flag word to associate with the addition of the record. Bit 0 controls what happens when you try to add a font name that has already been defined; if the bit is set, the old data gets overwritten, whereas if it is clear an error is generated.

PDriver_MiscOp 1 (SWI &8015A)

Removes font name(s) from a list of those known to the current printer

On entry

R0 = 1 (reason code)
R1 = pointer to RISC OS font name (control-character terminated), or zero to delete all fonts

On exit

R0, R1 preserved

Use

This call is used to remove font name(s) from a list of those known to the current printer. It is used by the Printers application, and need not be called by other applications.

R1 points to the font name to be removed, but if this pointer is zero, all font names get removed.

If no job is selected, the font name(s) get removed from the global list; if a job is selected, the font name(s) instead get removed from the local list. See PDriver_MiscOp 0 on page 3-658 for more details of how these lists are used.

No error is generated if you attempt to remove all font names but none are registered, whereas an error will be generated if you attempt to remove a specific font name that is not present.

Current versions of this call ignore R1, and always remove all fonts.

PDriver_MiscOp 2 (SWI &8015A)

Enumerates the font name(s) in a list of those known to the current printer

On entry

R0 = 2 (reason code)
R1 = pointer to return buffer, or zero to return required size of buffer
R2 = size of return buffer, or zero to return required size of buffer
R3 = zero to enumerate first font names, or handle returned from previous call,
or (if R1 = R2 = 0) header size to add to returned buffer size
R4 = flags:
all bits reserved (must be zero)

On exit

if R1 ≠ 0 on entry then:

R1 = pointer to first free byte in buffer
R2 = number of free bytes in buffer
R3 = handle to enumerate next font names, or zero if no more
R4 preserved

else:

R1 preserved
R2 = required size of buffer to return data + header size passed in R3
R3, R4 preserved

Use

This call enumerates the font name(s) in a list of those known to the current printer. It is used by the Printers application, and need not be called by other applications.

To enumerate the complete list you should set R3 to zero and repeatedly call this SWI until R3 is returned as zero

If no job is selected, the global list is enumerated; if a job is selected, the local list is instead enumerated. See PDriver_MiscOp 0 on page 3-658 for more details of how these lists are used.

The font names are returned as a series of three word records in the return buffer:

Offset	Meaning
0	pointer to RISC OS font name (control-character terminated)
4	pointer to native font name (control-character terminated)
8	flag word for printer dependent code (see PDriver_MiscOp 0)

The font names are stored in blocks within the RMA. Ideally you should make a copy of these, as someone could later remove them by calling PDriver_MiscOp 1.

Before enumerating the fonts you can find the required size of the return buffer by calling this SWI with R1 and R2 set to zero, and R3 set to the size of any header for which you wish to pre-allocate room. The required buffer size is returned in R2 (ie sufficient to hold all enumerated fonts, and the given size of header).

PDriver_MiscOp &80000000 (SWI &8015A)

Registers a printer dumper with PDriverDP

On entry

R0 = &80000000 (reason code)
R1 = number of printer dumper to register (see page 3-675)
R2 = version of PDriverDP required by dumper × 100
R3 = pointer to dumper's private word (to be passed in R12 when calling dumper)
R4 = pointer to reason code handler for dumper
R5 = supported calls bit mask
R6 = supported strip types bit mask

On exit

R0 - R6 preserved

Use

This call registers a printer dumper with PDriverDP. A dumper should make this call when it is started, or when it receives Service_PDumperStarting (page 3-686). We recommend you use the PDriver_MiscOpForDriver form (see page 3-666), as this ensures correct operation even if PDriverDP is not the currently selected driver. Duplicate printer dumpers are not allowed, and an error is generated if the dumper is already registered.

The dumper passes pointers to a reason code handler, and to a private word (typically the dumper's private workspace pointer). The dumper's reason code handler provides entry points used by PDriverDP to implement those parts of its functionality that are printer dependent, such as initialising a printer, or outputting a strip of an image.

The dumper also passes a bit mask in each of R5 and R6. If bit *n* of the mask is set, then it shows (respectively) that the printer dumper supports reason code *n* or that it can output strip type *n*. A dumper must support reason codes 0 - 7, and strip types 0 - 2; PDriverDP will assume that it does so. Bits corresponding to undefined reason codes or strip types must be zero.

For details of the current range of reason codes and strip types – and of the entry conditions for the handler – see the section entitled *Reason code handler entry and exit conditions* on page 3-674.

PDriver_MiscOp &80000001 (SWI &8015A)

Deregisters a printer dumper with PDriverDP

On entry

R0 = &80000001 (reason code)
R1 = number of printer dumper to deregister (see page 3-675)

On exit

R0, R1preserved

Use

This call deregisters a printer dumper with PDriverDP. A dumper should make this call when it dies. This call may return an error, especially if the dumper is currently being used for a print job, in which case the dumper must refuse to die, returning the original error. We recommend you use the PDriver_MiscOpForDriver form (see page 3-666), as this ensures correct operation even if PDriverDP is not the currently selected driver.

PDriver_MiscOp &80000100 - &80000FFF (SWI &8015A)

An extension mechanism to provide direct control over a printer dumper

On entry

R0 = &80000100 - &80000FFF (reason code)
R1 = number of printer dumper to process call (see page 3-675)
R2 - R7 are reason code dependent

On exit

Reason code dependent

Use

These calls are an extension mechanism to provide direct control over a printer dumper. Registers R0 - R7 are passed straight through to the specified dumper using the MiscOp entry point; the processing of these registers is dumper-specific.

All current Acorn printer dumpers do not use this feature, and merely return control immediately the MiscOp entry point is called.

PDriver_MiscOpForDriver (SWI &8015B)

Processes miscellaneous printer driver operations using a specified driver

On entry

R0 = reason code
R8 = number of printer driver to which to pass call
Other registers are reason code dependent

On exit

Reason code dependent

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call processes miscellaneous printer driver operations using a specified driver. It is identical to PDriver_MiscOp, save that the call gets passed to the driver specified in R8. For details of the various reason codes see page 3-658 onwards.

This call is not available under RISC OS 2.

Related SWIs

PDriver_MiscOp (page 3-656)

Related vectors

None

PDriver_SetDriver (SWI 8015C)

Configures the current printer driver

PrinterDM version

Sets the current printer dumper, if PrinterDM is the current printer driver

On entry

R1 = printer dumper number (see page 3-675)
R2 = pointer to command to ensure printer dumper present
R3 = pointer to 256 byte data block giving dumper configuration data
R4 = pointer to 256 byte block giving PDriverDP and dumper configuration data
R5 = configuration word for dumper

On exit

R1 - R5 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 current printer dumper to that specified by the number held in R1. It does so by calling the dumper's reason code handler with reason code 0. For a list of current printer dumper numbers, see page 3-675.

R2 points to a command line, used to load the printer dumper if it is not already loaded. The length of this command line should not exceed 256 bytes including the terminating character.

R3 and R4 are both pointers to 256 byte data blocks containing configuration data for the dumper. PDriverDP copies each block, adds some information to the block pointed to by R4 (see below), and then passes the dumper pointers to the copies. Consequently you may free the original buffers on exit.

R5 is a configuration word, the meaning of which is dumper-specific.

The information that PDriverDP adds to the copy of the block pointed to by R4 consists of 12 unsigned bytes at the start of the block (which overwrite the existing contents):

Offset	Meaning
0	height in dots of a strip (pin height × no. of vertical interlace passes: ie PrintEdit's DumpDepth)
1	number of vertical interlace passes – 1(ie PrintEdit's x interlace)
2	number of horizontal interlace passes – 1(ie PrintEdit's y interlace)
3	number of passes over line – 1: for multiple pass printing, eg colour
4	strip type (see page 3-675)
5	output depth (bits per pixel): can only be 1 (monochrome) or 8 (grey or colour
6	number of passes per strip, 0 ⇒ 1 pass: useful for colour separation
7	number of current pass
8 - 11	PDriverDP's copy of dumper's private word

and 4 signed words that are appended to the block:

Offset	Meaning
256	configuration word for dumper (as passed in R5)
260	pointer to active printer dumper
264	printer dumper number (as passed in R1)
268	left margin in pixels (calculated from the Printers application's paper sizes)

For details of the information the Printers application places in the buffers and configuration word when it makes this call, see the documentation of *PDumperReason_SetDriver* on page 3-678.

This call is not available under RISC OS 2.

PDriverPS version

This SWI is used as part of the private interface between the Printers application and PDriverPS. You must not use it from your own applications; it is only of relevance to anyone wishing to replace the current PostScript printer drivers. See the section entitled *Extending the printing system* on page 3-602.

This call is not available under RISC OS 2.

Related SWIs

None

Related vectors

None

Example program

This is an example BASIC procedure that does a standard ‘two up’ printing job:

```
DEFPROCprintout(firstpage%, lastpage%, title$, filename$, fontptr%)
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 "PDriver_SelectJob",H%,title$ TO O%
LOCAL ERROR
ON ERROR LOCAL:RESTORE
ERROR:SYS"PDriver_AbortJob",H%:SYS"PDriver_SelectJob",O%:PROCpasserror

PROCdeclarefonts(fontptr%)
:
REM Now we decide how two pages are to fit on a piece of paper.
LOCAL left%, bottom%, right%, top%
REM see below for an explanation of PROCgetdocumentsize
PROCgetdocumentsize(box%)
SYS "PDriver_PageSize" TO ,,,left%,bottom%,right%,top%
REM see below for an explanation of PROCfittwopages
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 time around.
  SYS "PDriver_GiveRectangle", page%, box%, matrix%, origin1%, white%
  IF page%<lastpage% THEN
    SYS "PDriver_GiveRectangle", 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 "PDriver_DrawPage",1,box2%,0,0 TO copiesleft%,,pagetoprint%
  WHILE copiesleft%<>0
    REM see below for an explanation of PROCdrawpage
    PROCdrawpage(pagetoprint%, box2%)
    SYS "PDriver_GetRectangle",,box% TO copiesleft%,,pagetoprint%
  ENDWHILE
:
REM End of page loop
NEXT
```

```
REM All pages have now been printed. Terminate this print job.
SYS "PDriver_EndJob",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 if no print job was active).
SYS "PDriver_SelectJob",O%
:
REM Go back to the caller's error handler.
RESTORE ERROR
REM Close the destination file.
CLOSE#H%
ENDPROC
:
DEFPROCpasserror
SYS "BASICTrans_Message",42,ERL,REPORT$ TO ;flags%
IF (flags% AND 1)<>0 THEN
  REPORT:IF ERL<>0 THEN PRINT" at line "ERL ELSE PRINT
ENDIF
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:

`DEFPROCdeclarefonts(fontptr%)`

- checks the printer driver's features bit and, if necessary, declares the fonts in the structure pointed to by `fontptr%`.

`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%)
```

- draws 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, you may like to use multitasking. To do so, you should regularly use a sequence like the following during printing:

```
SYS "PDriver_SelectJob",0%
SYS "Wimp_Poll",mask%,area% TO reason%
…
```
process reason% as appropriate
```
…
SYS "PDriver_SelectJob",H% TO 0%
```

However, you should first see the section entitled *Multitasking whilst printing* on page 3-571, which explains the issues involved in multitasking printing.

65 Printer Dumpers

Introduction and Overview

This chapter describes printer dumper modules, used in conjunction with the PDriverDP module to provide support for bit image printing.

The way in which these modules fit in with the rest of the printing system is explained in the previous chapter on printer drivers, in the section entitled *The structure of the printing system* on page 3-598.

The relationship of printer dumpers to the PDriverDP module is very similar to that between printer drivers and the printer sharer module. In both cases the ‘parent’ module issues a service call when it starts, and it is the duty of the ‘child’ to register at this time. When registering the child passes to the parent module an entry point. The parent module calls this entry point to pass on those calls that it cannot handle because they are device dependent.

For printer dumper modules, the service call they should respond to is Service_PDumperStarting (see page 3-686). They should register themselves by calling PDriver_MiscOpForDriver &80000000 (see page 3-663 and page 3-666), either on receiving this service call or on starting up. They should ignore any errors from this call.

When a printer dumper module dies it must deregister itself by calling PDriver_MiscOpForDriver &80000001 (see page 3-664 and page 3-666), this time refusing to die if an error is returned.

Technical Details

Reason code handler entry and exit conditions

A printer dumper's reason code handler is called in SVC mode. R11 always contains the reason code for the call. The following reason codes are assumed to be supported by **all** PDumper modules:

Value	Name	on page
0	PDumperReason_SetDriver	page 3-678
1	PDumperReason_OutputDump	page 3-680
2	PDumperReason_ColourSet	page 3-681
3	PDumperReason_StartPage	page 3-682
4	PDumperReason_EndPage	page 3-683
5	PDumperReason_StartJob	page 3-683
6	PDumperReason_AbortJob	page 3-684
7	PDumperReason_MiscOp	page 3-684

Other reason codes are reserved for future use. If a dumper receives an unknown reason code, it should return the call with all registers preserved.

R12 is always a pointer to the printer dumper's private word, as passed in R3 when it first registered itself using PDriver_MiscOp &80000000 (see page 3-663). The remaining register usage is reason code dependent, and detailed below.

All calls can return an error, which is done in the normal way by returning with the V flag set and R0 pointing to an error block.

Escape will be enabled during most calls, especially for reason code 1 (PDumperReason_OutputDump) as this can often take quite a long time. If a dumper is going to spend a long time processing a request, it should check the escape state regularly and return an escape error if necessary.

If you are writing a dumper, you should preserve all registers, save for those explicitly used to return a value.

Common parameters

Printer dumper numbers and names

These are the current printer dumper numbers in use, and the names of the corresponding PDumper modules:

Value	Meaning	PDumper module
0	Sprite device	PDumperSP
1	Dot-matrix generic	PDumperDM
2	LaserJet compatible device	PDumperLJ
3	Apple ImageWriter device	PDumperIW
4	Dot reducing 24 pin device	PDumper24
5	Colour Deskjet compatible device	PDumperDJ

You must register any new printer dumper numbers with Acorn; see the section entitled *Printer driver and printer dumper numbers* on page 4-556.

