# Contents

# User Interface Toolbox

Published by Acorn Computers Technical Publications Department.



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

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

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

Acorn supplies its products through an international distribution network. Your supplier is available to help resolve any queries you might have.



. Published by Acorn Computers Limited
ISBN 1 85250 165 0
Part number 0484,231
Issue 1, December 1994

# 1 Introduction to the Toolbox

This chapter is intended to give the reader an overview of the RISC OS Toolbox, and to introduce the concepts used throughout the rest of this manual.

## Introduction

The Toolbox was designed with the following goals:

- to facilitate writing consistent, high-quality desktop applications under RISC OS 3.10 and later
- to encourage the writing of applications whose user interface complies with the RISC OS 3 Style Guide
- to be easy to learn
- to be language-independent
- to make it no harder to do operations which can currently be done using the Wimp.

The Toolbox has the following characteristics:

- it is structured as a set of RISC OS relocatable modules
- it will only run on RISC OS 3.10 or later
- it does not directly call back to code in the client application
- it is SWI-driven
- it can be used from C, C++, BASIC or Assembler with equal ease
- communication back to the client application is via events
- the client application does not have direct access to data structures maintained by the Toolbox
- it uses a new resource file format to hold templates for the user interface objects which the application will use at run-time.

Note: The appendix *Support for RISC OS 3.10* on page 505 describes support for RISC OS 3.10 machines.

### Installing C/C++

The instructions for installing Acorn C/C++ are in the chapter *Installing Acorn C/C++* on page 7 of the *Desktop Tools* manual.

## Terminology

The following terms are used throughout this manual:

| Term | Meaning |
|---|---|
| Class | A data type, together with a definition of the operations which can be performed on that data type |
| Client application | A piece of software which uses the Toolbox |
| Colours | Refers either to desktop colours (in the range 0-15), or to an RGB colour (represented by one word as 0xbbggrr00) |
| Dialogue box | A window which contains gadgets, and which is typically used to carry out a 'dialogue' with the user, ending in the user either cancelling the dialogue, or confirming that they want to apply the options indicated by the current dialogue state |
| Method | One of the operations defined for a class (it can be thought of as a 'function') |
| Persistent dialogue box | One which remains on the screen even when the menu tree is closed down. It must be explicitly removed by cancelling it, or by pressing Escape. |
| Resource file | Described in *Resource File Formats* on page 499. It is a file containing a sequence of templates from which to build objects. |
| String | A NUL-terminated sequence of ASCII characters. |
| Textual name (name) | Can be formed of any sequence of alphanumeric characters and underscores ('_'). It must begin with an alphabetic character. Special names used by the Toolbox can begin with the underscore character ('_').<br>A name cannot be longer than 12 characters, including the NUL terminator character. |
| Transient dialogue box | One which appears on the screen, and is removed when the current menu tree is closed down |
| User | The human user of a client application |
| User Interface Object (object) | A fundamental building block for windowed applications (e.g. a menu). All objects share a set of common methods which can be applied to them. An object consists of a fixed size header followed immediately in memory by a variable size body. |
| Word | A 4-byte entity, aligned at a 4-byte address. |

## General notes

- Where a buffer holds a string, this string will be NUL-terminated on exit from a SWI or when delivered in an event block. Strings which are given as input parameters to a SWI should be terminated by a control character (i.e. in the range 0-31 inclusive).
- Where the size of a buffer is specified, this includes any terminating character. If the size of buffer supplied for a string is not large enough an error is **not** returned; instead the buffer is filled (including a terminating NUL), and the returned number of bytes 'written to the buffer' will be the size of buffer which would be required. Thus you may wish to check that the number of bytes written to the buffer is less than or equal to the supplied buffer size.
- Note that **all** SWIs have a flags word in R0. All undefined bits in this flags word should be 0.
- Unless otherwise stated, changes to objects which are visible on the screen are immediate.

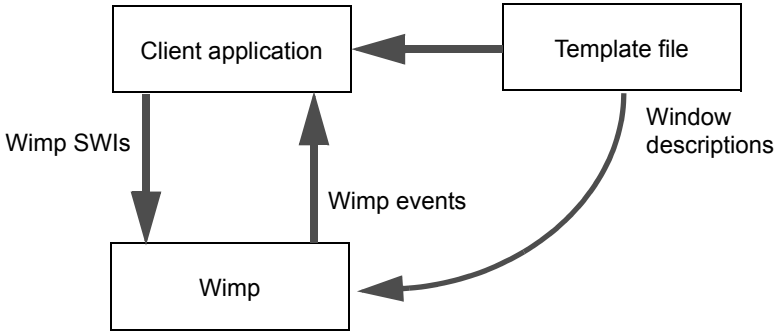
# Toolbox Application Model

The Toolbox is intended to provide a layer of abstraction between an application and the Wimp. In a manner analogous to the use of High Level Programming Languages, the Toolbox allows the programmer to think more in terms of the problem to be solved rather than the detailed mechanics of how to achieve a solution.

## Traditional desktop application

In a traditional desktop application, the programmer writes code which interfaces directly to the Window Manager (Wimp) through Wimp SWIs. Such an application uses a 'Templates' file to define templates from which it can create windows at run-time, but must create other user-interface objects from within its code (e.g. menus). The events which are delivered to a Wimp application refer to low-level Wimp operations like mouse clicks:

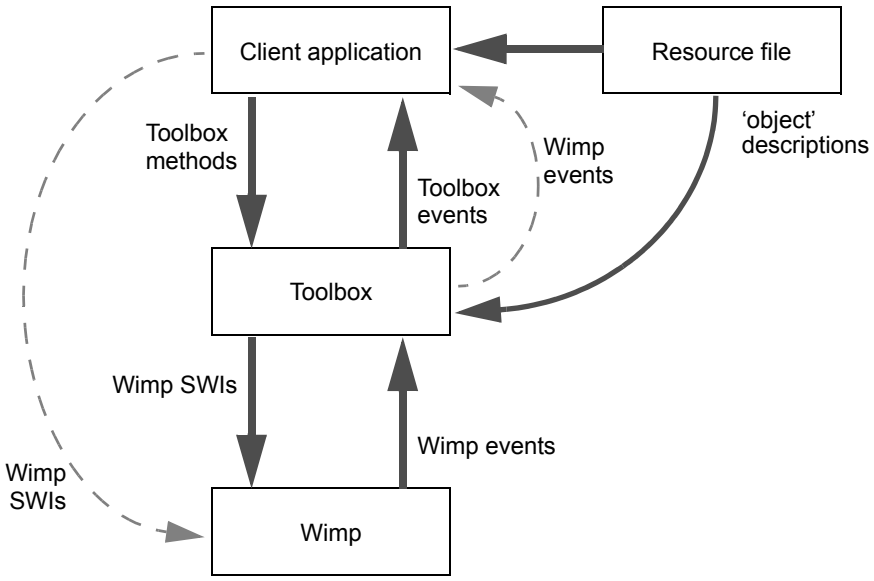
*Figure 1.1 Wimp application model*



## Toolbox application

In a Toolbox desktop application, the programmer writes code which interfaces mainly to the Toolbox through Toolbox 'methods', only occasionally resorting to making low-level Wimp SWI calls. A Toolbox application uses a 'Resources' file to define templates from which it can create a large number of user-interface objects including windows, menus and iconbar icons. Events which are delivered to a Toolbox application are at a higher level of abstraction than Wimp events.

*Figure 1.2 Toolbox application model*




### Wimp events

The application will generally see all Wimp events, with the following exceptions:

| | |
|---|---|
| ColourDbox | will not see redraw events.<br>Where it has input focus you will not see keypress events. |
| Window object | will not see Open Window Request or Close Window Request events if the window is marked as being auto-open or auto-close respectively. |

# Toolbox objects

An object is essentially one part of the user interface of a desktop application; for example, a window or a menu or an icon on the icon bar.

At run-time, each object is identified by an *object id* which is allocated when the object is created. An object id is a 32-bit integer, which should not be interpreted by the client application. An object id of 0 is used to indicate 'no object'.

## Object classes

The type of an object is called its 'class', which identifies its attributes and the set of operations which can be performed on it at run-time.

It is possible to determine the class of an object at run-time, using SWI Toolbox_GetObjectClass.