Strip types

Most calls to the printer dumper reason code handler specify the type of strip being printed. The values used are:

Value	Meaning
0	monochrome
1	grey scale
2	256 colour

How the PDumper reason codes get called

This is a ‘code fragment’ description of printing:

```
Use Printers message protocol if running under the Wimp

This is done by !Printers
PDriver_SetDriver

This is done by applications
PDriver_Info
REM check what features are available (eg PDriver_DeclareFont)

OPEN printer:

PDriver_SelectJob

IF driver supports PDriver_DeclareFont THEN
        WHILE fonts to be declared
                PDriver_DeclareFont font
        ENDWHILE
        PDriver_DeclareFont end of font list
ENDIF

FOR each page to print
        REPEAT
                PDriver_GiveRectangle
        UNTIL all rectangles declared
        REM typically only one rectangle given, specifying whole page

        PDriver_DrawPage
        WHILE more rectangles to print
                plot returned rectangle using supported output calls
                PDriver_GetRectangle
        ENDWHILE
ENDFOR

PDriver_EndJob

CLOSE printer:
```

Here is the same ‘code fragment’ description of printing showing where the various reason codes are used in calls to a dumper’s reason code handler:

```
Use Printers message protocol if running under the Wimp

This is done by !Printers
PDriver_SetDriver
  PDumperReason_SetDriver                                  the printer gets configured

This is done by applications
PDriver_Info
REM check what features are available (eg PDriver_DeclareFont)

OPEN printer:

PDriver_SelectJob
  PDumperReason_StartJob

IF driver supports PDriver_DeclareFont THEN
          WHILE fonts to be declared
                    PDriver_DeclareFont font
          ENDWHILE
          PDriver_DeclareFont end of font list
ENDIF

FOR each page to print
          REPEAT
                    PDriver_GiveRectangle
          UNTIL all rectangles declared
          REM typically only one rectangle given, specifying whole page

          PDriver_DrawPage
            PDumperReason_StartPage
          WHILE more rectangles to print
                    plot returned rectangle using supported output calls
                      PDumperReason_ColourSet
                    PDriver_GetRectangle
                      PDumperReason_OutputDump
          ENDWHILE
            PDumperReason_EndPage
            PDumperReason_AbortJob (R3=0)   to tidy workspace for page end
            PDumperReason_StartPage         for the next copy
ENDFOR

PDriver_EndJob
  PDumperReason_AbortJob (R3≠0)                            to tidy up job workspace

CLOSE printer:
```

Printer Dumper reason codes

PDumperReason_SetDriver (reason code 0)

On entry

R1 = printer dumper number
R2 = pointer to command to ensure printer dumper present
R3 = pointer to 256 byte data block giving dumper configuration data
R4 = pointer to 272 byte block giving PDriverDP and dumper configuration data
R5 = configuration word for dumper
R11 = 0 (printer dumper reason code)
R12 = pointer to dumper's private word

On exit

—

Details

This is called when the printer dumper is being selected by PDriver_SetDriver (see page 3-667).

R1 is unlikely to be useful to the printer dumper, which probably knows its own number.

The command pointed to by R2 is again unlikely to be useful to the printer dumper. The command may not have been used, as the dumper may already have been loaded when PDriver_SetDriver was called.

This call sets the current printer dumper to that specified by the number held in R1. It does so by calling the dumper's reason code handler with reason code 0.

R3 and R4 are both pointers to data blocks containing configuration data for the dumper. Both blocks are transient, and so you must copy any data you need before returning to the caller. When this reason code is called by the Printers application in RISC OS 3 (version 3.10) via PDriver_SetDriver (page 3-667), the contents of the data blocks are as follows:

- The data block pointed to by R3 holds the name of the palette file to be used, (eg: 'Printers:Palettes.0'), which is supplied by the Printers application.

- The data block pointed to by R4 is split into two categories:

  1 Bytes 0 - 11 and bytes 256 - 271 contain information added by PDriverDP, as detailed on page 3-668.

  2 Bytes 12 - 255 (244 bytes in all) contain information passed by the Printers application – mainly the control strings that are defined using the PrintEdit application. The location of each string within the buffer is given as a byte offset from the start of these 244 bytes; at this offset there will be a byte giving the string's length, followed by the string itself (without a terminator). An offset of zero implies that there is no corresponding string.

  The *italicised* words below show the names used by PrintEdit for the passed information. All bytes, whether offsets or values, are unsigned quantities; all words are signed:

Offset	**Meaning**
12+0	*data length multiplier*
+1	*data length added* (line as printer sees it is dlm × width + dla)
+2	*dump height* – ie bit rows high per dump
+3	offset to *page start* string
+4	offset to *page end* string
+5	offset to *line return* string (for x interlace)
+6	offset to *line skip* string (for blank lines)
+7	offset to *line end 1* string
+8	offset to *line end 2* string for 2nd vertical interlace
+9	offset to *line end 3* string for 3rd vertical interlace
+10	offset to *zero skip* string
+11	offset to *line start 1* string for pre length output
+12	offset to *line start 2* string for post length output
+13	offset to *line pass 1* string for colour 1, pre length output
+14	offset to *line pass 1b* string for colour 1, post length output
+15	offset to *line pass 2* string for colour 2, pre length output
+16	offset to *line pass 2b* string for colour 2, post length output
+17	offset to *line pass 3* string for colour 3, pre length output
+18	offset to *line pass 3b* string for colour 3, post length output
+19	offset to *line pass 4* string for colour 4, pre length output
+20	offset to *line pass 4b* string for colour 4, post length output
+21	offset to string to *set lines* per page
+22	number of lines per page (set from text height in !Printers)
+23	number of leading zeros to leave (always set to 1/6" by !PrintEdit)
+24	multiplier used to convert from output to no. of dpi to skip (derived from *skip resolution*)
+28	divider used to convert from output to no. of dpi to skip (derived from *skip resolution*)
+32	short advance used for roll paper (always set to 1" by !PrintEdit)

+36	offset to *form feed* string
+37	reserved (3 bytes)
+40	*paper x offset* (ie x pixels to subtract from margin)
+44	*paper y offset* (ie y pixels to subtract from margin)

R5 is a configuration word, the meaning of which is dumper-specific. The top byte will always be the version number of PDriverDP. This is 3 for RISC OS 3 (version 3.10); if you receive a lower value you should fault it.

Other bits of the configuration word currently defined are:

Bit	Meaning when set
0	Horizontal output (PDumperDM) Supports multiple copies (PDumperLJ)
1	Roll paper feed (PDumperDM) Supports compression (PDumperLJ)
2	Do not send form feeds (PDumperLJ)
3	Use PaintJet paper movement commands (PDumperLJ)

All bits not described above are reserved.

PDumperReason_OutputDump (reason code 1)

On entry

R0 = pointer to start of strip data giving bitmap for strip
R1 = file handle for output
R2 = strip type (see page 3-674)
R3 = width output dump should be, in pixels
R4 = height of strip in pixels
R5 = width of strip in bytes (ie amount to add to R0 to go down one line)
R6 = halftoning information:
    bits 0 - 7 = horizontal resolution in pixels
    bits 8 - 15 = vertical resolution in pixels
    bits 16 - 31 reserved
R7 = pointer to copy of PDriverDP and dumper configuration data (see page 3-678)
R8 = pointer to private word for job (see *PDumperReason_StartJob* on page 3-683)
R11 = 1 (printer dumper reason code)
R12 = pointer to dumper's private word

On exit

—

Details

This routine is called by PDriverDP when it has generated a strip for output at the dumper's required depth. The strip is passed as a bitmap stored in sprite format. R0 points to the bitmap data, not to a sprite header; there may be a header preceding the data, but the dumper must not rely on this.

This routine should then render the data to the file handle passed in R1. Interlacing will already have been catered for by PDriverDP.

The strip can be at either 1 or 8 bits-per-pixel. The values stored relate to the byte values returned from PDumperReason_ColourSet (see page 3-681).

PDumperReason_ColourSet (reason code 2)

On entry

R0 = physical colour (&BBGGRR00)
R2 = strip type (see page 3-674)
R3 = pointer to private word for job (see *PDumperReason_StartJob* on page 3-683)
R4 = halftoning information:
    bits 0 - 7 = horizontal resolution in pixels
    bits 8 - 15 = vertical resolution in pixels
    bits 16 - 31 reserved
R5 = pointer to copy of PDriverDP and dumper configuration data (see page 3-678)
R11 = 2 (reason code)
R12 = pointer to dumper's private word

On exit

R3 = strip type dependent colour number

Details

This call is made when ever the PDriver needs to convert a physical colour to a colour number. The colour number is specific to the printer dumper and strip type.

The printer dumper can use PDumperSupport to do this by calling PDumper_SetColour (page 3-700).

PDumperReason_StartPage (reason code 3)

On entry

R0 = copies requested
R1 = file handle for output
R2 = strip type (see page 3-674)
R3 = number of blank pixel rows to skip before start of data
R4 = pointer to private word for job (see *PDumperReason_StartJob* on page 3-683)
R5 = pointer to copy of PDriverDP and dumper configuration data (see page 3-678)
R6 = left margin in pixels
R7 = horizontal and vertical resolution:
    bits 0 - 15 = x pixel resolution in dpi
    bits 16 - 31 = y pixel resolution in dpi
R11 = 3 (reason code)
R12 = pointer to dumper's private word

On exit

R0 = number of copies to be performed
R3 = number of blank pixel rows remaining to skip before start of data

Details

This routine is called at the start of the page. This routine should set up the printer and skip to the correct print position.

If the printer can be requested to perform multiple copies itself then this routine should return the number of copies passed in adjusted appropriately – the returned number of copies being how many times PDriverDP will print a given page.

However much line skipping is performed at the page start should be subtracted from R3 before returning; PDriverDP will perform the rest. Note that R3 on return must not be negative.

The routine is also passed the horizontal margin. This cannot be modified, and it is assumed that the dumping routine will process it appropriately, for example by padding each line start with null bytes, or by moving the graphics origin to the right.

If the printer is a generic dot matrix (ie the printer dumper number is 1) and it has roll paper (ie bit 1 of R5!256 is set), then R3 should be ignored and instead R5!32 pixel rows should be skipped (ie the short advance used for roll paper). As above, you can return any remainder of this in R3.

PDumperReason_EndPage (reason code 4)

On entry

R1 = file handle for output
R2 = strip type (see page 3-674)
R3 = pointer to private word for job (see *PDumperReason_StartJob* on page 3-683)
R4 = pointer to copy of PDriverDP and dumper configuration data (see page 3-678)
R11 = 4 (reason code)
R12 = pointer to dumper's private word

On exit

—

Details

When called the PDumper should output the end of page sequence to the file and then return.

PDumperReason_StartJob (reason code 5)

On entry

R0 = pointer to private word for job (see below)
R1 = file handle for output
R2 = strip type (see page 3-674)
R5 = pointer to copy of PDriverDP and dumper configuration data (see page 3-678)
R11 = 5 (reason code)
R12 = pointer to dumper's private word

On exit

—

Details

When this is called the printer dumper should handle setting up the required workspace for the job. The private word for the job should be treated the same way as the private word for a dumper, that is, the meaning is defined by the dumper. Typically workspace is allocated and attached to the private word.

The printer dumper can use PDumperSupport to do this by calling PDumper_StartJob (page 3-696).

PDumperReason_AbortJob (reason code 6)

On entry

R0 = pointer to private word for job (see *PDumperReason_StartJob* on page 3-683)
R1 = file handle for output
R2 = strip type (see page 3-674)
R3 = subreason code: 0 ⇒ end of a page, else end of document
R11 = 6 (reason code)
R12 = pointer to dumper's private word

On exit

—

Details

When this is called the PDumper should release any workspace specified, ie all document specific workspace at the end of document, or all page specific workspace at the end of a page. The PDumper should not output anything to the file.

If an error occurs during a print sequence this call will be made with an 'end of document' subreason code; other calls will not be used.

The printer dumper can use PDumperSupport to do this by calling PDumper_TidyJob (page 3-698).

PDumperReason_MiscOp (reason code 7)

On entry

R0 = PDriver_MiscOp reason code
R11 = 7 (reason code)
R12 = pointer to dumper's private word
Other register usage as for PDriver_MiscOp SWI – see page 3-656 onwards

On exit

—

Details

This call is provided so that dumpers can provide specific features that require an interface other than the control block. The current Acorn printer system does not use this call; if this reason code is used, current printer drivers merely return the call with all registers preserved.

Service Calls

Service_PDumperStarting (Service Call &66)

PDriverDP module starting up

On entry

R1 = &66 (reason code)

On exit

All registers preserved

Use

This service call is issued when the PDriverDP module starts up. Any printer dumpers resident at that time should declare themselves to PDriverDP by calling PDriver_MiscOpForDriver &80000000 (see page 3-663 and page 3-666).

Service_PDumperDying (Service Call &67)

PDriverDP module dying

On entry

R1 = &67 (reason code)

On exit

All registers preserved

Use

This service call is issued as a broadcast to inform printer dumpers that they have been deregistered and that the PDriverDP module is about to die.

66 PDumperSupport

Introduction and Overview

This chapter describes the PDumperSupport module, introduced in RISC OS 3 (version 3.10). This module's SWI interface provides colour matching, error diffusion and halftoning facilities for the use of printer dumpers. This avoids unnecessary duplication of code in each module, and provides a service for third party printer dumpers.

The way in which this module fits in with the rest of the printing system is explained in the earlier chapter on printer drivers, in the section entitled *The structure of the printing system* on page 3-598.

SWI calls

PDumper_Info (SWI &41B00)

Returns information about the PDumper support module

On entry

—

On exit

R0 = version number ×100 (eg version 1.23 stored as 123)
R1 = bit field of optional features implemented by support module:
bit 0 set ⇒ supports halftone grey
bit 1 set ⇒ supports error diffuse grey
bit 2 set ⇒ supports halftone colour
bit 3 set ⇒ supports error diffuse colour
bits 4 - 31 reserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call simply returns information about the PDumper support module, giving its version number and which optional features it supports.

This call is not available under RISC OS 2, nor under RISC OS 3 (version 3.00).

Related SWIs

None

Related vectors

None

PDumper_Claim (SWI &41B01)

Allocates a block of memory and links it into the chain

On entry

R0 = pointer to anchor word
R3 = size of block to be claimed
R4 = tag for block

On exit

R0 preserved
R2 = pointer to block allocated (on a word boundary)
R3, 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 allocates a block of memory and links it into the printer dumper's chain. The chain is specified by the anchor word, which is typically the printer dumper's private word. The size specified need not be a word multiple. The tag is a four byte value stored after the link point. Although you may claim multiple blocks with the same tag, you must be aware that if you subsequently call PDumper_Find it is uncertain which of these blocks it will find; see page 3-695.

This call is not available under RISC OS 2, nor under RISC OS 3 (version 3.00).

Related SWIs

PDumper_Free (page 3-694), PDumper_Find (page 3-695)

Related vectors

None

PDumper_Free (SWI &41B02)

Attempts to release a block of memory from the printer dumper's chain

On entry

R0 = pointer to anchor word
R2 = pointer to block to be released

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 attempts to release a block of memory from the printer dumper's chain. The chain is specified by the anchor word. If the block is not part of the specified chain then it is not released, and an error is generated.

This call is not available under RISC OS 2, nor under RISC OS 3 (version 3.00).

Related SWIs

PDumper_Claim (page 3-692), PDumper_Find (page 3-695)

Related vectors

None

PDumper_Find (SWI &41B03)

Scans the printer dumper's chain for a block of memory with the given tag

On entry

R0 = pointer to anchor word
R2 = tag for block

On exit

R2 = pointer to block found

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call scans the printer dumper's chain for a block of memory with the given tag, returning the first match it finds. The chain is specified by the anchor word.

If you have claimed several blocks with the same tag, you cannot be certain which one this call will return. If there is no match, this call generates an error.

This call is not available under RISC OS 2, nor under RISC OS 3 (version 3.00).

Related SWIs

PDumper_Claim (page 3-692), PDumper_Free (page 3-694)

Related vectors

None

PDumper_StartJob (SWI &41B04)

Sets up any workspace that is required for a job

On entry

R0 = pointer to anchor word
R1 = flags word: all bits reserved (must be zero)
R2 = pointer to filename of palette to load, or 0 if none to be loaded

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 sets up any workspace required for a job. The support module links this into the chain specified by the anchor word, which is typically the printer dumper's private word. You should call it at the start of a job (ie when your dumper is called with the reason code PDumperReason_StartJob).

The flags word in R1 is reserved for future expansion, and for the time being you must set it to zero. If non-null, R2 contains a pointer to the filename of a palette file to use for the job, which is loaded into a block with a tag of 1.

This call is not available under RISC OS 2, nor under RISC OS 3 (version 3.00).

Related SWIs

PDumper_TidyJob (page 3-698)

Related vectors

None

PDumper_TidyJob (SWI &41B05)

Releases workspace used for a job

On entry

R0 = pointer to anchor word
R1 = pointer to list of tags terminated by a null word, or 0
R2 = reason code: 0 ⇒ end of page, else end of document

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 routine releases workspace used for a job, as specified by R1 and R2. The support module releases this from the chain specified by the anchor word.

R1 points to a list of tags; any block having a matching tag will be released. Furthermore, any blocks allocated by the support module that are specific to the page or document (as given in R2) will be released.

This call is not available under RISC OS 2, nor under RISC OS 3 (version 3.00).

Related SWIs

PDumper_StartJob (page 3-696)

Related vectors

None

PDumper_SetColour (SWI &41B06)

Processes the colour setting required by the printer dumper

On entry

R0 = pointer to anchor word
R1 = physical colour (&BBGGRR00)
R2 = strip information:
    bits 0 - 7 = strip type (see page 3-675)
    bits 24 - 31 = pass number
R4 = halftoning information:
    bits 0 - 7 = horizontal resolution in pixels
    bits 8 - 15 = vertical resolution in pixels
    bits 16 - 31 reserved (must be zero)

On exit

R0 - R2 preserved
R3 = strip type dependant colour number
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 processes the colour setting required by the printer dumper. In doing so, it scans the chain specified by the anchor word for any palette block to use, decodes the strip type, and then returns a suitable colour number.

If this call generates an error, R3 may be corrupted on return.

This call is not available under RISC OS 2, nor under RISC OS 3 (version 3.00).

Related SWIs

PDumper_Claim (page 3-692), PDumper_Find (page 3-695)

Related vectors

None

PDumper_PrepareStrip (SWI &41B07)

Processes a bitmap into a format suitable for printing

On entry

R0 = pointer to anchor word
R1 = pointer to bit image data
R2 = resulting format of the strip
    bits 0 - 7 = format:
        0 ⇒ grey level (halftoned)
        1 ⇒ grey level (diffused)
        2 ⇒ colour (halftoned)
        3 ⇒ colour (diffused)
    all other bits reserved (must be zero)
R3 = width output dump should be, in pixels
R4 = height of strip in pixels
R5 = width of strip in bytes (ie amount to add to R1 to go down one line)
R6 = halftoning information:
    bits 0 - 7 = horizontal resolution in pixels
    bits 8 - 15 = vertical resolution in pixels
    bits 16 - 31 reserved (must be zero)

On exit

R0 - R6 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call processes the specified 8 bit-per-pixel bitmap generated by the PDriverDP module into a format suitable for printing by the relevant output routine.

This call is not available under RISC OS 2, nor under RISC OS 3 (version 3.00).

Related SWIs

None

Related vectors

None

PDumper_LookupError (SWI &41B08)

Accesses the internal error handling routines within the support module

On entry

R0 = pointer to error block, including message token
R1 = pointer to string to substitute for '%0', or zero if no string

On exit

R0 = pointer to resolved error block
V flag set

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call accesses the internal error handling routines within the support module. On entry R0 is a pointer to an error block, the error message in which is a token for one of the messages in the file Resources:$.Resources.PDrivers.Messages. The support module extracts the corresponding message from the file; it then scans it for the string '%0', for which (if found) it substitutes the string pointed to by R1.

Using this call removes any need to have MessageTrans routines within a printer dumper, as most printer dumpers simply resolve errors.

This call is not available under RISC OS 2, nor under RISC OS 3 (version 3.00).

Related SWIs

None

Related vectors

None

PDumper_CopyFilename (SWI &41B09)

Copies a specified filename into a buffer

On entry

R0 = pointer to buffer into which to copy string
R1 = size of buffer
R2 = pointer to string to be copied (control-character terminated)

On exit

R0 = pointer to character in buffer after terminating null
R2= pointer to last character copied from string

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call copies the specified filename into a buffer. The routine terminates on any character ≤32 and converts it to a null. An error is generated if an overflow occurs (ie more than R1 characters need to be copied).

This call is not available under RISC OS 2, nor under RISC OS 3 (version 3.00).

Related SWIs

None

Related vectors

None

67 Printer definition files

Introduction and Overview

The *RISC OS User Guide* has a chapter describing how to use PrintEdit to create new printer definition files, either by starting from scratch or by editing an existing definition. This section contains extra information to help you with more complex tasks.

When using PrintEdit, it is important you understand the Printers back end and the printer dumper used by the printer definition file being edited. The data held in a printer definition file is just that – data. It has no meaning and no pre-ordained use until the printer driver software starts to interpret it. The meaning of any individual data item in the printer definition file is actually imposed by the Printers application and the back end it is using, and by PDriverDP and the printer dumper it is using, rather than by PrintEdit. If PrintEdit lets you type in a certain number or select a certain option, it does not necessarily follow that this will have the desired effect on the software. An example of this is the Dump depth field:

- PrintEdit will let you type in any number.
- PDriverDP uses this number when rendering the bit image, and can also cope with any number (except zero).
- PDumperDM, on the other hand, assumes both that the Dump depth is a multiple of eight, and that the Dump height times the number of vertical interlace passes is equal to the Dump depth. You can easily type in a number which does not satisfy these conditions, but if you do so PDumperDM may fail in an arbitrary way; even if it doesn't, the printout will almost certainly be incorrect.

If an existing Printers back end and/or an existing printer dumper will not do what you need, then you will have to write one. To be able to make this decision, you need to find out precisely what the existing back ends and printer dumpers can do by reading this section. It gives a set of PrintEdit example windows, and discusses what can be done with each field. All of the information is available to a Printers back end, but the Printers application only imposes a meaning on some of it. Likewise, all of the data for the current graphics resolution is available to a printer dumper, but PDriverDP only imposes a meaning on a small amount of it.

Notation

This information is passed to the dumpers using PDumperReason_SetDriver. This section uses BBC BASIC conventions to show how such data is passed: so R4!(12+40) indicates the word at offset 52 from the location pointed to by R4, whereas R4?(12+0) indicates the byte at offset 12 from the same location. For full details of how the data is passed, you must see page 3-678.

Version numbers

There have been some changes between RISC OS 3 (version 3.00) and RISC OS 3 (version 3.10) to the graphics sections of printer definition files. Quite a few extra fields have been added, and exactly what all of the fields are used for and when have changed slightly too. Only the RISC OS 3 (version 3.10) behaviour is documented here. Some information on the differences is given in the Printers.Read_Me file. To ensure that your dumper is dealing with format data from RISC OS 3 (version 3.10) or later, check the version number passed to PDumperReason_SetDriver, which should be 3 or greater.

Overview of following sections

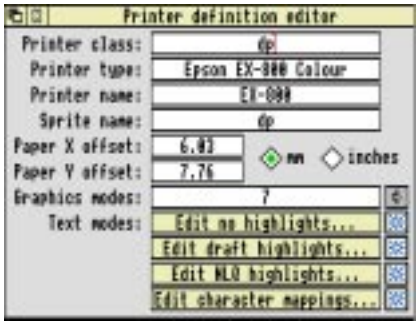
The most common use of PrintEdit is for dealing with Epson/IBM compatible printers. In each of the sections below we will use them as the core example and for discussing general points. Any points specific to other classes of printers appear at the end of each section.

Technical details

Printer definition editor

General points, and Epson and IBM compatible printers

The appearance of the **Printer definition editor** window when the Epson EX-800 printer definition file is loaded is as follows:



The **Printer class** represents a type of printer; it determines which back end Printers uses for the printer. For Epson and IBM-compatible printers this field should be set to **dp**, so that the !Printers.dp back end is used.

Printer type is the full name of the printer.

Printer name is the name you want to appear underneath the printer on the icon bar. The name can be up to 10 characters long.

Sprite name determines the sprites to be used by the Printers application as the printer icon on the icon bar. When the printer is the default, Printers uses the named sprite, which should be cream. When the printer is not the default, Printers precedes the sprite name with ‘s_’ and instead uses that sprite – which should be grey. See !Printers.dp.Resources.!Sprites for example sprites.

For Epson and IBM-compatible printers this field should be set to **dp**, which makes the Printers application use the two sprites dp and s_dp.

The **paper offsets** represent the top (Y) and left (X) sections of the paper on which the printer cannot physically print. The **Paper Y offset** is the amount of cut sheet paper which has already gone past the print head before it can print anything; this differs for different printer models. Similarly the **Paper X offset** is the small section at the left hand edge on which the head cannot print, although why this is so is not always obvious.

Together the paper offsets define the logical (0,0) origin on the physical paper. The Printers.Read_Me file contains details on using the Printers.Top_Left file to set the offsets for Epson and IBM compatible dot matrix printers.

Normally the paper offsets will be set correctly for the printer being used. However, if necessary you can change the paper offsets away from their true values, probably by using negative numbers. This allows you to move the image around on the paper if you need to do so and there is no facility for this in the application doing the printing. Unfortunately it is not easy to do the same sort of trick with PostScript printers, so think about whether you ever need to use PostScript before resorting to this trick. Do not try to use the Graphics margins in Printers to move the image, as that is not their function.

PrintEdit actually holds the paper offsets in each of the graphics resolution data blocks in units of pixels at that printer resolution, converting them from the units in which they were specified as it does so. These values are passed to the dumper by PDumperReason_SetDriver in words R4!(12+40) and R4!(12+44); it is then the responsibility of the printer dumper to act on this information if it wishes to. The Acorn printer dumpers all subtract the paper offsets from the top and left margins passed to them by PDriverDP (also in units of pixels), since the section of the margins which is within the paper offsets has already been skipped implicitly by the printer mechanism.

Graphics modes shows the number of graphics modes that have been defined for your printer. For details of editing their settings, see the sections starting on page 3-714.

Text modes defines the type of text modes your printer can use. For Epson and IBM-compatible printers there are four categories available.

Integrex printers

The appearance of the **Printer definition editor** window with the Printers.Integrex.ColJet132 file loaded is:

The Integrex back end is combined with the generic dot-matrix back end. Consequently the **Printer class** remains as **dp**. However, the **Sprite name** is **ix**, so that the icon on the icon bar is the same under RISC OS 3 as it was for !PrinterIX on RISC OS 2.

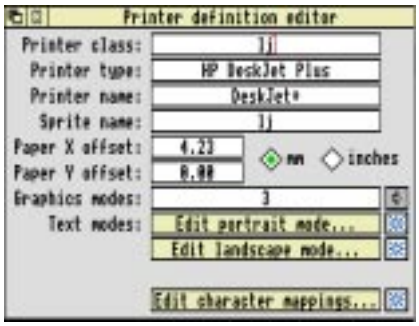
ImageWriter printer

The appearance of the **Printer definition editor** window with the Printers.Apple.ImgWriteII file loaded is:

The Image Writer back end is combined with the generic dot-matrix back end. Consequently the **Printer class** remains as **dp**.

HP LaserJet compatible printers

The appearance of the **Printer definition editor** window with the Printers.HP.DeskJet+ file loaded is:

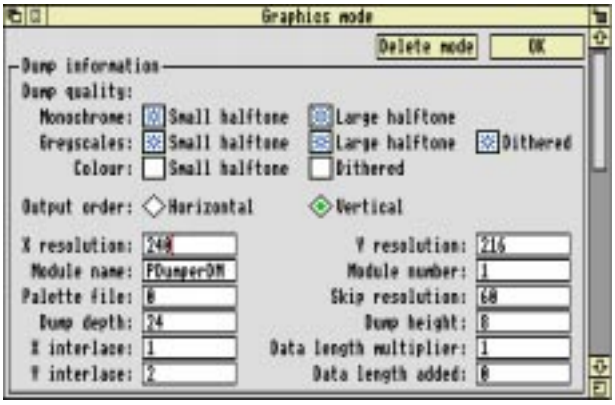


The differing text mode titles (**Edit portrait mode** and **Edit landscape mode** rather than ‘Edit no highlights’, ‘Edit draft highlights’ and ‘Edit NLQ highlights’) are set up by PrintEdit when the **Printer class** is **lj**. The information is still stored in exactly the same way as for Epson and IBM-compatibles.

Graphics mode: Dump information

General points, and Epson and IBM compatible printers

The **Dump information** in the **Graphics mode** window for the Epson.EX-800 file at a resolution of 240 by 216 dpi is shown below:



The meaning of much of this information is imposed by PDriverDP, and to a lesser extent the back end. The items which are not used by PDriverDP or the Printers application and are passed to a dumper using PDumperReason_SetDriver are:

- the **Output order** (bit 0 of R5 or of R4!256)
- the **Data length multiplier** (R4?(12+0))
- the **Data length added** (R4?12+1).

It is the dumper that gives meaning to the information these fields hold. You can therefore use them to pass **any** numeric information to a new dumper, irrespective of their title in this PrintEdit window.

The **Dump quality** boxes should normally all be selected, as most of the software supports these features on most printers. You should not select the colour options for dot matrix printers that do not support colour. It is also good practice not to enable the colour options for resolutions which use horizontal or vertical interlace. The Dump quality boxes which you select will be made available by the relevant back end in the Quality menu of the printer configuration window.