The set of classes which are supported in this release of the Toolbox are:

| Class name | Meaning | page |
|---|---|---|
| Colour Menu | a menu for selecting a desktop colour | 79 |
| Colour Dbox | a dialogue box for selecting any colour | 65 |
| DCS | a dialogue box for discard/cancel/save for unsaved data | 89 |
| File Info | a dialogue box showing information on a given file | 103 |
| Font Dbox | a dialogue box for selecting font characteristics | 119 |
| Font Menu | a menu for selecting a font | 137 |
| Iconbar Icon | an icon on the left or right of the iconbar | 145 |
| Menu | a Wimp menu | 165 |
| Print Dbox | a dialogue box for selecting print options | 201 |
| Prog Info | a dialogue box for showing program information | 221 |
| Quit | a dialogue box for handling quit with unsaved data | 233 |
| SaveAs | a dialogue box for saving data by icon drag | 245 |
| Scale View | a dialogue box for selecting a scale factor | 271 |
| Window | a Wimp window | 287 |

The Toolbox is designed to be extensible, so this set of classes will be increased in future releases, and can also be increased by third party developers.

## Object components

An object 'component' defines one of a set of distinct parts which make up an object; for example a menu entry is a component of a Menu object, and a gadget (see later) is a component of a Window object. A component is allocated a component id by which to identify it uniquely within its containing object; this component id is chosen by the

client application when the component is created. For menus it can have a value in the range 0 to 0xfffffffd, and for windows a value in the range 0 to 0x7fffff. All higher component ids are reserved for internal Toolbox use. A component id of 0xffffffff is used to indicate 'no component'.

## Object Methods

At run-time, the client application manipulates its objects by using 'methods', which are in fact implemented via Toolbox SWIs. The Toolbox will dispatch these methods to the appropriate module which implements the class of object to which the method is being applied.

### Creating an object

An object is created using SWI Toolbox_CreateObject (see page 19). The client application supplies either the name of a template for the object, or the address of a block of memory containing such a template. If a name is provided, then the Toolbox will look for the template in the application's Resource file (see later). The client application will be passed back an object id for the newly-created object if successful.

When an object which has 'attached' objects is created, then the attached objects are also created. See *Attached objects* on page 11 for a fuller description of this process.

Given its object id, it is possible to find out the name of the template used to create an object using SWI Toolbox_GetTemplateName.

### Deleting an object

An object is deleted using SWI Toolbox_DeleteObject (see page 20). If the object is visible on the screen and it is deleted, then the Toolbox first hides the object.

When an object which has attached objects is deleted, then unless the 'non-recursive' bit is set in this SWI's flags word, all its attached objects are also deleted. See *Attached objects* on page 11 for a fuller description of this process.

### Showing an object

An object is shown on the screen using SWI Toolbox_ShowObject (see page 21).

By setting bits in the SWI's flags word, the client may choose to show the object with either SWI Wimp_CreateMenu semantics or SWI Wimp_CreateSubMenu semantics. This is generally referred to as showing the object 'transiently', and can be used, for example, to show transient dialogue boxes. By default, an object is shown 'persistently', in other words it must be explicitly dismissed from the screen. Not all objects support both sets of semantics.

When an object is shown, the client application chooses where the object will appear on the screen by specifying one of three 'show types'.

- A ‘default’ show type means that the object will be shown at a place determined by the module which implements the object's class. For example, a Menu object will be shown by default at a place 64 OS units to the left of the mouse pointer's position, to comply with the RISC OS 3 Style Guide.
- A ‘top left’ show type means that the client application supplies the coordinates of the top lefthand corner of where the object should be shown.
- A ‘full specification’ show type means that the client application supplies a buffer which contains all the information needed to position the object on the screen; the contents of this buffer is separately defined for each object class.

**Hiding an object**

An object is hidden using SWI Toolbox_HideObject (page 22). If the object was not visible on the screen, then this method has no effect.

**Object-specific methods**

Each object class provides a number of methods which are specific to that class (for example, a Window object's title can be set using the Window_SetTitle method). These methods are all accessed using SWI Toolbox_ObjectMiscOp (see page 24), with an appropriate reason code.

## Shared objects

It is often useful in an application for many objects to refer to one single instance of another object. A typical example is a multi-document editor, where a potentially large number of Windows all refer to a single shared Menu structure.

A shared object is specified as such in its template description. Whenever an attempt is made to create an object from such a template, the Toolbox first checks to see if there is already a copy of the object in existence, and in which case the id of this object is returned.

Reference counts are maintained for Shared objects. When the client tries to create such an object the reference count is incremented, and it is decremented when the client attempts to delete the object. The Shared object is only really deleted when its reference count reaches zero.

Shared objects can also be used effectively in conjunction with attached objects which are described on page 11.

Note: Sharedness is inherited by attached objects.

## Client handles

Each object can have associated with it a one-word value called its client handle. The value of this handle is specified entirely by the client application and is not interpreted by the Toolbox. This mechanism is intended to allow a state to be associated with an object by the client application (e.g. in a multi-document editor a Window object's client handle might be a pointer to the data which must be displayed in the Window).

An object's Client Handle is set and read using SWIs Toolbox_SetClientHandle (see page 25) and Toolbox_GetClientHandle (see page 25) respectively.

## Parent and ancestor objects

When an object is shown (using SWI Toolbox_ShowObject), there are two other objects which may be useful for the client application; these are the parent and ancestor objects.

### Parent objects

The parent of an object is defined as the object (and optionally a component of that object) which caused the object to be shown. This is represented by the parent object id and parent component id. For example if a Window object has been displayed as the result of a Menu selection, then that Window object has a parent with an object id given by the Menu's id, and a parent component id given by the component id of the entry which was selected.

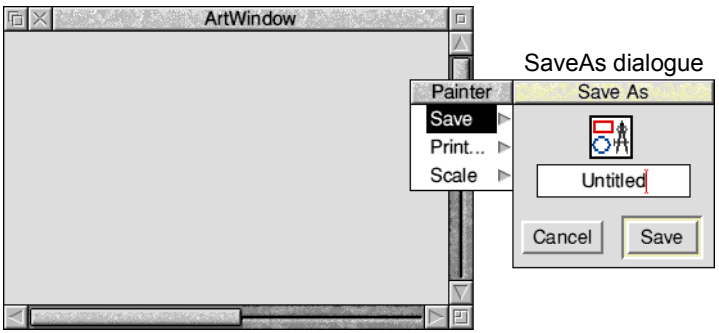
When SWI Toolbox_ShowObject is called explicitly by the client, the parent object and component ids must be specified. When this SWI is called on the client's behalf (for example, when a Menu is shown automatically for a Window), then the Toolbox fills this value in for the client.

### Ancestor objects

It is always possible to trace the ‘parentage’ of an object by recursively requesting the Parent of that object, thus moving ‘up’ the invocation hierarchy of objects which have been displayed. Since this is a common operation, an object can be designated as a potential so-called ‘Ancestor’. When an object is shown, it normally inherits the ancestor of its parent object; however, if the parent is marked as a potential ancestor, then the ancestor of the shown object is set to the id of the parent object.

Take the case where a multi-document editor has a document Window which has a Menu, which has a SaveAs dialogue box as a submenu. When an event occurs for the dialogue box, the client is probably most interested in getting the id of the document

Window (to get at its data and save it). By designating the document Window as an ancestor, the client can ensure that its id is available when events occur on the SaveAs dialogue box.




window designated as ancestor

The processes in the above example are as follows:

1. When the user presses Menu over the window, a Toolbox_ShowObject is raised on the Menu with the window as parent. As the window has been designated as ancestor, the Menu's ancestor will be the window.
2. When the user moves the pointer over the Save submenu arrow, the Menu module will show the SaveAs dialogue with itself (i.e. the Menu) as the parent object, and the Save component as the parent component. The SaveAs dialogue will inherit the Menu's ancestor (in this case the window).
3. Any event now raised on the SaveAs dialogue box will have the id block filled in with the Menu as the parent and the window as the ancestor.

The parent and ancestor of an object can be obtained by calling the SWIs Toolbox_GetParent and Toolbox_GetAncestor. Normally this will not be necessary, since (as shown in *The id block* on page 12) these values are made available on every return from Wimp_Poll.

## Auto-create and Auto-show objects

In order to save on coding required, it is possible to get the Toolbox to create an object from its template as soon as the resource file containing the template is loaded by the application. This is achieved by setting the Auto-create bit in the object template's flags word (see the chapter *ResEd* on page 417 to see how to do this). When such an object is created, the Toolbox raises a Toolbox_ObjectAutoCreated event, to allow the application to ascertain and store the object id of the newly-created object; the name of the template used to create the object is reported in this event.

It is also possible to specify that as soon as an object is created, it should be ‘shown’ on the screen. This is achieved by setting the Auto-show bit in the object template's flags word (see the chapter *ResEd* on page 417 to see how to do this). When such an object is created, it is shown using SWI Toolbox_ShowObject in its default place, and with no parent given.