Output order is specific to dp class printers, and is only acted on by PDumperDM. It selects between the parts of the dumper that are for Epson/IBM compatible printers (output order is **Vertical**), or the parts that are for Integrex ColourJet 132 compatible printers (output order is **Horizontal**).

Note that when **Horizontal order** has been selected many of the other **Dump information** and **Dump strings** settings either become irrelevant, or must be set to certain values. See the section entitled *Integrex printers* on page 3-719.

X (horizontal) and **Y resolution** (vertical) define the graphics resolution in dots per inch. These should be given in your printer manual, but may be in different units. The printer manual will usually quote resolutions before vertical interlacing has been applied (see later), so in this case the manual would quote 240 by 72 dpi, rather than 240 by 216 dpi (since 216/3 = 72). The manual is also likely to give dots per line rather than dots per inch for the horizontal resolution; for example, 960 dots per line on 8 inch paper is 120 dpi horizontal resolution. The vertical resolution is often omitted altogether, in which case it is likely to be 72 dpi for 9 pin printers, 180 dpi for 24 pin printers doing 24 pin graphics, 60 dpi for Epson 24 pin printers doing emulated 8 pin graphics, and 72 dpi for IBM 24 pin printers doing emulated 8 pin graphics.

Module name and **Module number** define the PDumper module used for the printer. For more information see the section entitled *Printer dumper numbers and names* on page 3-675.

Palette file defines the palette file name, which is currently always 0. This corresponds to the file !Printers.Palettes.0 (or rather Printers:Palettes.0). This pathname is constructed by the back end from the filename given in the printer definition file. It is passed to the dumpers by PDumperReason_SetDriver in the string pointed to by R3. The Acorn printer dumpers all pass this pathname to the PDumperSupport module, which loads and uses this file for colour matching and halftoning data.

Skip resolution defines the leading zero skip resolution in dots per inch. This is almost always 60 for Epson printers (the resolution of 27, '$') and 120 for IBM printers (the resolution of 27, 'd').

The zero skip resolution as passed to the dumpers needs some explanation. It is passed in a form that is easier for the dumper code to use, as a multiplier (R4!(12+24)) and a divider (R4!(12+28)). There is also a specified number of leading zeros you should leave to allow the print head time to accelerate (R4?(12+23)). Finally you should include the left margin in output pixels (R4!268), by adding it to the number of actual zero printer pixels at the start of your output data.

To convert from a number of zero pixels at the output horizontal resolution to a number of zero skip pixels, use the following formula:

skip_zeros = ((output_zeros + left_margin – run_up) × multiplier) DIV divider

The number of actual zero data pixels which should still be output as print data is a combination of the remainder from this and the run_up itself, given by:

run_up + (((output_zeros + left_margin – run_up) × multiplier) MOD divider) DIV multiplier

You can probably perform the DIV and the MOD of ((output_zeros + left_margin – run_up) × multiplier) as a single operation if you are writing your dumper in assembler. If (output_zeros + left_margin – run_up) is negative, then you should do no leading zero skipping, and the actual number of print data zeros to output should be:

output_zeros + left_margin

Remember that if you need to use paper offsets in your dumper (which depends on the type of printers it is for), before using the left_margin in the above calculations you must subtract the paper X offset from it:

left_margin = (R4!268) – (R4!(12+40))

If this gives a negative left margin, then set it to zero. The divider and/or multiplier are optimised down to 1 if possible by PrintEdit (eg. 180 dpi output, 60 dpi skip gives multiplier of 1 and divider of 3), so your DIV and MOD code should be optimised for small numbers (1 in particular), or should treat 1 as a special case if not optimised.

Dump depth (R4?0) is the number of pins on the print head (ie **Dump height**) multiplied by the number of vertical interlace passes (ie **Y interlace** + 1); so for the EX-800 at 240 by 216 dpi, the **Dump depth** of 24 is obtained from 8 pins and a **Y interlace** of 2, giving 8 × (2 + 1) = 24. PDumperDM (with vertical output) requires the **Dump depth** to be a multiple of 8.

Dump height (R4?(12+2)) is the number of pins used for graphics printing on the print head; for example it is 8 on 9 pin printers. PDumperDM (with vertical output) requires the **Dump height** to be a multiple of 8.

X interlace (R4?2) is the number of horizontal interlace passes of the print head (starting from pass 0). For example at 240 dpi horizontal resolution most printers cannot print adjacent dots, so two passes (**X interlace** of 1) are required, with alternate dots set to zero:

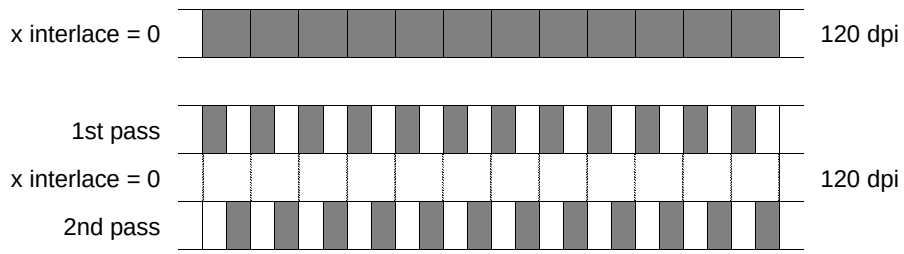


Figure 67.1 X interlacing with two horizontal passes

Note that this diagram shows a simplified view of the situation. In particular, it shows the dots made by the pins as being half as wide when performing interlacing (240 dpi). This is not the case, since the pins are obviously fixed in size, and the dots are just as

wide as for 120 dpi. Each dot printed actually covers the entire of the blank (zero) dot to the right of it, with the obvious effect that dots from the two interlace passes actually overlap. This does not really affect the resolution of the printout, the actual effect being that the centre of a dot is 1/480" further right than the code thinks it is, and hence so is the entire printout. It does however make the printout darker.

Y interlace (R4?1) is the number of vertical interlace passes of the print head (starting from pass 0). For example a 9 pin printer doing 8 pin graphics is fundamentally doing 72 dpi vertical resolution, because that is how far apart the pins are. But by feeding the paper by a fraction of the pin separation and making another pass of the head, greater resolution can be achieved at the cost of speed:

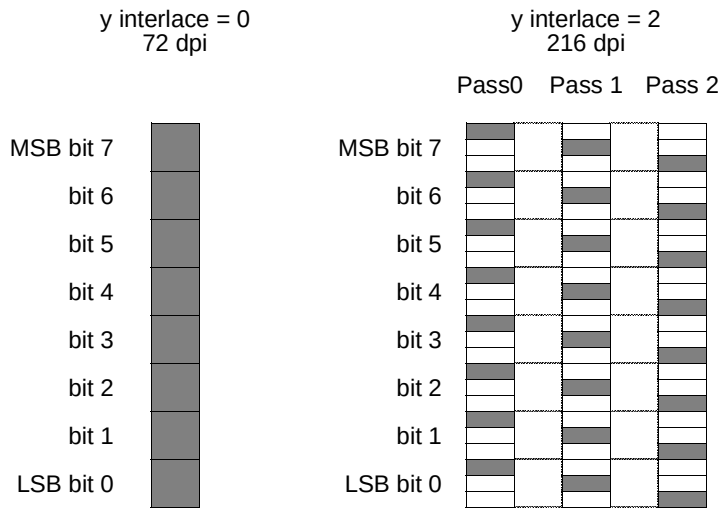


Figure 67.2 Y interlacing with two horizontal passes

After pass 0 the paper is fed by 1/216", and again after pass 1. After pass 2 the paper is fed by 22/216", so the total feed is a full head width for an eight pin head, since 24/216" = 8/72".

Just as for horizontal interlacing, this diagram shows a simplified view of the situation. In particular, it shows the dots made by the pins as being one third height when performing interlacing (216 dpi). This is not the case, since the pins are obviously fixed in size, and the dots are just as tall as for 72 dpi. Each dot printed actually covers both of the blank dots below it, with the obvious effect that dots from the three interlace passes actually overlap. This does not really affect the resolution of the printout, the actual effect being that the centre of a dot is 1/72" further down than the code thinks it is, and hence so is the entire printout. It does make the printout darker however.

The **Data length multiplier** (R4?(12+0)) and **Data length added** (R4?(12+1)) are used to convert the number of columns in a row of graphics out to the data length that must be passed to the printer.

- For Epson printers, you specify a line of graphics by saying

  27, '*', *graphics mode*, *number of columns*

  Since you always pass the number of columns, the Data length multiplier is 1 regardless of whether the printer is 8 pin, 24 pin or 48 pin. Because the graphics mode precedes the data length, there is no need to include it in the data length, and so the Data length added is 0.

- For IBM printers, you specify a line of graphics by saying

  27, '[', 'g', *number of bytes*, *graphics mode*.

  An 8 pin printer requires 1 byte of data per column, a 24 pin printer requires 3 bytes per column, and a 48 pin printer requires 6 bytes per column. The Data length multiplier is respectively 1, 3 or 6. Because the graphics mode follows the data length, it must be included in the data length, and so the Data length added is 1.

In both cases the data length (after manipulation) is sent out as a 2 byte binary number, low byte first then high byte. For example:

27, '*', *graphics mode*, *low byte*, *high byte*

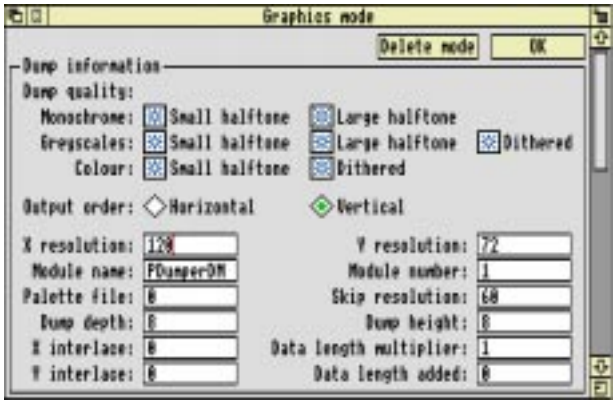
where *low byte* × 256 + *high byte* = *size*.

Both schemes are sensible; they are just different ways of counting how much data there is in the graphics line.

Colour printing

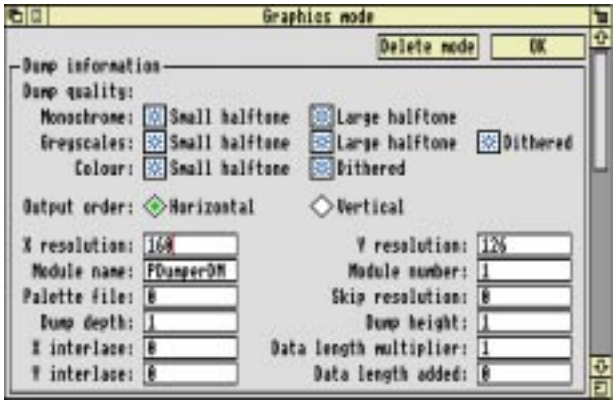
For colour ribbon printing, the contents of a '(dump depth)' are different. This is described below after the relevant PrintEdit fields have been described.

For colour printing, consider the Epson EX-800 at 120 by 72 dpi. The **Dump information** (which is fairly obvious and therefore needs no description) is as follows:



Integrex printers

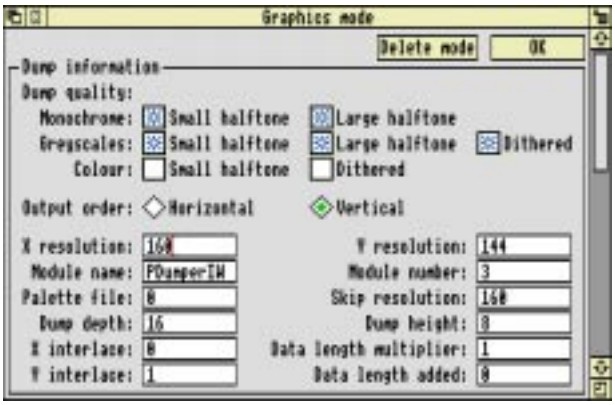
The **Dump information** at 160 by 126 dpi for an Integrex printer is:



The Integrex dumper is combined in PDumperDM with the generic dot-matrix dumper. Consequently the **Module name** and **Module number** remain as **PDumperDM** and **1** respectively. However, as noted above, the **Output order** must be Horizontal for Integrex printers to enable the correct parts of PDumperDM. The dumper also relies on the **Dump depth** and **Dump height** both being set to 1.

ImageWriter printer

The **Dump information** at 160 by 144 dpi for an ImageWriter printer is:

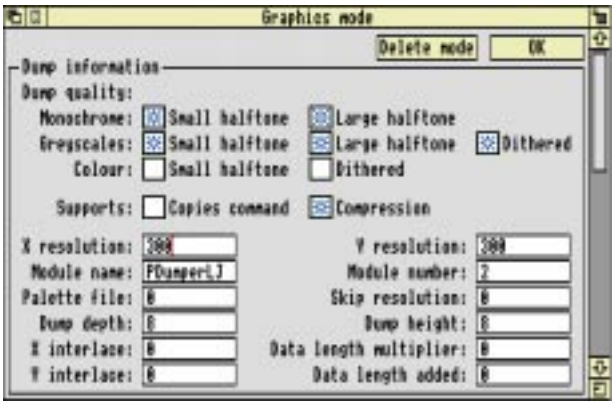


For an ImageWriter, the **Module name** and **Module number** are **PDumperIW** and **3** respectively; although it uses the same back end as generic dot-matrix printers, it requires its own dumper. The reasons for this are outlined on page 3-730.

ImageWriter printers need no extra data after the length and so have a **Data length multiplier** of 1 and a **Data length added** of 0. The **Skip resolution** is usually the same as the **X resolution**.

HP LaserJet compatible printers

The **Dump information** for the HP DeskJet+ at 300 by 300 dpi is:



Most of these fields must have these values, otherwise PDumperLJ will not work. The current version of PDumperLJ has no support for colour printing, hence you must not select the colour options.

The **Printer class** of **lj** (set in the main window) causes PrintEdit to show the **Supports** options, rather than the **Output order** shown by dp class printers:

- **Copies command** should be set if your printer supports the multiple copies command (page printers such as the LaserJet II do, whereas DeskJets usually don't)
- **Compression** should be set if it supports compression mode 2 (older printers don't).