It is also possible for an object to be auto-show but not auto-create.

### Attached objects

Certain objects allow other objects to be attached to them. When an object is created, all of its attached objects are also created, and a Toolbox_ObjectAutoCreated event is raised for each such attached object.

An example of an attached object is the object which will be shown when a user clicks the Select mouse button on an Iconbar Icon object. This attached object is created when the Iconbar Icon object is created.

Such side-effects of creating a given object are described in the *Application Program Interface* section in the chapter on each object class.

When an object with attached objects is deleted using SWI Toolbox_ObjectDelete, unless the non-recursive delete bit has been set, all attached objects are also deleted.

Attached objects can also usefully be combined with Shared objects. For example, if an application wishes the same Window to be displayed when the use clicks Select and Adjust on an Iconbar object, this can be achieved by specifying the same Window template name as the attached object to show for each of these mouse clicks, and marking the Window object as shared, so that the same object id is used for both cases.

It is important to note this side-effect of creating an object. For example, a Window object which has a complex menu tree attached to it, with many submenus and dialogue boxes, will have considerable side-effects when it is created.

Thus, in many cases, it is only necessary to create explicitly the ‘topmost’ object, and to allow the Toolbox to create the entire tree of attached objects.

## Event handling

An important part of managing the user interface using the Toolbox is the concept of a *Toolbox event*.

A Toolbox event is a Wimp event (not a message) which is delivered to the client application with an event code of Wimp_ToolboxEvent (0x200). Each Toolbox event has its own event code, which is a 32-bit integer defined in a similar manner to Wimp message numbers.

Toolbox events are essentially an abstraction on Wimp events, and are generated by the Toolbox modules in response to user interaction with Toolbox objects, and also in response to client application operations. Toolbox events are also used to warn the client application that a particular action has been taken by the Toolbox.

For example, if a client application creates and shows a Print Dialogue Box, when the user clicks on the **Print** button, a Toolbox event will be delivered to the application indicating that a Print operation has been requested, and giving the number of pages to be printed, the scale factor to use during printing etc.

Note that underlying events will also be received by the client.

## Toolbox event Codes

Event codes are allocated by Acorn. Events which are delivered by a Toolbox module will have codes which start at the SWI chunk base of the module.

The allocations are as follows; event codes are in the range 0 - 0x9ffff:

| Event codes | Use |
|---|---|
| 0x00001 - 0x0ffff | Available for use by the client |
| 0x10000 - 0x3ffff | Reserved for inter-application protocols |
| 0x40000 - 0x9ffff | Reserved for Toolbox module events |

## Format of a Toolbox event

When a Toolbox event is delivered to an application, the Wimp Poll block has the following format:

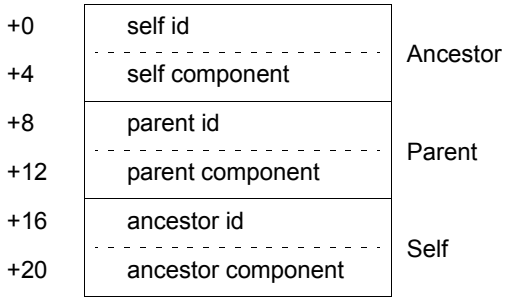
| Offset | Contents |
|---|---|
| + 0 | size of Toolbox event block<br>(16 - 236 in a multiple of four bytes; i.e. words) |
| + 4 | unique reference number |
| + 8 | Toolbox event code |
| +12 | flags |
| + 16... | Event-specific data |

Unless otherwise stated flags will be zero.

## The id block

Whenever the client application calls SWI Wimp_Poll, the Toolbox fills in a 6-word block of memory known as the *id block*, to indicate which object an event has occurred on. However, as Wimp messages do not typically occur on an object the id block will not be updated for a Wimp message.

This block is laid out as follows:




When a Toolbox event occurs, the object id of the object on which this event occurred is placed in the ‘self id’ field of the id block, and the ‘self component’ field is also filled in if the event has occurred for a particular component of that object. For example, a mouse click on an action button gadget within a Window object will result in an ActionButton_Selected Toolbox event being raised, with the Window object's id in the self id field of the id block, and the component id of the action button in the self component field.

The ‘parent id’ and ‘parent component’ fields are filled in by the Toolbox using the values which were last passed to SWI Toolbox_ShowObject. The ‘ancestor id’ and ‘ancestor component’ fields are filled in accordingly (being the ancestor of the parent).

The Toolbox uses a value of 0 as an object id to indicate ‘no object’, and a value of -1 as a component id to indicate ‘no component’.

When a Wimp event happens on an object, then the setting of the contents of the id block is object-specific, and is described in the object *events* section in the chapter on each object class.

The address of the 6-word block of client memory used as the application's id Block is passed to the Toolbox when the application registers itself using SWI Toolbox_Initialise (see page 32).

Note that Toolbox events are delivered to the object to which they are most appropriate, so for example a SaveAs object will receive SaveAs_DialogueCompleted events, whereas mouse clicks on a SaveAs object's underlying Window will be seen as being delivered to the Window object.

This behaviour can best be seen by taking some example Resource Files and dragging them to !ResTest, and monitoring the contents of the id Block as shown in !ResTest's log window, as events occur on the objects created from the Resource File.

### Raising a Toolbox event

A Toolbox event is raised using SWI Toolbox_RaiseToolboxEvent. Normally a client application will not need to use this SWI directly; the client simply quotes the Toolbox event code (or number), and associates it with a particular user action in its description of an object in the resource file. For example, one of the attributes of a Menu object, is the Toolbox event which is raised when a particular Menu entry is selected by the user. The Toolbox will raise this Toolbox event on the application's behalf, whenever a Menu Selection event is returned for that menu entry.

## Resource files

A resource file contains templates for the objects which a client application will create at run-time.

### Loading resource files

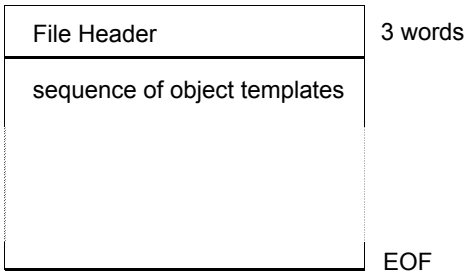
An application can load a resource file at run-time using SWI Toolbox_LoadResources. This is done on the application's behalf for a file called ‘res’ when the application calls SWI Toolbox_Initialise as described in *Task initialisation and run-time information* on page 15. SWI Toolbox_LoadResources could then be called after task start-up to load any further Resource Files which it needs to use.

### Resource file format

Resource files replace Wimp template files as the means to define templates for the user interface objects which an application will create at run-time. Whereas Wimp template files only allowed window descriptions to be given, a resource file will contain templates for any kind of Toolbox object.

A resource file consists of a fixed size header, followed by a contiguous sequence of object templates, where each template has a fixed size header, followed by an object body.

A resource file format is similar to a Drawfile, and can be represented diagrammatically as follows:




Each template has a textual name which can have no more than 12 characters (including the terminating NUL). This name is used by the application when using a template in a call to SWI Toolbox_CreateObject.

If a resource file is loaded which has named templates whose names clash with earlier loaded templates, the latest loaded template will be used, and the earlier template will no longer be accessible.

For a full description of the resource file format see the appendix *Resource File Formats* on page 499.

## Task initialisation and run-time information

Before it can use the Toolbox, a client application must first call SWI Toolbox_Initialise to register itself as a Toolbox task. This has several side-effects:

- If there is a file called `res` in the application's resource directory then it is loaded using SWI Toolbox_LoadResources; if such a file is not found, then the Toolbox tries a file called `res`<*n*>, where *n* is the currently configured country number, to allow for national variants.
- The application directory is searched for a Sprites file called `Sprites`, `Sprites22`, `Sprites23` or `Sprites24` depending on the current screen mode. This file is then loaded into a block of memory and will be used as the application's sprite area.
- The application directory is searched for a file called `Messages`, which is then loaded and registered with MessageTrans. If no such file is found, then a file called `Message`<*n*> is searched for, where *n* is the currently configured country number. The minimum requirement is that the Messages file should contain a message whose tag is `_TaskName`, giving the name of the application.
- SWI Wimp_Initialise is then called on behalf of the application.

When a Toolbox task has been registered with the Toolbox, the client application can obtain the following information by calling SWI Toolbox_GetSysInfo:

- the task's name (as given by the _TaskName message in the Messages file).
- the 4-word message file descriptor returned when the task was initialised.
- the application's directory name.
- the application's Wimp task handle.
- a pointer to the sprite area used to load the application's Sprites file.