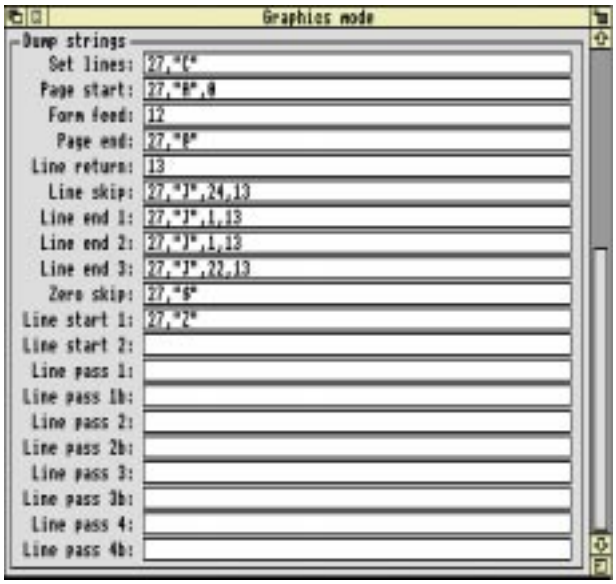
These flags are passed to PDumperReason_SetDriver in R5; if bit 0 (LSB) is set, then **Copies command** was selected; if bit 1 is set, then **Compression** was selected.

There is nothing else you can usefully do with this window when the printer definition file is used in conjunction with the RISC OS 3 (version 3.10) PDumperLJ.

Graphics mode: Dump strings

General points, and Epson and IBM compatible printers

The second part of the **Graphics mode** window gives the **Dump strings**:



These are the codes sent to the printer to tell it to perform certain actions. Only the dumper uses these strings, and hence a new dumper can use any of these strings to hold anything. The PrintEdit window reflects the usage that PDumperDM and PDumperIW make of these strings.

Set lines (R4?(12+21)) is the string to define the number of lines per page. The number of lines (R4?(12+22)) is sent as a single byte after the set lines string (eg. 27, ‘C’, 70 when using A4 paper, which is 70 lines long). The length used is the height in lines from the **Text margins** section of the Printers application’s **Paper sizes** window, and is put into the graphics data block by the !Printers.dp back end.

This sequence is not sent if the number of lines is set to 0, since on Epson compatible printers 27, ‘C’, 0 actually means that the next byte sent after the 0 is the length of the paper in inches.

This sequence solves the problem with the RISC OS 2 and RISC OS 3 (version 3.00) printer drivers whereby the printer’s DIP switches had to be set correctly for the paper size you were using. Normally the switches allow only 11" or 12", and since A4 paper is 11.64" long, form feeds between pages on A4 fanfold paper (or any other non standard paper size) fed the paper to the wrong place.

Page start (R4?(12+3)) is the string sent at the start of a page, after the set lines sequence. It is generally used to switch emulations, or to set the line feed pitch to zero. The reason for setting the line feed pitch to zero is to avoid problems with auto line feed on carriage return. With the line feed pitch set to zero, it really doesn’t matter if an auto line feed happens since it will do nothing anyway. Thus graphics printing does not care which way the auto line feed DIP switch is set on the printer, due to the careful construction of the printer definition files.

Setting the line feed pitch to zero (or to the head height of $^{8}/_{72}$", or indeed to any other number) relies on such a change not affecting the page length previously set up by the **Set lines** sequence. (Similarly you must not reset the printer in the **Page start** sequence, or the page length setting will be reset to the default.) Epson and IBM compatible printers convert the page length to an absolute size internally, so they are not affected by a subsequent change in line pitch.

If your printer is affected by a subsequent change in line pitch, then you must not change the pitch in the **Page start** string. You could change it at the start of the **Set lines** string and have a blank **Page start** string, but you would then need the number of lines in the Printers application’s text margins to be correct for both text and graphics spacing, which are seldom the same. Possible work rounds are:

- Setting up two definitions of the same printer, one of which uses a paper size set up for graphics spacing, and the other of which uses a paper size set up for text spacing.

- Setting the conversion from the number of text lines per page to graphics lines per page in your PDumper code – assuming it is fixed.

  For example, text is usually 6 lines per inch, whereas 8 pin graphics is usually 9 lines per inch (72 dpi pitch / 8 dots per line), so the number of graphics lines per page would be 3/2 times the number of text lines per page.

  Similarly, 24 pin graphics are usually 7 1/2 lines per inch (180 dpi / 24 pins), leading to a conversion factor of 5/4.

Doubtless there are other solutions to this problem.

Form feed (R4?(12+36)) is the string sent to the printer to tell it to form feed the paper after each page has been printed. This string is not sent by PDumperDM when the Roll paper feed setting is selected in the Printers application.

Page end (R4?(12+4)) is the string sent at the end of each page, after the form feed string (if present). It is usual to reset the printer (eg 27, '@' for Epson printers) in this string, so that if this page is the last one, the printer is reset to a known state set by the user on the DIP switches. This string must also set the printer 'top of form' (TOF), or the Roll paper feed setting in the Printers application may not work correctly, because the printer may do a form feed of its own when it thinks it has printed a full page. On Epson compatible printers, 27, '@' resets TOF at the same time as resetting the printer to the default settings, but on other printers (eg. IBM compatibles) a separate string is needed to reset TOF.

Line return (R4?(12+5)) moves the print head to the beginning of the current line. Usually this will be a carriage return. This string is used when performing horizontal interlacing and multi pass colour ribbon printing.

Line skip (R4?(12+6)) moves the print head to the beginning of the next line, and is used for skipping entirely blank lines. For example on the EX-800 at 240 by 216 dpi this string feeds the paper by 24/216" and performs a carriage return.

Line end 1 (R4?(12+7)) to **Line end 3** (R4?(12+9)) are the strings sent at the end of each vertical interlace pass. For example on the EX-800 at 240 by 216 dpi, after the first vertical interlace pass **Line end 1** is sent to the printer, which feeds the paper by 1/216" and performs a carriage return. **Line end 2** does the same after the second vertical interlace pass, and then after the final vertical interlace pass **Line end 3** feeds the paper by 22/216" (27, 'J', 22) and performs a carriage return.

Zero skip (R4?(12+10)) is issued to skip leading zeros on graphics data lines, hence optimising out the white section at the left hand edge of the paper. The string is followed by a two byte number (low byte, high byte) of dots (columns) to skip at the skip resolution; for example 27, '$', 1, 97. A small amount of leading zeros (1/6" worth) is left in the graphics data. This is necessary to allow the print head to accelerate up to speed before the pins print anything.

Line start 1 (string offset R4?(12+11)) is the string sent at the beginning of a graphics line. This is followed by the length of the line, reduced to the minimum necessary to represent the data that is to be printed (to avoid sending unnecessary trailing zeros), and with the **Data length multiplier** and **Data length added** applied. The length is then followed by **Line start 2** (R4?(12+12)).

Therefore, for an Epson **Line start 1** is typically 27, '*', *graphics mode*, and **Line start 2** is unused; whereas for an IBM **Line start 1** is typically 27, '[', 'g', and **Line start 2** is *graphics mode*. For more details, see the description of the Data length multiplier and adder on page 3-718.

If you want to use PrintEdit to look at some printer definition files with **Line start 2** sequences present, the IBM Pro-X24E and the Canon BubbleJet are good examples.

Sequence of data output

The sequence of data output for a black and white page to the EX-800 at 240 by 216 dpi is shown below. This is for PDumperDM with the **Output order** set to **Vertical**.

PDumperDM has a system variable associated with it named PDumperDM$Extra. This is currently unused, but is included in the sequence for the sake of completeness.

The notation used is as follows:

- < > is a string from a PrintEdit field
- [ ] is a single byte output by PDumperDM
- ( ) is a sequence of bytes output by PDumperDM
- {length} is a length field output as [count low][count high] by PDumperDM and (four digit decimal number) by PDumperIW.

The sequence for a page is:

(PDumperDM$Extra) *(currently unused)*
<Set lines>[line count]<Page start>
<Line skip> *repeated until last line at top of page that does not print output*
(dump depth) *repeated until last line that prints output to page (ie is non-zero)*
<Form feed><Page end>

Each '(dump depth)' is a complete set of horizontal and vertical interlace passes, and each one occupies the same amount of paper as a '<Line skip>'. The contents of a '(dump depth)' for our example (2 horizontal interlace passes for each of 3 vertical interlace passes) are:

<Zero skip>{length}<Line start 1>{length}<Line start 2>
(graphics data for 0th horizontal pass)<Line return>
<Zero skip>{length}<Line start 1>{length}<Line start 2>
(graphics data for 1st horizontal pass)<Line end 1>

<Zero skip>{length}<Line start 1>{length}<Line start 2>
(graphics data for 0th horizontal pass)<Line return>
<Zero skip>{length}<Line start 1>{length}<Line start 2>
(graphics data for 1st horizontal pass)<Line end 2>

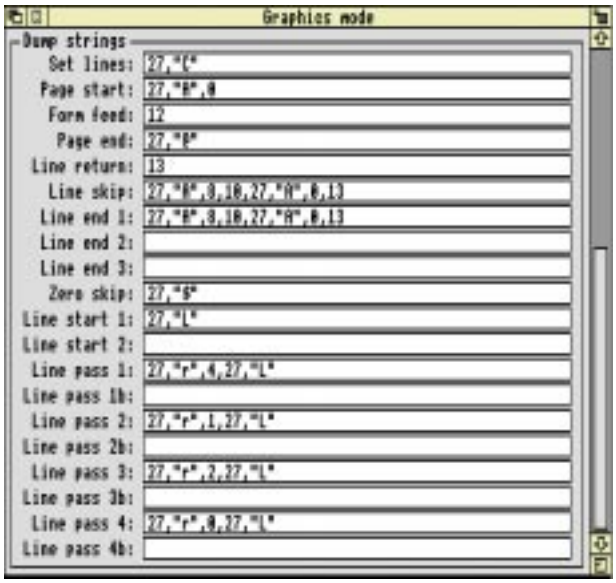
<Zero skip>{length}<Line start 1>{length}<Line start 2>
(graphics data for 0th horizontal pass)<Line return>
<Zero skip>{length}<Line start 1>{length}<Line start 2>
(graphics data for 1st horizontal pass)<Line end 3>

To achieve horizontal interlacing the graphics data for each ‘0th horizontal pass’ goes byte, 0, byte, 0, byte, 0 etc, whereas the graphics data for each ‘1st horizontal pass’ goes 0, byte, 0, byte, 0, byte etc.

Any individual vertical interlace pass which is entirely blank (ie consists of zeros) will not be output at all. However, if any one of a set of horizontal interlace passes is not blank (ie a pair of passes in the above example), all those horizontal passes will still be output. Leading and trailing zero suppression (using Zero skip and data length reduction as mentioned above) are also performed at this level.

Colour printing

The **Dump strings** for colour printing are:



When printing at 120 by 72 dpi in monochrome or grey scale, the strings documented earlier are used. When printing in colour, Line start 1 and Line start 2 are not used. The eight line pass strings are used instead.

Line pass 1 (R4?(12+13)) and **Line pass 1b** (R4?(12+14)) are the equivalents of Line start 1 and Line start 2 for the yellow ribbon pass. The two strings are usually the same as the Line start strings, with the addition that Line pass 1 will select yellow before starting the graphics data.

Similarly, **Line pass 2** (R4?(12+15)) and **Line pass 2b** (R4?(12+16)) are used for the magenta ribbon pass; **Line pass 3** (R4?(12+17)) and **Line pass 3b** (R4?(12+18)) are used for the cyan ribbon pass; and **Line pass 4** (R4?(12+19)) and **Line pass 4b** (R4?(12+20)) are used for the black ('Key black' in CMYK parlance) ribbon pass.

The four colour passes are performed one after the other – yellow, then magenta, then cyan, then black – with a Line return between each pass. If you were to attempt to use interlacing and colour (which is not recommended), you would find that horizontal interlacing occurs before the 4 colour process, and vertical interlacing occurs afterwards over the entire 4 colours. You should use horizontal interlacing in preference to vertical interlacing since horizontal will not contaminate the light ribbon colours, whereas vertical interlacing would by printing the lighter colours on top of the darker ones already present on the paper from the previous interlace pass. For an ink jet this is irrelevant, but interlacing is also usually irrelevant.

Thus a '(dump depth)' for a colour printout (with no interlacing) will be:

<Zero skip>{length}<Line pass 1>{length}<Line pass 1b>
(graphics data for yellow pass)<Line return>
<Zero skip>{length}<Line pass 2>{length}<Line pass 2b>
(graphics data for magenta pass)<Line return>
<Zero skip>{length}<Line pass 3>{length}<Line pass 3b>
(graphics data for cyan pass)<Line return>
<Zero skip>{length}<Line pass 4>{length}<Line pass 4b>
(graphics data for black pass)<Line end 1>

At 240 by 144 dpi (2 horizontal interlace passes for each of 2 vertical interlace passes) a '(dump depth)' would be:

<Zero skip>{length}<Line pass 1>{length}<Line pass 1b>
(graphics data for 0th horizontal yellow pass)<Line return>
<Zero skip>{length}<Line pass 1>{length}<Line pass 1b>
(graphics data for 1st horizontal yellow pass)<Line return>

<Zero skip>{length}<Line pass 2>{length}<Line pass 2b>
(graphics data for 0th horizontal magenta pass)<Line return>
<Zero skip>{length}<Line pass 2>{length}<Line pass 2b>
(graphics data for 1st horizontal magenta pass)<Line return>

<Zero skip>{length}<Line pass 3>{length}<Line pass 3b>
(graphics data for 0th horizontal cyan pass)<Line return>
<Zero skip>{length}<Line pass 3>{length}<Line pass 3b>
(graphics data for 1st horizontal cyan pass)<Line return>

<Zero skip>{length}<Line pass 4>{length}<Line pass 4b>
(graphics data for 0th horizontal black pass)<Line return>
<Zero skip>{length}<Line pass 4>{length}<Line pass 4b>
(graphics data for 1st horizontal black pass)<Line end 1>

<Zero skip>{length}<Line pass 1>{length}<Line pass 1b>
(graphics data for 0th horizontal yellow pass)<Line return>
<Zero skip>{length}<Line pass 1>{length}<Line pass 1b>
(graphics data for 1st horizontal yellow pass)<Line return>

<Zero skip>{length}<Line pass 2>{length}<Line pass 2b>
(graphics data for 0th horizontal magenta pass)<Line return>
<Zero skip>{length}<Line pass 2>{length}<Line pass 2b>
(graphics data for 1st horizontal magenta pass)<Line return>

<Zero skip>{length}<Line pass 3>{length}<Line pass 3b>
(graphics data for 0th horizontal cyan pass)<Line return>
<Zero skip>{length}<Line pass 3>{length}<Line pass 3b>
(graphics data for 1st horizontal cyan pass)<Line return>

<Zero skip>{length}<Line pass 4>{length}<Line pass 4b>
(graphics data for 0th horizontal black pass)<Line return>
<Zero skip>{length}<Line pass 4>{length}<Line pass 4b>
(graphics data for 1st horizontal black pass)<Line end 2>

where <Line end 1> is 27, ‘J’, 1, 13 and <Line end 2> is 27, ‘J’, 23, 13 for the Epson EX-800.

To achieve horizontal interlacing the graphics data for each ‘0th horizontal pass’ goes byte, 0, byte, 0, byte, 0 etc, whereas the graphics data for each ‘1st horizontal pass’ goes 0, byte, 0, byte, 0, byte etc.

Any individual colour pass (Yellow, Cyan, Magenta or Key black) or vertical interlace pass which is entirely blank (ie consists of zeros) will not be output at all. However, if any one of a set of horizontal interlace passes is not blank (ie a pair of passes in the above example), all those horizontal passes will still be output. Leading and trailing zero suppression (using Zero skip and data length reduction as mentioned above) are also performed at this level.

The two most likely scenarios are some of the colour passes being absent (particularly yellow, magenta and cyan being absent with just black present), and an entire ‘(dump depth)’ being absent. Below is an example dump depth at 240 by 144 dpi where only yellow and cyan are in use:

<Zero skip>{length}<Line pass 1>{length}<Line pass 1b>
(graphics data for 0th horizontal yellow pass)<Line return>
<Zero skip>{length}<Line pass 1>{length}<Line pass 1b>
(graphics data for 1st horizontal yellow pass)<Line return>

<Zero skip>{length}<Line pass 3>{length}<Line pass 3b>
(graphics data for 0th horizontal cyan pass)<Line return>
<Zero skip>{length}<Line pass 3>{length}<Line pass 3b>
(graphics data for 1st horizontal cyan pass)<Line end 1>

<Zero skip>{length}<Line pass 1>{length}<Line pass 1b>
(graphics data for 0th horizontal yellow pass)<Line return>
<Zero skip>{length}<Line pass 1>{length}<Line pass 1b>
(graphics data for 1st horizontal yellow pass)<Line return>

<Zero skip>{length}<Line pass 3>{length}<Line pass 3b>
(graphics data for 0th horizontal cyan pass)<Line return>
<Zero skip>{length}<Line pass 3>{length}<Line pass 3b>
(graphics data for 1st horizontal cyan pass)<Line end 2>

You might like to think of printing in black only as a special case of coloured printing, the only difference being that it uses the **Line start** strings rather than the **Line pass** strings.

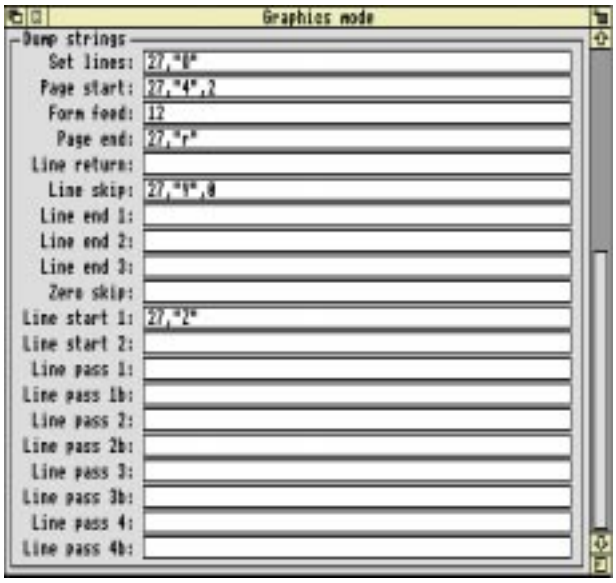
Below is an example entirely blank ‘(dump depth)’ at 240 by 144 dpi:

<Line end 1>
<Line end 2>

The **Line end** sequences are the minimum that it is reasonable to optimise the output down to, because they contain the paper movement commands. It would be possible to spot an entirely empty ‘(dump depth)’ and replace the sequence of **Line end** strings with a single **Line skip**, but in practice it saves too little to be worth the extra code complexity.

Integrex printers

The **Dump strings** for an Integrex printer at 160 by 126 dpi are:



Only the strings present above will be acted on by PDumperDM, which ignores any of the rest that you might fill in. The strings present have the same broad meaning as they did for Epson and IBM printers, and are even output in the same order. The subtle detail is slightly different though.

Set lines, **Form feed** and **Page end** are exactly as for Epson and IBM compatible printers, as is the **Page start** string, save that it is used to specify the vertical resolution.

Line skip is strictly speaking the same, but since it skips a '(dump depth)', and the Dump depth is always set to 1 for Integrex printers, the actual amount of paper skipped is a lot less: in fact a single printer pixel row, since the Integrex is effectively a one pin printer.

Line start 1 is also the same in a strict sense, since it is output before the data. The main data sequence for Integrex printers is the same as for other printers:

(PDumperDM$Extra) *(currently unused)*
<Set lines>[line count]<Page start>
<Line skip> *repeated until last line at top of page that does not print output*
(dump depth) *repeated until last line that prints output to page (ie is non-zero)*
<Form feed><Page end>

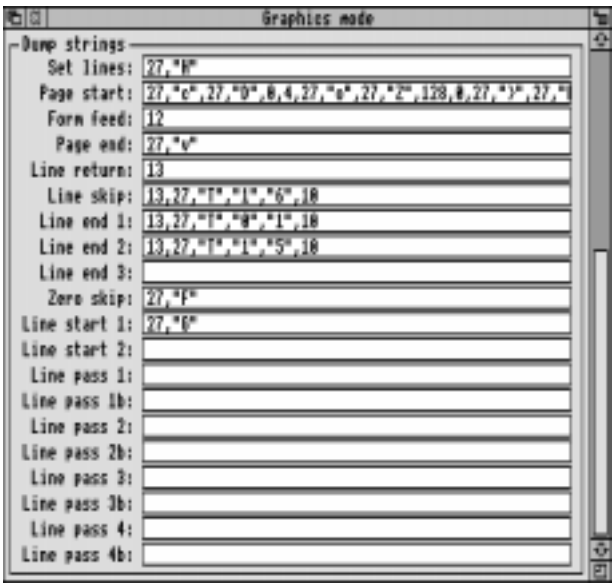
but each '(dump depth)' is substantially different, consisting of a single horizontal printer pixel row:

<Line start 1>[*n*](*n* bytes red data)(*n* bytes green data)(*n* bytes blue data)

The printer merges the red, green and blue data to form a single raster line of CMYK dots, which it then prints. The MSB of a single byte of raster data is printed first (ie on the left).

ImageWriter printer

The **Dump strings** at 160 by 144 dpi for an ImageWriter printer are:



All of these are used in a similar way as for Epson and IBM printers, as is the PDumperIW$Extra system variable (again, currently unused). PDumperIW also follows the same sequence of data output as PDumperDM (see page 3-724), However, there are differences in the format the data in each '(dump depth)' takes – which is why the ImageWriter requires its own dumper. In particular:

- The ImageWriter requires the bits of graphics data to be output in the opposite order vertically to an Epson/IBM compatible printer; ie the LSB (bit 0) is at the top of the print head, and the MSB (bit 7) is at the bottom.

- It requires all the {length} parameters given on page 3-724, and some others, to be passed as four digit decimal numbers rather than as two byte binary numbers.
  For example:
  - The **Dump depth** is given as 27, ‘G’, ‘1’, ‘5’, ‘3’, ‘2’ for 1532 columns.
  - The number of lines is given as the length of the page in $^1/_{144}$ths of an inch eg 27, ‘H’, ‘1’, ‘6’, ‘8’, ‘0’ for $^{1680}/_{144}$".
    In obtaining this number, PDumperIW assumes that there are 6 lines per inch, so the conversion factor is 144/6 = 24. A4 paper is 70 lines long, so $70 \times 24 = 1680$.
  - The **Zero skip** is given as 27, ‘F’, ‘0’, ‘3’, ‘5’, ‘3’.

HP LaserJet compatible printers

For HP LaserJet compatible printers, the options for graphics printing configuration are very limited. The dump strings are unused, the control strings being set in the PDumperLJ module instead. With hindsight we recommend that – if at all possible – you do not follow this approach when writing a dumper. Use the dump strings, even if the contents are used for something totally different to what the PrintEdit names imply they should be for. The flexibility gained by doing this is worth the potential confusion.

Because PDumperLJ has so little flexibility and can only drive standard LaserJet and DeskJet compatible printers, there is no point in describing the data sequence. Either your printer will work with one of the four supplied printer definition files, or it will not work at all and using PrintEdit will not help. See also the file Printers.HP.Read_Me.

System variables

The PDumperLJ$Extra system variable is set by the !Printers.lj back end and is output by PDumperLJ. It contains the control sequence to select either manual feed or auto feed, and the control sequence to select the correct paper size. These strings are read from the !Printers.lj.Resources.Messages file by the !Printers.lj back end, with the following tokens:

- AUTO_FEED
- MANUAL_FEED
- PT_A4
- PT_Letter
- PT_Legal.

The strings after each token may safely be changed if your printer needs a different control sequence, and you may add extra PT_ tokens for any additional paper sizes you may need. For example, if you define a paper size of A3 (Generic LJ) and the control sequence to select A3 paper is:

```
|[&l99A
```

then you would need to add the line:

```
PT_A3:|[&l99A
```

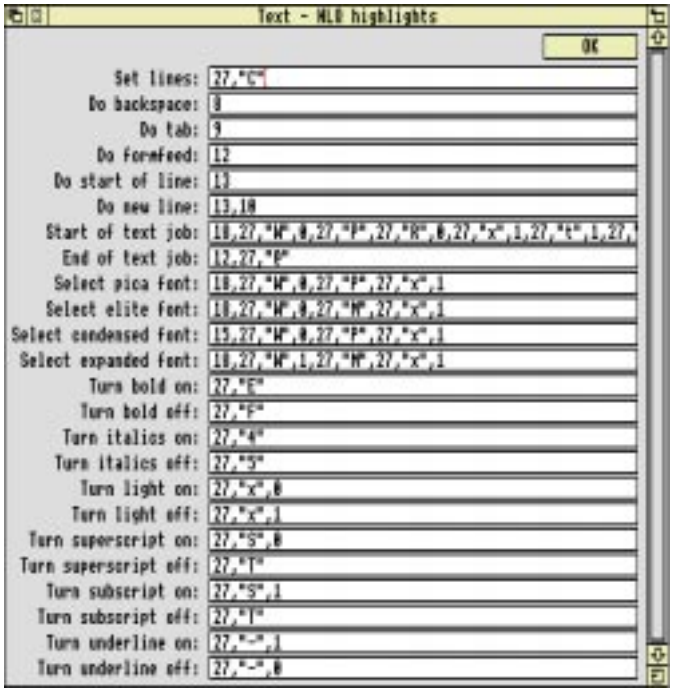
to the file.

The string for PT_A4 is used if there is no token for the paper size selected in the Printers application's printer configuration window.

Text modes

General points, and Epson and IBM compatible printers

Text printing is done by the Printers back end in use (dp, lj or ps) and as such has nothing to do with dumpers. However, there are some points worth mentioning since you will need some text printing definitions to go with a new dumper. **No highlights**, **Draft highlights** and **NLQ highlights** are very similar, so only one will be described in detail. Below is the **Text – NLQ highlights** window for a dp class printer: the Epson.EX-800 file:



Set lines operates in the same way as for graphics printing; it is followed by a byte specifying the length, and is not sent if the length is zero. Unlike graphics printing, the **Set lines** sequence is sent after the **Start of text job** sequence.

Do backspace, **Do tab** and **Do formfeed** are obvious.

Only one of **Do start of line** and **Do new line** is used. If **Print linefeeds** is selected in the Printers application's printer configuration window, then **Do new line** is sent at the end of each text line. If **Print linefeeds** is not selected, then **Do start of line** is sent instead.

Start of text job is sent at the very beginning of the text job. You may assume that the printer has been reset at the end of the preceding text or graphics job, but you should not assume the DIP switch settings. The contents of the **Select pica font** string should always be included in the **Start of text job** string, because the !Printers.dp back end assumes that the printer is set up for pica at the start of the job. On a printer that can do NLQ printing, you should enable or disable NLQ mode appropriately for all three sets of highlights. The full string does not fit into the PrintEdit icon as shown above. It is:

18, 27, 'W', 0, 27, 'P', 27, 'R', 0, 27, 'x', 1, 27, 't', 1, 27, '6'

Splitting this up into components:

- 18, 27, 'W', 0, 27, 'P' selects the pica font and is part of the 'Select pica font' string
- 27, 'R', 0 selects the USA character set
- 27, 'x', 1 turns NLQ on, and is the remainder of the 'Select pica font' string
- 27, 't', 1 sets the top bit set printer characters to the Epson Character Graphics set (the same as IBM Code Page 437) rather than the italics character set
- 27, '6' enables characters 128 to 159 and 255 as printable characters within the Epson Character Graphics set.

Much of this is needed for the character mappings supplied. As you can see, the Start of text job sequence has to get lots of things set up correctly to simplify everything else.

End of text job usually just does a form feed and resets the printer. It is standard Acorn practice to reset dp printers at the end of jobs (text and graphics) but not at the start. One of the reasons behind not doing it at the start of the job is that it lets the user use some of the printer's front panel controls to select various options if he wants, which with luck will remain in effect throughout the print job (although they will probably be turned off by the reset at the end). If the reset were at the start of print jobs, this would not be possible.

Select pica font selects a 10 cpi (characters per inch) printer font. This is the default, and it is assumed that all printers can support it. All font selection strings should ensure they correctly disable anything that any of the other font selection strings enable.

Select elite font selects a 12 cpi printer font. If there is no 12 cpi font then you should select the closest, which will usually be 10 cpi.

Select condensed font selects a 17 cpi printer font, which nearly all printers support. Again you should select the closest size if 17 cpi is not available.

Select expanded font selects a 6 cpi printer font. This is usually achieved by selecting 12 cpi, and turning double width printing on with 27, 'W', 1 (which is why all the other font selection strings turn double width off). If there is no 6 (12) cpi font then you should select the closest size. This usually means selecting 10 cpi and double width printing to give 5 cpi.

Turn bold on/off, **Turn italics on/off**, **Turn superscript on/off**, **Turn subscript on/off** and **Turn underline on/off** are obvious.

Turn light on and **Turn light off** are difficult to support as dot matrix printers generally have no concept of light printing. In NLQ highlights, it is Acorn practice to turn NLQ mode off for light printing, and turn it back on again when normal printing resumes. There could be a conflict if a font selection string is issued half way through the light printing, as the font selection strings usually enable NLQ. This is a specific instance of a general problem with 1st Word Plus file fancy text printing, which is that different styles and effects should be mixed sparingly. For example many printers cannot underline superscripts and subscripts, or print them in bold.