**Important**: Since the Toolbox uses Wimp messages, a client aplication should **not** call SWI Wimp_AddMessages or SWI Wimp_RemoveMessages.

## Message texts and nationalisation

When using the Toolbox, the writer of a client application should be aware of where textual messages are held, which will need translating if the client is to be ‘nationalised’ for a particular RISC OS territory.

All of the modules contained in the Toolbox have a default set of messages and object templates which they will use when displaying windows, reporting errors, displaying menus etc. These are registered with ResourceFS, and are looked up using MessageTrans. So in order to produce a nationalised Toolbox, these messages and templates will need replacing.

In a resource file, textual messages are held in Messages Tables, and objects created at run-time will contain pointers to these messages. These messages are the ones which have been specified by the client of the Toolbox to be used when creating objects, and will often consist of alternative text to use instead of the defaults provided by the Toolbox modules themselves. These messages are **not** tagged messages looked up using MessageTrans, but are actual strings.

The client application will also have a file called Messages in its application directory. This file is automatically loaded by the Toolbox when the client calls SWI Toolbox_Initialise. The Messages file will contain at least the name of the application (in a message whose tag is _TaskName), and any other messages which the application wishes to look up using MessageTrans at run-time. This will typically contain error messages, and ones which are not associated with objects. After calling SWI Toolbox_Initialise, the client will have a MessageTrans file descriptor to use when looking up these Messages.

This means that in order to nationalise an application, the writer will need to provide new Messages and new resource file messages (using **Export messages** in ResEd).

## An Example object

Let us look at an example of a Toolbox object, to illustrate some of the features detailed in earlier sections.

An Iconbar Icon object is used to place an application icon sprite (and optionally some text) on the RISC OS icon bar. The template for such an object has the following fields, which can be set using !ResEd (the Resource Editor):

| Field | Meaning |
|---|---|
| position | a negative integer giving the position of the Icon on the Iconbar (as specified in SWI Wimp_CreateIcon) |
| priority | the priority of this Icon on the Iconbar (as specified in SWI Wimp_CreateIcon) |
| sprite name | the name of the sprite to use for this Iconbar Icon |
| max sprite name | the maximum length of sprite name to be used |
| text | an optional string which will be used for a Text&Sprite Iconbar Icon (ie the text that will appear underneath the Icon on the Iconbar) |
| max text length | if the Iconbar Icon has text, then this field gives the maximum length of a text string which will be used for it |
| menu | the name of the template to use to create a Menu object for this Iconbar Icon |
| select event | the Toolbox event code to be raised when the user clicks Select on the Iconbar Icon (if 0 then Iconbar_Clicked is raised) |
| adjust event | the Toolbox event code to be raised when the user clicks Adjust on the Iconbar Icon (if 0 then Iconbar_Clicked is raised) |
| select show | the name of a template to use to show an object when the user clicks Select on the Iconbar Icon |
| adjust show | the name of a template to use to show an object when the user clicks Adjust on the Iconbar Icon |
| help message | the message to respond to a help request with, instead of the default |
| max help | the maximum length of help message to be used |

The client application will create an Iconbar Icon object by calling SWI Toolbox_CreateObject, supplying a template which gives values for all of the above fields.

As a side-effect of this creation, the Iconbar Icon's attached objects are also created (if their templates have been provided) i.e. menu, select show and adjust show. The object ids of these attached objects are then held within the Toolbox internal data structure which represents the Iconbar Icon.

When the application calls SWI Toolbox_ShowObject on an Iconbar Icon, it will be shown in a Style Guide compliant place on the Iconbar. When SWI Toolbox_HideObject is called, the Icon will be removed from the Iconbar.

When a HelpRequest message is received, the supplied help message will automatically be returned to the sender of the message.

When the user clicks the Select or Adjust mouse buttons on the Iconbar Icon, then if the names of suitable object Templates have been supplied, these objects will be shown automatically by the Toolbox.

When the user clicks the Menu button on the Iconbar Icon, then if the name of a suitable Menu object Template has been supplied, it will be shown in a RISC OS 3 Style Guide compliant place (i.e. 96 OS units above the bottom of the screen).

There are a number of methods which have been defined for an Iconbar Icon to allow the client application to manipulate it at run-time; for example if it wishes to change the sprite used on the Iconbar for this Icon, then the Iconbar_SetSprite method will be used; if it wishes to provide a new Menu object which will be displayed when the Menu button is clicked on the Iconbar Icon, then the Iconbar_SetMenu method will be used.

## Toolbox SWIs

## SWI Toolbox_CreateObject (0x44ec0)

### On entry

R0 = flags (bit 0 set means create from memory)
R1 = pointer to name of template
(R1 = pointer to description block if bit 0 of flags word set)

### On exit

R0 = id of created object
R1-R9 preserved

### Use

This SWI creates an object either from a named template description which has been loaded from the resources file or from a template description block in memory. The exact format of the description block depends on the class of the object.

If the client application wishes to use the description block form of this SWI, then the block should begin with a standard object header, and the body of the object should be as specified in the *Templates* section of the chapter for that object. Any StringReferences, MsgReferences, and SpriteAreaReferences should hold 'real' pointers, and should not require relocation; also the 'body offset' field should contain a real pointer to the object body.

### C veneer

```
extern _kernel_oserror *toolbox_create_object ( unsigned int flags,
                                                void *name_or_template,
                                                ObjectId *id
                                              );
```

# SWI Toolbox_DeleteObject (0x44ec1)

## On entry

R0 = flags (bit 0 set means do not delete recursively)
R1 = object id

## On exit

R1 - R9 preserved

## Use

This SWI deletes a given object.

By default, any objects ‘attached’ to this object are also deleted. If bit 0 of the flags word is set, then this does not happen.

If it is a Shared object, this will result in its reference count being decremented, and it will only be really deleted when this reaches 0.

The Toolbox raises a Toolbox_ObjectDeleted event when the object's reference count reaches zero.

## C veneer

```
extern _kernel_oserror *toolbox_delete_object ( unsigned int flags,
                                                ObjectId id
                                              );
```

## SWI Toolbox_ShowObject (0x44ec3)

### On entry

R0 = flags
bit 0 set means show using the semantics of Wimp_CreateMenu
bit 1 set means show using the semantics of Wimp_CreateSubMenu
R1 = object id
R2 = show 'type':

| Type value | Meaning |
|---|---|
| 0 | show in the 'default' place. This has a different meaning depending on the type of object shown |
| 1 | R3 points to a buffer giving full details of how to show the object |
| 2 | R3 points to a 2-word buffer giving the screen coordinates of the top left corner of the object to be displayed |

R3 = 0
or pointer to buffer giving object-specific data for showing this object
or pointer to 2-word buffer giving coordinates of top left corner of object
R4 = Parent object id
R5 = Parent component id

### On exit

R1-R9 preserved

### Use

This SWI shows the given object on the screen.

R2 gives the type of 'show' operation which is being performed. Not all types of show operation will be appropriate to all objects.

The buffer pointed at by R3 may hold data specific to this class of object, including information as to where the object should appear on the screen. The exact format of the buffer is specified separately for each object class. For example for a Window object, the buffer will hold a block of data which can be passed to SWI Wimp_OpenWindow.

Note: some objects support a bit in their flags word specifying that a warning should be raised before the object is shown. In this case, the SWI Toolbox_ShowObject will return, but the object will not yet be visible on the screen. The object will be visible (at the earliest) after the next call to Wimp_Poll after the warning is delivered.

### C veneer

```
extern _kernel_oserror *toolbox_show_object ( unsigned int flags,
                                              ObjectId id,
                                              int show_type,
                                              void *type,
                                              ObjectId parent,
                                              ComponentId parent_component
                                            );
```

## SWI Toolbox_HideObject (0x44ec4)

### On entry

R0 = flags
R1 = object id

### On exit

R1-R9 preserved

### Use

This SWI removes the given object from the screen, if it is currently being shown.

### C veneer

```
extern _kernel_oserror *toolbox_hide_object ( unsigned int flags,
                                              ObjectId id
                                            );
```

## SWI Toolbox_GetObjectState (0x44ec5)

### On entry

R0 = flags
R1 = object id

### On exit

R0 = object state

### Use

This SWI returns information regarding the current state of an object. The state is indicated by bits in the value returned in R0. Bits 0-7 refer to all objects and bits 8-31 are used to indicate object-specific state.