Integrex printers

Text modes on Integrex printers work in the same way as for Epson/IBM compatible printers.

ImageWriter printer

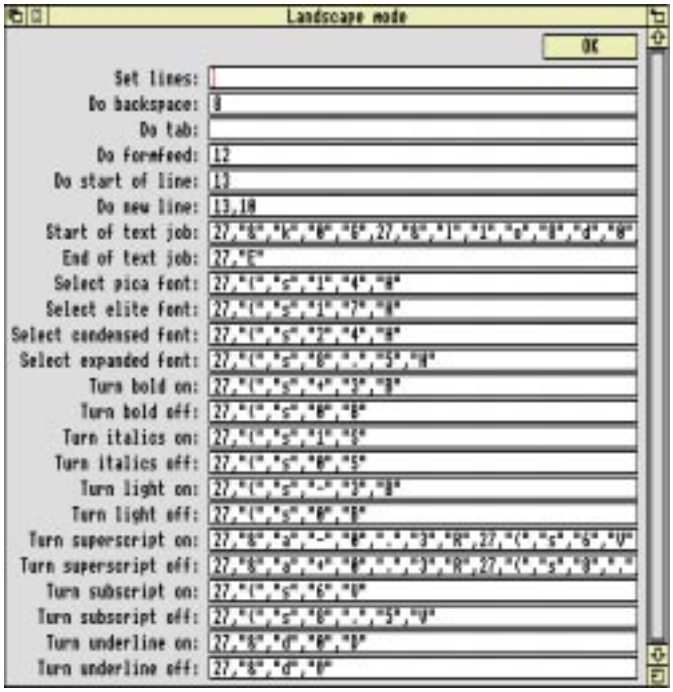
Text modes on the ImageWriter printer works in the same way as for Epson/IBM compatible printers, save that the length is sent as a four digit decimal string using the same units and conversion factor as for graphics printing. This is done by the !Printers.dp back end if the name of the dumper to use is PDumperIW, and was done to avoid having a new printer class and back-end just to handle this one difference in text printing.

HP LaserJet compatible printers

The text highlights and character mappings in all the lj printer class files supplied with RISC OS 3 (version 3.10) are identical. This fortunate state of affairs is because when talking to LaserJet compatible printers, you can tell the printer to do something and if it can’t it is up to the printer itself to decide what compromise to make. This means the output differs between different printers, even though the printer definition files are the same. See the Printers.HP.Read_Me file for more details of this effect.

As mentioned earlier, Portrait mode and Landscape mode highlights are available instead of No, Draft and NLQ highlights for lj class printers. In fact, the strings for Portrait and Landscape mode are nearly all the same, the only difference being that in landscape mode smaller font sizes are specified, and landscape printing is selected. Both Portrait and Landscape mode must be defined in the printer definition file, and they must be used for their intended purpose. This is because the lj back-end assumes much, to the extent that some control sequences are burnt into the code rather than being in the printer definition file; again you should see the Printers.HP.Read_Me file. Some control sequences also come from the !Printers.lj.Resources.Messages file, as described on page 3-732.

Landscape mode in the HP.DeskJet+ file is shown below:



Set lines and **Do tab** are unused, **Do backspace** and **Do formfeed** are obvious, and all of the other strings except **Start of text job** are as for dp class printers above.

The **Start of text job** string has seriously overflowed the icon in the window; use PrintEdit on the DeskJet+ file to see all of it. There is much that is assumed about this string.

At the start of a text job, the lj back-end resets the printer with Esc E, as per Hewlett-Packard guidelines. Next it sends the paper feed (auto or manual) selection sequence and paper size settings from the Messages file, in the same manner as the setting of PDumperLJ$Extra described earlier. Then the actual **Start of text job** string is output, and finally the number of lines per page is set with an Esc &l*xx*F sequence. Thus the position of the **Start of text job** string is fixed, and there are HP guidelines which specify the order in which various things must be set to ensure that the printer chooses the closest match it can manage. The fixed position may impose some limitations on what you can put in the string. As well as getting the order of everything correct, the string must select landscape or portrait orientation as appropriate. It must also define the settings for both the primary and secondary printer fonts: the secondary font is used for page titles (if enabled), and the primary font is used for everything else.

The superscript and subscript strings deserve mentioning. There are no such effects on LaserJet compatible printers. The **Turn subscript on** string merely reduces the height of the text. The **Turn superscript on** string moves the baseline of the text up, and reduces the height of the text. Quite a few printers do not perform these actions well.

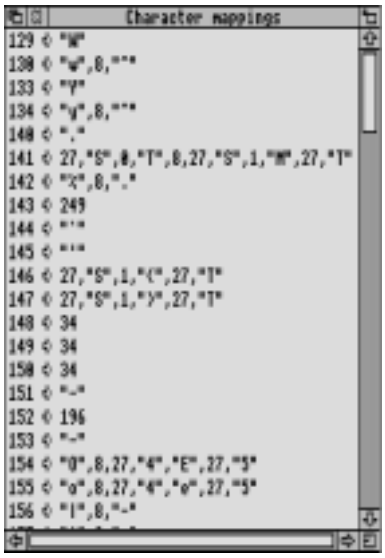
There is actually a control sequence for requesting light printing on LaserJets (used above), but very few printers are capable of doing it. This feature is only really present because it is in the 1st Word Plus file format.

Character mappings

General points, and Epson and IBM compatible printers

Editing **Character mappings** allows a character from the Acorn character set currently in use to be converted to the same character in the printer's character set. For example on UK systems the mappings provided are for Acorn Extended Latin1, whereas for a Turkish system a new set of printer definition files would have to be supplied containing Latin3 character mappings.

Part of the character mappings for the Epson.EX-800 file are show below:



These mappings are used to convert from the Acorn character set in use to the printer character set. For example to print character 130 ('ŵ' – w circumflex) a 'w' is printed, then the print head is backspaced, and then a '^' (circumflex) is printed on top of the 'w'.

The character mappings in a given printer definition file are specific to a certain RISC OS alphabet. For instance, the character mappings in the files supplied with RISC OS 3 (version 3.10) are all for Acorn Extended Latin1 since that is the correct alphabet for the UK territory. A localised system would come with a new set of printer definition files containing mappings for the correct alphabet (eg. Latin3 for a Turkish system). All characters from 32 to 255 can have mappings, even though none of the supplied files have mappings for characters less than 128. If there is no mapping, the character is sent straight through to the printer unaltered. The USA character set is selected for Epson compatible printers in the **Start of text job** string because the USA printer character set matches exactly characters 32 to 126 in Acorn Extended Latin1.

A mapping should take up only one character position on the paper. For example character 154 (‘Œ’ - OE diphthong) might reasonably be mapped as an ‘O’ and an ‘E’ next to each other ie. ‘OE’. However, this takes up two character spacings on the paper, which is not allowed. This is because the Printers application’s and the back end’s idea of where the output had got to would not match what was on the paper. Also 1st Word Plus (and hence 1st Word Plus fancy text files) assumes that all characters only take one character spacing on the paper. Finally, with the example given above you can’t tell from the paper copy whether the file has an ‘Œ’ diphthong in it or an ‘OE’ pair, hence it is not an accurate representation of the file. An ‘O’ and an ‘E’ printed on top of each other doesn’t look very good (although on the EX-800 the ‘E’ is italic which improves matters) but at least it is unique and distinguishable from all other characters.

Earlier we mentioned that text highlights sometime clash with each other. They also clash with some of the character mappings. For example character 141 (‘™’ – the trademark symbol) is obtained by printing a superscript ‘T’ and a subscript ‘M’ on top of each other. This disables subscript (and superscript) at the end of the sequence. So if superscript or subscript is in effect when the ‘™’ character is printed, they will be switched off on the printer afterwards even though they should still be turned on. The italic ‘E’ in the ‘Œ’ example used earlier would cause the same problem with italics. If these clashes are a problem for you, then you can change the character mappings to try to avoid them. As a result some of your mappings will not be as good, and you need to decide what compromise to arrive at.

You must map every character in the Acorn character set in use, regardless of how poor a mapping you can provide. The Printers.Generic.Text file maps each top bit set Acorn Extended Latin1 character to a single top bit clear character, for use with line printers and other extremely primitive print mechanisms. Use the mappings in this file as the absolute minimum representations.

The printer definition files supplied with RISC OS 3 (version 3.10) contain three main groups of character mappings for dp printers. These groups are based on what character sets in the printer are used. All files make use of styles and backspace over-printing where possible to supplement these printer character sets.

The first and simplest is the ‘FX-80’ group (see the Epson.FX-80 file). This uses the basic top bit clear USA character set, and some switches to other countries’ character sets for extra characters. The USA character set is selected in the **Start of text job** sequence (ESC, ‘R’, 0).

The second is the ‘ESC t 1’ or ‘Code Page 437’ group (see the Epson.EX-800 file). This uses the basic top bit clear USA character set, the Epson Character Graphics top bit set characters (the same as IBM Code Page 437), and some switches to other countries’ character sets – though less so than in the ‘FX-80’ group. The USA character set (ESC, ‘R’, 0) and the Epson Character Graphics set (ESC, ‘t’, 1 ESC, ‘6’) are selected in the **Start of text job** sequence.

The third and best is the ‘Code Page 850’ group (see the IBM.Pro-X24E file). This uses the IBM Code Page 850 character set, which has a perfect single character mapping for all characters except the Acorn extensions to Latin1 between characters 128 and 159. Code Page 850 (Esc, ‘[’, ‘T’, 4, 0, 0, 0, 3, ‘R’ Esc, ‘6’) is selected in the **Start of text job** sequence. Also, if the printer is not already in IBM emulation mode this should be selected; for example this is done in the Citizen.Swift-24 file.

Obviously there are deviations within each group, and there are in fact too many to go into here. However, it is useful to recognise what the major groupings are, pick one, and then tweak it to fit your printer. There are also some files that are not in any of the groupings; for example the Generic.Text and Star.DP-510 mappings.

Integrex printers

The character mappings for the Integrex differ from the classes given above. You can view them from PrintEdit; there are no unusual features worthy of mention here.

ImageWriter printer

The character mappings for the ImageWriter differ from the classes given above. You can view them from PrintEdit; there are no unusual features worthy of mention here.

HP LaserJet compatible printers

The complete character mappings for the DeskJet+ file are:

Only the Acorn extensions to Latin1 (characters 128 to 159) need to be mapped, because the ‘Start of text job’ string selects the ECMA-94 character set, which is the same as ISO Latin1. Character mappings 140 and 150 are of interest. Character 140 (‘…’ – the ellipsis symbol) is obtained by moving the text baseline down and printing the ‘¨’ (diaeresis) character, and then, of course, moving the baseline back. This gives two dots at the right height, which is better than one, although there should be three. Similarly character 150 ‘„’ (bottom double quote) is obtained by moving the baseline down and printing a normal double quote. These two examples show that it pays to be imaginative when dealing with a flexible printer like a LaserJet compatible. Such tricks are impossible on a dot matrix printer.

68 MakePSFont

Introduction

The MakePSFont module provides a a SWI used by PDriverPS to make PostScript fonts available to printers. It is a private interface between the Printers application and PDriverPS. You must not use it from your own applications; it is only of relevance to anyone wishing to replace the current PostScript printer drivers. See the section entitled *Extending the printing system* on page 3-602.

SWI calls

MakePSFont_MakeFont (SWI &43440)

This call is for internal use only; you must not use it in your own code.

It is not available in RISC OS 2, nor in RISC OS 3 (version 3.00).

Part 10 – Internationalisation

69 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.

MessageTrans is not available in RISC OS 2.

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).

Global messages

If MessageTrans is passed a null pointer to a file descriptor, it uses a file of global 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 global 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 *xxx* 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_MessageFileClosed

If a messages file is held in ResourceFS, MessageTrans does not make a copy of the message file, but instead directly accesses the file. Service_MessageFileClosed is used to notify MessageTrans that the ResourceFS file has been removed for one reason or another.

Service Call

Service_MessageFileClosed (Service Call &5E)

Message files have been closed

On entry

R0 = 0, or – under RISC OS 3 (version 3.00) only – the 4-word data structure passed to MessageTrans_OpenFile
R1 = &5E (reason code)

On exit

All registers are preserved

Use

This call is issued by MessageTrans as a broadcast to warn that all message files have been closed. You must not claim it.

If your application has any direct pointers into message data, 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 is open, and delete it using Wimp_CreateMenu (–1) if so.

You only need to act on this service call if you are using direct pointers into the message file data. Otherwise, the MessageTrans module will make a note in the file descriptor that the file has been closed, and simply re-open it when you next call MessageTrans_Lookup or MessageTrans_MakeMenus on that file.

We recommended that you don't use direct pointers into message file data (eg indirected icons with MessageTrans_MakeMenus) if your application cannot trap service calls. You can still use such indirected icons, if you provide a buffer pointer in R2 on entry to MessageTrans_OpenFile (so that the message file data is copied into the buffer).

Under RISC OS 3 (version 3.00) this service call is instead issued for each open message file that 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. The file is identified by the 4-word data structure passed to MessageTrans_OpenFile. If you recognise this value, you should claim the service call and act accordingly.

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 (page 3-752)

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 (page 3-750), MessageTrans_CloseFile (page 3-759)

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,
or 0 to use global messages file (see page 3-746)
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 preserved
R1 = pointer to terminator of token
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 global messages file; see the section entitled *Global messages* on page 3-746.

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 3-746 for further details of the file format used to hold message tokens and their corresponding strings.

Related SWIs

MessageTrans_ErrorLookup (page 3-762), MessageTrans_GSLookup (page 3-764)

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,
or 0 to use global messages file (see page 3-746)
R1 = pointer to menu definition (see below)
R2 = pointer to buffer to hold menu structure
R3 = size of buffer

On exit

R0, R1 preserved
R2 = pointer to end of constructed menu structure
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 3-153).

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 3-746 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 3-153.

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 (page 3-752)

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 3-746 for further details of the file format used to hold message tokens and their corresponding strings.

You cannot pass R0 = 0 to enumerate the global message tokens.

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,
or 0 to use global messages file (see page 3-746)
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 global messages file; see the section entitled *Global messages* on page 3-746.

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. To test for successful error lookup, you should check the error number returned in the error block.

If R2 is 0 on entry, MessageTrans will use one of its internal buffers for the result. There are 10 buffers for foreground processes and 2 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 3-746 for further details of the file format used to hold message tokens and their corresponding strings.

Related SWIs

MessageTrans_Lookup (page 3-754), MessageTrans_GSLookup (page 3-764), MessageTrans_CopyError (page 3-766)

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,
or 0 to use global messages file (see page 3-746)
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

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not 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 global messages file; see the section entitled *Global messages* on page 3-746. 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 3-746 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 (page 1-466), MessageTrans_Lookup (page 3-754), MessageTrans_ErrorLookup (page 3-762)

Related vectors

None

MessageTrans_CopyError (SWI &41508)

Copies an error to one of the MessageTrans internal buffers

On entry

R0 = pointer to error block (word aligned)

On exit

R0 = pointer to error buffer used

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 copies an error to one of the MessageTrans internal buffers. There are 10 buffers for foreground processes and 2 for calls made from within IRQ processes. MessageTrans will cycle between these buffers.

Related SWIs

MessageTrans_ErrorLookup (page 3-762)

Related vectors

None

70 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.

You will find tables of the various character sets, and details of keyboard shortcuts both for selecting keyboard layouts and for entering top-bit-set characters, in *Table D: Character sets* on page 4-569.

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*
24	Russia	Cyrillic
25	Russia2	Cyrillic2

26	Israel	Hebrew
27	Mexico	Latin1
28	LatinAm	Latin1
80	ISO1	Latin1
81	ISO2	Latin2
82	ISO3	Latin3
83	ISO4	Latin4

Countries 20 - 23 and 81 - 83, although allocated, are not supported by this module's * Commands.

The keyboard ID byte read by country 0 ('Default') has changed in meaning between RISC OS 2 and 3; it now represents the keyboards physical layout. Consequently you should no longer use this value.

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
105	Cyrillic
106	Arabic
107	Greek
108	Hebrew
120	Cyrillic2

Alphabets 106, although allocated, is not supported by this module's * Commands.

Alphabet

OS_Byte 71 (page 3-780) 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 (page 3-773) 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 3-767 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, most of the characters &80-&8B 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 (page 3-782)

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 when being set; eg to set keyboard 3, you must pass 131 in R1. However, when being read its actual value is returned.

Related SWIs

OS_Byte 70 (page 3-779)

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 (page 3-779)

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, Israel, Italy, LatinAm, Master, Mexico, Norway, Portugal, Russia, Russia2, Spain, Sweden, and UK. A list of parameters can be obtained with the *Countries command.
`alphabet_name`	Valid alphabets are currently BFont, Cyrillic, Cyrillic2, Hebrew, Latin1, Latin2, Latin3, Latin4 and Greek. A list of parameters can be obtained with the *Alphabets command.

Use

*Alphabet selects an alphabet, setting the character set according to the country name or alphabet name.

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. However, since under RISC OS 3 the keyboard ID is used to represent the physical layout of the keyboard rather than the country for which it is layed out, we recommend you don't use this option. (Standard Archimedes keyboards all have a keyboard ID of 1, which would select the UK alphabet; the A4 internal keyboard and PC external keyboard each have a keyboard ID of 2, which would select the French alphabet.)

With no parameter, this command displays the currently selected alphabet.

Example

```
*Alphabet Latin3
```

Related commands

*Alphabets

Related SWIs

OS_Byte 71 (page 3-780)

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  Cyrillic
Greek   Hebrew  Cyrillic2
```

Related commands

*Alphabet

Related SWIs

OS_Byte 71 (page 3-780)

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, Israel, Italy, LatinAm, Master, Mexico, Norway, Portugal, Russia, Russia2, 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 countries other than the UK you will also need to load a relocatable module that defines the keyboard layout. (The IntKey application on the RISC OS 3 Support Disc provides several drivers.)

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. However, since under RISC OS 3 the keyboard ID is used to represent the physical layout of the keyboard rather than the country for which it is layed out, we recommend you don't use this option. (Standard Archimedes keyboards all have a keyboard ID of 1, which would select the UK alphabet and layout; the A4 internal keyboard and PC external keyboard each have a keyboard ID of 2, which would select the French alphabet and layout.)

Example

```
*Configure Country Italy
```

Related commands

*Country, *Countries

Related SWIs

OS_Byte 70 (page 3-779), OS_Byte 240 (page 3-782)

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  Russia  Russia2
Israel  Mexico  LatinAm ISO1
```

Related commands

*Configure Country, *Country, *Alphabet, *Alphabets, *Keyboard

Related SWIs

OS_Byte 70 (page 3-779), OS_Byte 240 (page 3-782)

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, Israel, Italy, LatinAm, Master, Mexico, Norway, Portugal, Russia, Russia2, 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 countries other than the UK you will also need to load a relocatable module that defines the keyboard layout. (The IntKey application on the RISC OS 3 Support Disc provides several drivers.) 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. However, since under RISC OS 3 the keyboard ID is used to represent the physical layout of the keyboard rather than the country for which it is layed out, we recommend you don't use this option. (Standard Archimedes keyboards all have a keyboard ID of 1, which would select the UK alphabet and layout; the A4 internal keyboard and PC external keyboard each have a keyboard ID of 2, which would select the French alphabet and layout.)

With no parameter, this command displays the currently selected country.

Example

```
*Country Italy
```

Related commands

*Configure Country, *Countries, *Alphabet, *Alphabets, *Keyboard

Related SWIs

OS_Byte 70 (page 3-779), OS_Byte 240 (page 3-782)

Related vectors

None

*Keyboard

Selects the appropriate 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, Israel, Italy, LatinAm, Master, Mexico, Norway, Portugal, Russia, Russia2, 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 countries other than the UK you will also need to load a relocatable module that defines the keyboard layout. (The IntKey application on the RISC OS 3 Support Disc provides several drivers.)

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. However, since under RISC OS 3 the keyboard ID is used to represent the physical layout of the keyboard rather than the country for which it is layed out, we recommend you don't use this option. (Standard Archimedes keyboards all have a keyboard ID of 1, which would select the UK layout; the A4 internal keyboard and PC external keyboard each have a keyboard ID of 2, which would select the French layout.)

With no parameter, this command displays the currently selected keyboard layout.

Example

```
*Keyboard Denmark
```

Related commands

*Country

Related SWIs

OS_Byte 71 (page 3-780)

Related vectors

None

71 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 character (such as upper/lower case, control characters, and accented 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 3-800 and the section entitled *Territory module SWIs* on page 3-813, 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 3-854), 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. In such cases you may consider supplying a single generic territory with an extra configuration option.

For example, to support the whole of the USA you might think you would need five territories identical in every respect, except for their use of time zones. Instead, you can provide a single USA territory that uses a command to configure the correct time zone. Because supporting different time zones is so common a requirement, the Territory module supplies the *Configure TimeZone command to do so.

For other such minor differences, you can provide your own configuration commands with your territory. For example, an Irish territory might have a configuration command to choose the currency symbol used (‘£’ for Northern Ireland, or ‘Ir£’ for Éire).

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 4-553.

Territory numbers and names

Territory numbers and names must be allocated by Acorn; see the section entitled *Territory, country and alphabet numbers and names* on page 4-553.

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

This call is issued by the territory manager when it has started, announcing 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.

Service_TerritoryStarted (Service Call &75)

New territory starting

On entry

R1 = &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 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 3-795, and **Configure Territory* on page 3-854).

Related SWIs

Territory_NumberToName (page 3-804), Territory_NameToNumber (page 3-849)

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 table 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 table pointed to by R1 should contain 43 entries, each of which is a pointer to code to handle one of the SWIs that – although in the territory manager SWI chunk – are actually handled by a territory module. The first entry corresponds to the SWI &4304A, the second to SWI &4304B, and so on through to the last entry which is for SWI &43074. The entry and exit conditions for the SWI handler are as follows:

On entry

R0 - R9	preserved from original entry to the SWI
R11	SWI handler number (0 - 42: ie offset within table)
R12	value of R2 passed to Territory_Register
R13	pointer to supervisor stack

On exit

R0 - R9	return values for the SWI

For a full description of the SWIs themselves, see the section entitled *Territory module SWIs* on page 3-813.

Some of these SWI numbers (currently from &43062 upwards) are reserved for future expansion, and so you obviously cannot implement them. The code for such SWIs must return an error, **not** just return directly. The error number **must** be &43040 (for all territories), and the text should be 'Unknown Territory SWI' (or a translation to your territory's language and alphabet).

Related SWIs

Territory_Deregister (page 3-803)

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

Territory_Register (page 3-801)

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

Territory_NameToNumber (page 3-849)

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

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 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

Territory_Alphabet (page 3-829)

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 (null terminated)
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

Territory_ReadTimeZones (page 3-814)

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
32	day of year

Related SWIs

Territory_ConvertTimeToOrdinals (page 3-823)

Related vectors

None

Territory_ConvertTextToString (SWI &73075)

Not yet implemented

On entry

—

On exit

All registers preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call is not yet implemented, and returns immediately to the caller, with all registers preserved.

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 a territory module passes by calling Territory_Register when it starts, or when the territory manager restarts. If you are writing your own territory module, you should see the documentation of Territory_Register on page 3-801.

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 3-795, and **Configure Territory* on page 3-854) 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

Territory_ReadCurrentTimeZone (page 3-809)

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 1-412.

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

Territory_ConvertStandardDateAndTime (page 3-817),
Territory_ConvertStandardDate (page 3-819),
Territory_ConvertStandardTime (page 3-821)

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

Territory_ConvertDateAndTime (page 3-815),
Territory_ConvertStandardDate (page 3-819),
Territory_ConvertStandardTime (page 3-821)

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

Territory_ConvertDateAndTime (page 3-815),
Territory_ConvertStandardDateAndTime (page 3-817),
Territory_ConvertStandardTime (page 3-821)

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

Territory_ConvertDateAndTime (page 3-815),
Territory_ConvertStandardDateAndTime (page 3-817),
Territory_ConvertStandardDate (page 3-819)

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	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
32	day of year

Related SWIs

Territory_ConvertTimeToOrdinals (page 3-823)

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	centiseconds
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

Territory_ConvertDateAndTime (page 3-815)

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	centiseconds
4	seconds
8	minutes
12	hours (out of 24)
16	day number in month
20	month number in year
24	year number

Related SWIs

Territory_ConvertTimeToUTCOrdinals (page 3-810),
Territory_ConvertTimeToOrdinals (page 3-823)

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

Territory_SelectAlphabet (page 3-807)

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

Territory_AlphabetNumberToName (page 3-806),
Territory_SelectAlphabet (page 3-807), Territory_Alphabet (page 3-829)

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 white 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 or 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.

The C library uses this SWI to build tables for the *isalnum*, *isalpha*, *iscntrl*, *isgraph*, *islower*, *isprint*, *ispunct*, *isspace* and *isupper* functions/macros. If you're programming in C you can instead use these functions/macros to test a character's properties, provided you have previously called the *setlocale* function.

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.

The C library uses this SWI to build tables for the *tolower* function/macro. If you're programming in C you can instead use *tolower* to perform lower case conversion, provided you have previously called the *setlocale* function.

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.

The C library uses this SWI to build tables for the *toupper* function/macro. If you're programming in C you can instead use *toupper* to perform upper case conversion, provided you have previously called the *setlocale* function.

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 Processor 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.

The C library function *srtrcoll* calls this SWI. If you're programming in C you can instead use *srtrcoll* to compare two strings, provided you have previously called the *setlocale* function.

Related SWIs

None

Related vectors

None

Territory_ReadSymbols (SWI &4305E)

Returns various information telling you how to format numbers

On entry

R0 = territory number, or –1 to use current territory
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.

The C library function *localeconv* calls this SWI. If you're programming in C you can instead use *localeconv* to return formatting information, provided you have previously called the *setlocale* function.

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 number of the given territory

On entry

R0 = territory number, or –1 to use current territory
R1 = pointer to territory name in the alphabet of the territory pointed to by R0 (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 accepts the name of one territory in the language of another territory (probably – but not necessarily – different). It returns the number of the named territory.

Related SWIs

None

Related vectors

None

Territory_TransformString (SWI &43061)

Transforms a string to allow direct territory independent string comparison

On entry

R0 = territory number, or –1 to use current territory
R1 = pointer to buffer to hold transformed string
R2 = pointer to source string (null terminated)
R3 = length of buffer to hold transformed string

On exit

R0 = length of transformed string (excluding terminating null)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call transforms the string pointed to by R2 and places the resulting string into the buffer pointed to by R1. The transformation is such that if a byte by byte comparison is applied to two transformed strings, then the strings will compare less than, equal to or greater than (as though Territory_Collate had been applied to the original strings).

If you call this SWI with R3 set to 0 on entry, R1 may be a null pointer. On exit R0 will contain the length of the transformed string, without altering the buffer. You may then set up a buffer of the required size (remembering to allow for the terminating null) before again calling this SWI to place the string in the buffer.

If R0 on exit is ≥ R3 on entry (ie the string was too long to fit in the buffer) the contents of the buffer are undefined, but writing will not have occurred beyond the bounds of the buffer, since this call never writes more than R3 bytes.

If copying takes place between strings that overlap the behaviour is undefined.

The C library function *strxfrm* calls this SWI. If you're programming in C you can instead use *strxfrm* to transform strings, provided you have previously called the *setlocale* function.

This call is not available in RISC OS 3 (version 3.00), and leaves the string unaltered in RISC OS 3 (version 3.10).

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 to supply an alternative name for this command. For example, if the UK territory module is registered, the command *Configure BST (short for British Summer Time) has the same effect as *Configure DST. This alternative name is also used by the *Status command (see page 1-407).

Example

```
*Configure DST
```

Related commands

*Configure NoDST, *Configure TimeZone

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 to supply an alternative name for this command. For example, if the UK territory module is registered, the command *Configure GMT (short for Greenwich Mean Time) has the same effect as *Configure NoDST. This alternative name is also used by the *Status command (see page 1-407).

Example

```
*Configure NoDST
```

Related commands

*Configure DST, *Configure TimeZone

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. Use this command with caution; if you set a territory that is unavailable your computer will not start, and so you will have to reset your CMOS RAM.

Example

```
*Configure Territory UK
```

Related commands

*Territories

Related SWIs

None

Related vectors

None

*Configure TimeZone

Sets the configured local time offset from UTC

Syntax

```
*Configure TimeZone [+|-]hours[:minutes]
```

Parameters

hours	offset from UTC in hours
minutes	offset from UTC in minutes

Use

*Configure TimeZone sets the configured local time offset from UTC. You should use this command to configure the local time on your machine rather than changing the system clock as was necessary for RISC OS 2. Using the *Configure TimeZone command will ensure that (since the system clock on all machines will represent UTC) timestamps on files will be valid across machines, networks will work correctly across time zones and electronic mail will be correctly timestamped.

The time offset must be in the range –13:45 to +13:45, and must be an exact multiple of 15 minutes.

Example

`*Configure TimeZone 9:30`	*Northern Territory, Australia*
`*Configure TimeZone -5`	*Eastern USA*

Related commands

*Configure DST, *Configure NoDST

Related SWIs

Territory_ReadTimeZones (page 3-814)

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