The generic state bits are:

| Bit | Meaning when set |
|---|---|
| 0 | object is currently showing |

### C veneer

```
extern _kernel_oserror *toolbox_get_object_state ( unsigned int flags,
                                                   ObjectId id,
                                                   unsigned int *state
                                                 );
```

# SWI Toolbox_ObjectMiscOp (0x44ec6)

## On entry

R0 = flags
R1 = object id
R2 = method code
R3-R9 contain method-specific data.

## On exit

R1-R9 preserved

## Use

The exact operation of this SWI depends on the class of the object being manipulated, and on the reason code supplied.

Each object class implements a number of methods which are specific to that object (e.g. a Window class may implement a method for adding/removing keyboard short-cuts for a Window object).

## SWI Toolbox_SetClientHandle (0x44ec7)

### On entry

R0 = flags
R1 = object id
R2 = client handle

### On exit

R1-R9 preserved

### Use

This SWI sets the value of the client handle for this object.

### C veneer

```
extern _kernel_oserror *toolbox_set_client_handle ( unsigned int flags,
                                                    ObjectId id,
                                                    void *client_handle
                                                  );
```

## SWI Toolbox_GetClientHandle (0x44ec8)

### On entry

R0 = flags
R1 = object id

### On exit

R0 = client handle for this object

### Use

This SWI returns the value of the client handle for this object.

### C veneer

```
extern _kernel_oserror *toolbox_get_client_handle ( unsigned int flags,
                                                    ObjectId id,
                                                    void *client_handle
                                                  );
```

# SWI Toolbox_GetObjectClass (0x44ec9)

## On entry

R0 = flags
R1 = object id

## On exit

R0 = object class

## Use

This SWI returns the class of the specified object. This is a 32-bit integer, which identifies a given class; allocation of class identifiers is handled by Acorn.

## C veneer

```
extern _kernel_oserror *toolbox_get_object_class ( unsigned int flags,
                                                   ObjectId id,
                                                   ObjectClass *object_class
                                                 );
```

## SWI Toolbox_GetParent (0x44eca)

### On entry

R0 = flags
R1 = object id

### On exit

R0 = Parent id
R1 = Parent component id

### Use

This returns the value of the object id which was passed as the parent in a SWI Toolbox_ShowObject call (even if the parent has subsequently been deleted). The component id is for cases where the parent has a subcomponent like a Menu with a Menu entry. An object which has not yet been shown will have a parent object id of 0 and a component id of -1.

### C veneer

```
extern _kernel_oserror *toolbox_get_parent ( unsigned int flags,
                                             ObjectId id,
                                             ObjectId *parent,
                                             ComponentId *parent_component
                                           );
```

# SWI Toolbox_GetAncestor (0x44ecb)

## On entry

R0 = flags
R1 = object id

## On exit

R0 = Ancestor id
R1 = Ancestor component id

## Use

This returns the id of the Ancestor of the given object (and its component id, in the case of an ancestor which has subcomponents like a Menu with a Menu entry). Note that the Ancestor may have been deleted, since this object was shown. An object which has not yet been shown will have an ancestor object id of 0 and a component id of -1.

## C veneer

```
extern _kernel_oserror *toolbox_get_ancestor ( unsigned int flags,
                                               ObjectId id,
                                               ObjectId *ancestor,
                                               ComponentId *ancestor_component
                                             );
```

## SWI Toolbox_GetTemplateName (0x44ecc)

### On entry

R0 = flags
R1 = object id
R2 = pointer to buffer to hold template name
R3 = length of buffer

### On exit

R3 = length of buffer required (if R2 was zero)
else buffer pointed at by R2 holds template name
R3 holds number of bytes written to buffer

### Use

This SWI returns the name of the template used to create the object whose id is passed in R1.

### C veneer

```
extern _kernel_oserror *toolbox_get_template_name ( unsigned int flags,
                                                    ObjectId id,
                                                    char *buffer,
                                                    int  buff_size,
                                                    int *nbytes
                                                  );
```

# SWI Toolbox_RaiseToolboxEvent (0x44ecd)

## On entry

R0 = flags
R1 = object id
R2 = component id
R3 = pointer to Toolbox event block

## On exit

R1-R9 preserved

## Use

This SWI raises the given Toolbox event. The block pointed at by R3 should have the format described in *Format of a Toolbox event* on page 12. The Toolbox will put the unique reference number into the block before exit from this SWI. The object id and (optional) component id will be those filled in on return from Wimp_Poll; they refer to the object on which the Toolbox event is being raised; the Toolbox does not check the validity of these values.

## C veneer

```
extern _kernel_oserror *toolbox_raise_toolbox_event ( unsigned int flags,
                                                      ObjectId id,
                                                      ComponentId component,
                                                      ToolboxEvent *event
                                                    );
```

## SWI Toolbox_GetSysInfo (0x44ece)

### On entry

R0 = flags

**R0 ValueMeaning**

0 return task name
1 return 4-word messages file descriptor
2 return name of directory passed to Toolbox_Initialise
3 return task's Wimp task handle
4 return pointer to sprite area used

R1, R2 depends on entry value of R0 (see below)

### On exit

R0

**On entryOn exit**

0 R2 holds size of buffer required (if R1 was 0)
else buffer pointed at by R1 holds task name

1 buffer pointed at by R1 contains a 4-word messages file descriptor

2 R2 holds size of buffer required (if R1 was 0)
else buffer pointed at by R1 holds directory name passed to Toolbox_Initialise

3 R0 contains task handle

4 R0 contains sprite area pointer

### Use

This SWI is used to get information for the client application. The nature of the information required is indicated by R0.

### C veneer

```
extern _kernel_oserror *toolbox_get_sys_info ( unsigned int reason_code,
                                               _kernel_swi_regs *regs
                                             );
```

## SWI Toolbox_Initialise (0x44ecf)

### On entry

R0 = flags
R1 = last Wimp version number known to task * 100 (must be ≥310)
R2 = pointer to list of Wimp message numbers which the client wishes to receive, terminated by a 0 word
If R2 points to just a 0 word, then all messages are delivered
If R2 = 0, then no messages are delivered (apart from the Quit message).
R3 = pointer to list of Toolbox event codes which the client wishes to receive, terminated by a 0 word
If R3 points to just a 0 word, then all Toolbox events are delivered
If R3 = 0, then no Toolbox events are delivered
R4 = pointer to Directory name in which to find resources
R5 = pointer to 4-word buffer to receive messages file descriptor
R6 = pointer to buffer to hold object ids on return from Wimp_Poll (the id block)

### On exit

R0 = current Wimp version number * 100
R1 = Wimp task handle for this client
R2 = Pointer to Sprite area used
Buffer pointed to by R5 is filled in with a MessageTrans file descriptor for the messages file to be used

### Use

This SWI is used by the client application before any other Toolbox SWIs.

First the Toolbox tries to load a file called `res` in the directory given by the string pointed to by R4; this is done by calling SWI Toolbox_LoadResources.

If a file called `res` is not found, then the Toolbox tries `res<`*n*`>` where *n* is the currently configured country number.

The application directory is searched for a Sprites file appropriate for the current mode (i.e. called `Sprites`, `Sprites22`, or `Sprites23`) and if such a file exists, a sprite area is allocated, and the file loaded into this area. A pointer to the area is returned in R2 (or 1 is returned if there was no such file found, and so the Wimp Sprite pool is used for Sprite references in the client application).

This SWI registers a file called ‘Messages’ found in the given directory with MessageTrans and passes back a 4-word MessageTrans file descriptor for use by the client. SWI Wimp_Initialise is called on the client’s behalf, using the Wimp version number passed in R1, and the messages list pointed at by R2.

If a file called `Messages` is not found, then the Toolbox tries `Message<`*n*`>` where *n* is the currently configured country number.

The task name passed to SWI Wimp_Initialise must be given in the client's messages file; it should be an entry with tag '_TaskName'.

The buffer pointed at by R6 will be used on each call to Wimp_Poll to inform the client which object an event occurred on, and that object's parent and ancestor objects. On return from Wimp_Poll this block will be filled in as follows:

R6 + 0 ancestor object id
R6 + 4 ancestor component id
R6 + 8 parent object id
R6 + 12parent component id
R6 + 16'self' object id
R6 + 20'self' component id

## C veneer

```
extern _kernel_oserror *toolbox_initialise ( unsigned int flags,
                                             int wimp_version,
                                             int *wimp_messages,
                                             int *toolbox_events,
                                             char *directory,
                                             MessagesFD *mfd,
                                             IdBlock *idb,
                                             int *current_wimp_version,
                                             int *task,
                                             int *sprite_area
                                           );
```

# SWI Toolbox_LoadResources (0x44ed0)

## On entry

R0 = flags
R1 = pointer to resource filename

## On exit

R1 - R9 preserved

## Use

This SWI loads the given resource file, and creates any objects which have the auto-create bit set. When such an object is created, the Toolbox raises a Toolbox_ObjectAutoCreated Toolbox event.

The filename of the resource file should be a full pathname.

After this SWI has been called, any templates from the resource file can be used to create objects, by quoting the template name.

## C veneer

```
extern _kernel_oserror *toolbox_load_resources ( unsigned int flags,
                                                 char *resources
                                               );
```

## SWI Toolbox_TemplateLookUp (0x44efb)

### On entry

R0 = flags
R1 = pointer to template name (Ctrl terminated)

### On exit

R0 = pointer to description block

### Use

This SWI returns a pointer to a block suitable to pass to Toolbox_CreateObject or Window_ExtractGadget.

### C veneer

```
extern _kernel_oserror *toolbox_template_lookup    ( unsigned int flags,
                                                     char *name,
                                                     void **id,
                                                    );
```

# Toolbox events

## Toolbox_Error (0x44ec0)

**Block**

+ 8 0x44ec0
+ 16 error number
+ 20... error text

**Use**

All Toolbox SWIs may return direct errors, with the V bit set. If any part of the Toolbox detects an error, whilst it is not processing a SWI, it will raise a Toolbox_Error event which the client can report when he next calls Wimp_Poll.

For example, if a client uses Toolbox_ShowObject on an object which has the bit set to warn the client before the object is shown, the Toolbox will wait until the next call to Wimp_Poll before actually showing the object; if there is an error when it tries to do the show, then this will be reported through a Toolbox_Error event, since the SWI Toolbox_ShowObject will have already returned with no error indicated.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                errnum;
  char               errmess [256-20-sizeof(ToolboxEventHeader)
                              -sizeof(ObjectId)
                              -sizeof(ComponentId)
                              -sizeof(int)];
} ToolboxErrorEvent;
```

## Toolbox_ObjectAutoCreated (0x44ec1)

**Block**

+ 8 0x44ec1
+ 16... Name of template from which object was created

**Use**

This Toolbox event is raised by the Toolbox after it creates objects from templates which have their *auto-create* bit set, when the application's resource file is loaded. This allows the client application to get the ids of such objects for later use.

This event is also raised when an attached object is created as a side-effect of creating the object to which it is atached.

The client can establish the object's id by looking at the 'self' field of the id block which it passed to Toolbox_Initialise (see later).

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  char               template_name
[256-20-sizeof(ToolboxEventHeader)-sizeof(ObjectId)-sizeof(ComponentId)];
} ToolboxObjectAutoCreatedEvent;
```

## Toolbox_ObjectDeleted (0x44ec2)

**Block**

+ 8 0x44ec2

**Use**

This Toolbox event is raised by the Toolbox after it deletes an object. It is useful when a 'recursive' delete is done, resulting in other objects being deleted.

The client can establish the object's id by looking at the 'self' field of the id block which it passed to Toolbox_Initialise.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
} ToolboxObjectDeletedEvent;
```

# 2 Building an application

This chapter describes how an application (!Hyper, which can be found in the Examples directory) was designed with Acorn C/C++. In particular it demonstrates how using !ResEd and !ResTest can lead to very short design times. The first section describes how to use !Hyper, and the second section is a description of how it was designed and implemented.

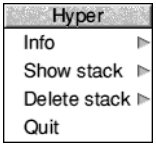
## Guide To Hyper

!Hyper is a multi-document viewer for HCL files (see *HyperCard Control Language* on page 63 for the syntax). HCL files define stacks of cards allowing multiple Draw objects to be linked such that a user may click on active areas (called *hot spots*) of a viewer to navigate between different cards. Only one card from a stack is visible at any time in a viewer, although being multi-document, !Hyper may display several views onto the same stack, each of which may be displaying a different card.

!Hyper is started by double-clicking on its application icon or by double clicking on an HCL file (but only after !Hyper has been seen by the Filer).

### Application icon menu

Clicking Menu over the application icon will display the following menu:




**Info** leads to a standard program information dialogue box.

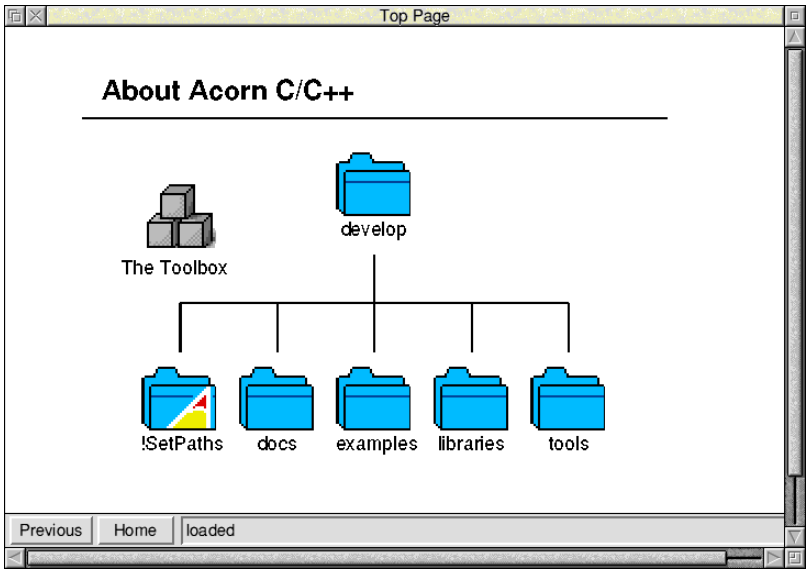
**Show stack** allows any closed viewers to be reopened or brings to the top an already opened one.

**Delete stack** will remove it from memory.

Note that if no stacks have been loaded then the show stack/delete stack will be greyed out.

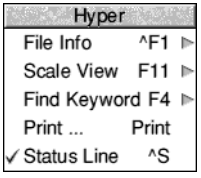
**Quit** will exit the application.

Once a stack has been loaded, !Hyper will open a viewer displaying the 'Home Card' of that stack. For example:




The user can move from one card to another by clicking on hotspots. Hot Spots will usually be identifiable in some way, though !Hyper will change the pointer shape whilst it is over one. It is also possible to jump to the Home Card or back to the previous card by clicking on the action buttons in the status area at the bottom of the window.

Pressing menu over a viewer window will display the following menu:




This allows various operations to be performed on the stack being displayed:

**File Info** displays information about the file.

**Scale View** leads to a standard scale dialogue box which lets the user zoom in and out on a card.

**Find Keyword** allows searching for keywords that are stored in the stack. This allows an index type search to be applied.

**Print...** allows the current card to be printed.

**Status Line** controls whether or not the status area is to be displayed at the bottom of the viewer window.

### Keyboard Short-cuts

Clicking in a viewer gives it the keyboard input focus. This then allows various keyboard short-cuts to work. The standard keys for **Find Keyword**, **Scale View**, **File Info** and **Print...** all work (as can be seen from the menu, pictured above) as well as **p** and **h** for previous and home.

## How !Hyper was designed

It is worth having !Hyper at hand whilst reading this section. Loading its resource file into !ResEd and !ResTest will make it easier to see the various linkages between objects and observe the events that are raised when interacting with the user interface. The chapters later in this manual give full information on each of the classes involved.

### Requirements

Before designing the structure of !Hyper we had to decide what it must be able to do. We wanted to design a HyperCard-type application with the following features:

- multi-document capability
- navigation between cards (based around Draw files) using hotspots
- home/previous facility
- keyboard driven option
- suitable for range of screen modes/scalable output
- easily extendible
- easy to make a demo version
- find capability
- ability to print a card
- maintain history of all loaded cards.

## Design decisions

From the required features, we made the following design decisions.

**Shared objects and client handles**

The multi-document support suggested the use of shared objects and the use of client handles for maintaining what file the viewer was showing. By doing this we would reduce memory usage (by just having one copy of the shared menus and dialogues) without complicating the association between events on a menu and the viewer that it was opened from.

**Event driven interface**

Given that we wanted to extend and modify the interface easily, we decided to make it event driven as opposed to object driven. In other words when registering event handlers, we register for specific event numbers, rather than a generic event (e.g. ActionButton_Selected) on a specific component of an object. In this way we are able to modify the interface (e.g. reorder a menu or even move menu entries off onto a submenu) without having to change the code.

**AboutToBeShown events**

We also decided to take advantage of a number of features offered by the toolbox such as the 'About To Be Shown' events. These made it possible to set up dialogue boxes as they were being shown, and not have to update them constantly as other parts of the application altered data. A less obvious benefit of this mechanism is that since the toolbox tells us the object id of what is being shown, we do not have to remember this ourselves, and in fact it is possible to let the toolbox automatically create such objects.

A good example of this is the Program Information box. This is created by the toolbox as a side effect of creating the iconbar (which is created on initialisation due to it having its AutoCreate bit set). We then just need to register for the ProgInfo_AboutToBeShownEvent and in our handler set the version string from our message file.

**Standard objects**

To be Style Guide compliant (and to make less work for ourselves) we can use the standard PrintDbox, Scale, ProgInfo and FileInfo object templates supplied by the Toolbox.

**Keyboard short-cuts**

As we want !Hyper to be keyboard drivable, we can make use of the Toolbox's keyboard short-cuts facility.

## How !Hyper was implemented

The rest of this chapter takes you through the stages involved in implementing !Hyper. It breaks down into the following sections:

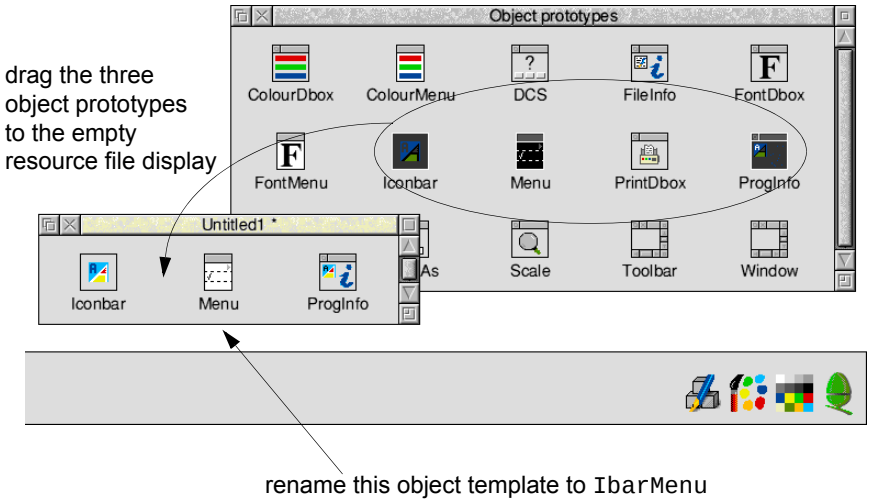
- *Creating and testing a simple resource file for !Hyper* (below).
- *File loading* on page 48 – coping with Filer_Open messages on HCL files.
- *Handling views* on page 50 – extending our simple resource file, redraw handlers, implementing hotspots, linking data structures, showing and hiding views, adding keyboard short-cuts etc.
- *Modifying the interface* on page 59 – changing the interface by editing the resource file.
- *Client Events* on page 63 – a list of client events used in !Hyper.
- *Summary* on page 63 – features of the toolbox demonstrated in this chapter.

### Creating and testing a simple resource file for !Hyper

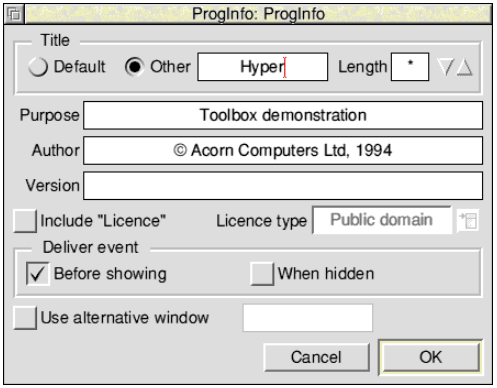
The first stage in implementing !Hyper was to create and test a very simple resource file consisting of an IconBar object template, a Menu object template for the iconbar icon, and a ProgInfo object template.

### Creating a basic resource file

1 We began by starting the resource file editor (ResEd – described in the chapter *ResEd* on page 417), and then opened a new resource file display. Next we opened an object prototypes window and dragged an IconBar object template, menu and ProgInfo object template to our empty resource file:

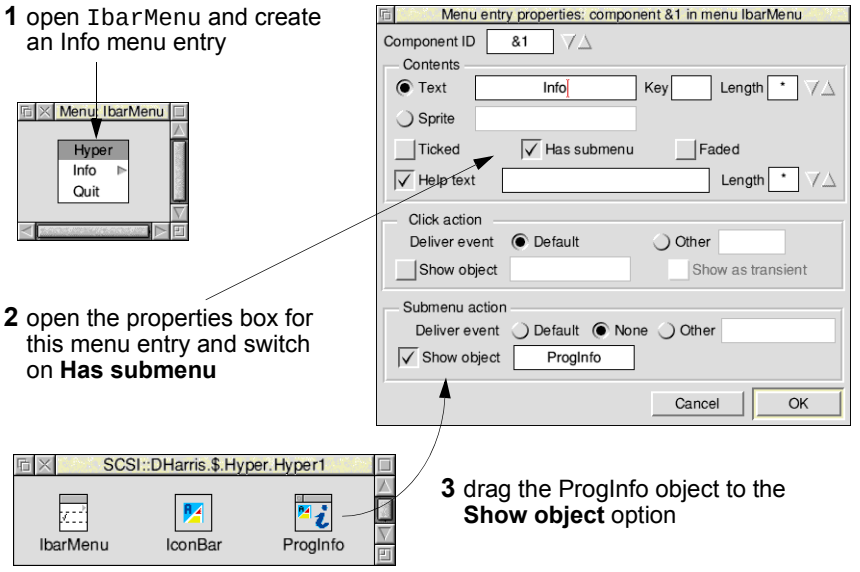



2 Next we double-clicked on the ProgInfo object template in the resource file display. This opened its properties box and we entered the information we wanted to appear in this box. We also switched on **Deliver event Before showing**:

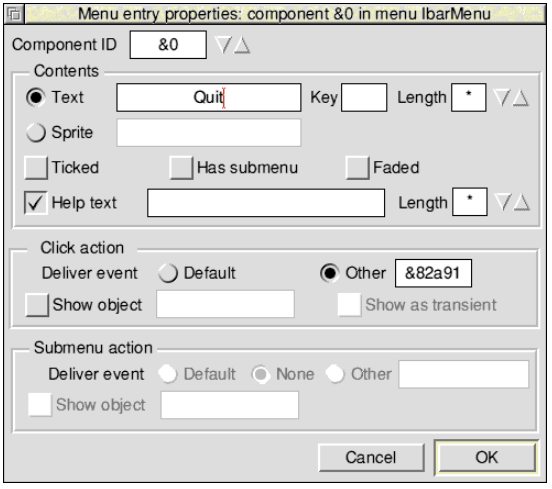



3 Then we edited the Menu object template in the resource file display and renamed it to `IbarMenu`. Next we double-clicked on `IbarMenu` and created two menu entries. The first entry we named `Info`, and the second entry `Quit`.

The Info entry we edited to include a submenu option to display the ProgInfo object template:

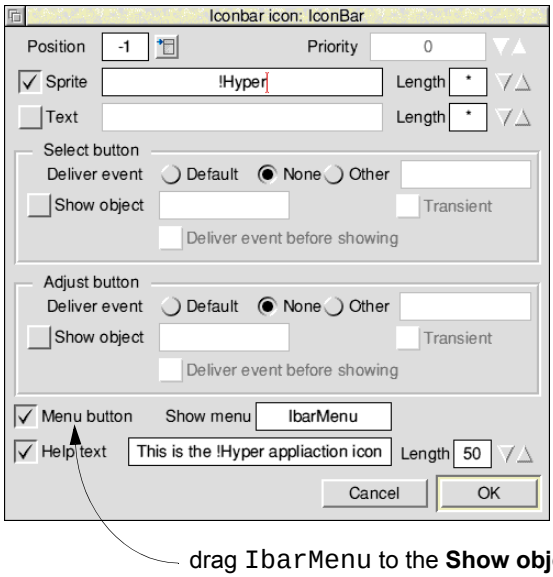



The Quit entry was edited to return a particular event:



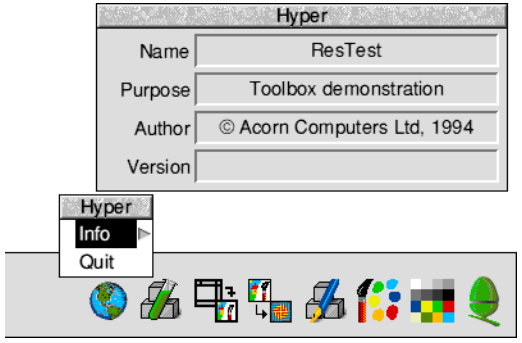

As we could choose our own events, the choice of 82a91 may seem strange. However, this is the same event that is generated by the Quit dialogue class, hence if we added editor features and required a quit confirmation, we could still use the same handles.

4 Finally we edited the Iconbar object template. We set up the sprite name, inserted some Help text, and dragged `IbarMenu` to the **Menu button** option:




### Using ResTest to check the resource file

To test out this initial design we dragged the resource file from !ResEd to !ResTest's iconbar icon (ResTest is described on page 491). As we had set the AutoCreate and AutoShow options for the iconbar object template, it appeared immediately on the iconbar. Pressing Menu over the icon opened our menu (`IbarMenu`) with the Quit and Info options. Sliding the mouse pointer over the submenu arrow opened the ProgInfo box:



Clicking on !ResTest's iconbar icon opened its Event Log window. We could now see what events were being raised when we tested the interface:


**Coding**

We could now start writing some code. Being event driven, we decided to use eventlib. Our initial code merely consisted of initialising the Toolbox and eventlib and then registering our handlers. At this point we just needed some quit handlers (for the event generated by the Quit menu option and for the Wimp messages) and a handler to fill in the version string on the ProgInfo box.

Note the use of wimplib to provide easy access to the Wimp SWIs.

(from main.c)

```
static void app_init(void)
{
     /* initialise as a toolbox task */
     _kernel_oserror *e;
     if ((e=toolbox_initialise(0,310, messages, tbcodes,
          "<hyper$dir>",&mbl, &idblk,0,0,0)) != NULL) {
        wimp_report_error(e,0,0,0,0,0);
        exit(1);
     }

     /* initialise event lib */

     event_initialise(&idblk);

     /* not interested in nulls or keypresses- the toolbox
        handles all our keyboard shortcuts */

     event_set_mask(1+256);

     /* register events */


     event_register_message_handler(Wimp_MQuit,quit_handler,0);
     event_register_toolbox_handler(-1,Quit_Quit,
                                    tbquit_handler,NULL);

}
```

```
(from handler.c)
  int tbquit_handler(int event_code, ToolboxEvent  *event,
                     IdBlock *id_block, void  *handle)
  {
    IGNORE(event);
    IGNORE(event_code);
    IGNORE(handle);
    IGNORE(id_block);
    quit =1;
    return 1;
  }
  int quit_handler(WimpMessage *message, void *handle)
  {
    IGNORE(message);
    IGNORE(handle);
    quit =1;
    return 1;
  }
  int proginfo_show(int event_code, ToolboxEvent  *event,
                    IdBlock *id_block, void  *handle)
  {
      IGNORE(handle);
      IGNORE(event);
      IGNORE(event_code);
      proginfo_set_version(0,id_block->self_id,
                           lookup_token("Version"));
      return 1;
  }
```

## File loading

Next we turned our attention to file loading. This involved coping with Filer_Open messages on HCL files and files that are dragged to the iconbar icon. To do this we registered some more Wimp message handlers.

```
(from main.c)
       event_register_message_handler(Wimp_MDataOpen,file_loader,0);
       event_register_message_handler(Wimp_MDataLoad,file_loader,0);
(from file.c)
```

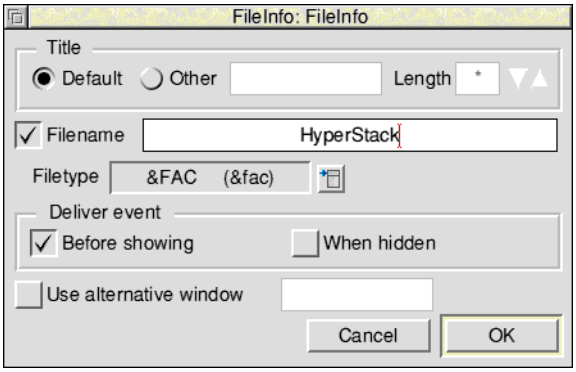
```
int file_loader(WimpMessage *message, void *handle)
{
     /* only interested in HCL files */
  WimpMessage msg;
  IGNORE(handle);

  if (message->data.data_open.file_type != 0xfac) return 0;

  msg = *message;

  msg.hdr.your_ref = msg.hdr.my_ref;

  load_hcl_file(msg.data.data_load_ack.leaf_name);

  if (message->hdr.action_code == Wimp_MDataLoad)
      msg.hdr.action_code = Wimp_MDataLoadAck;
      wimp_send_message(Wimp_EUserMessage,&msg, msg.hdr.sender,0,0);

  return 1;
}
```

## Handling views

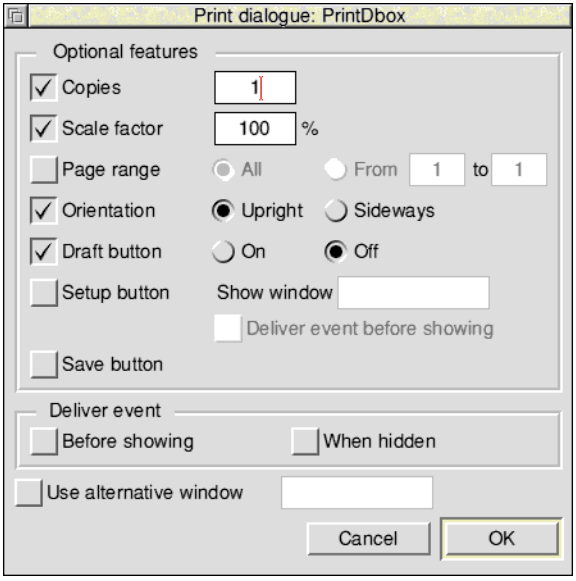
Now it was time to open a viewer onto a file. This involved going back to our resource file and adding some more object templates:

- a window object template to view the files in, which we called HyperViewer
- a menu to be shown on the viewer, which we called ViewerMenu
- attached to this menu a FileInfo box, a Scale box and a PrintDbox object template.

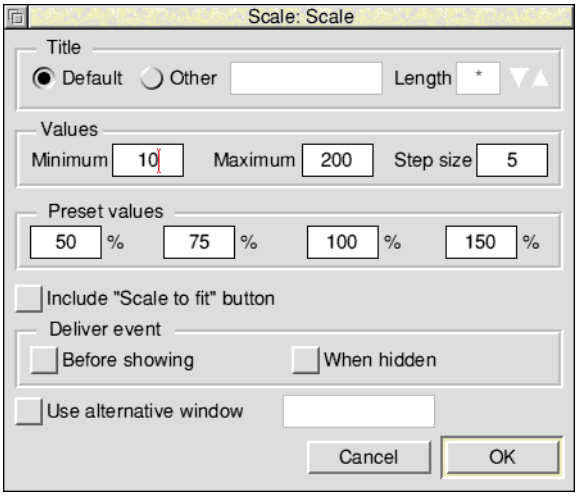
The dialogue box for FileInfo we filled in as follows (note that we switched on **Deliver event Before showing**):




The dialogue box for Print we filled in as follows:

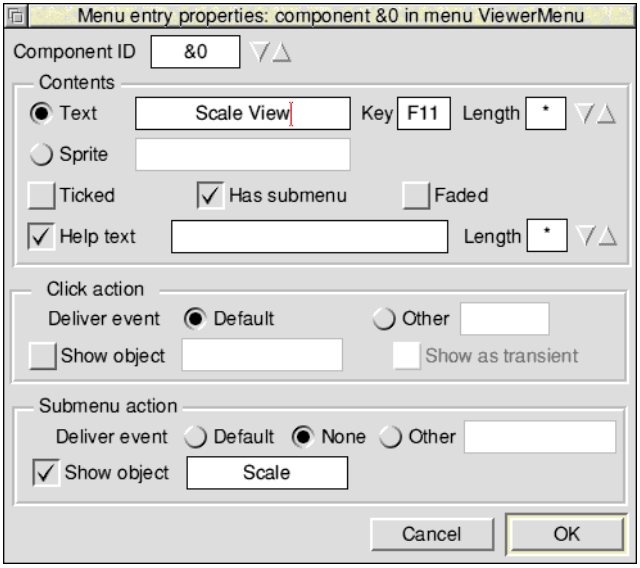


We changed the default values in the dialogue box for Scale as follows:




We then edited `ViewerMenu`, dragging the above three object templates to the **Show object** options in the appropriate Menu entry properties boxes.

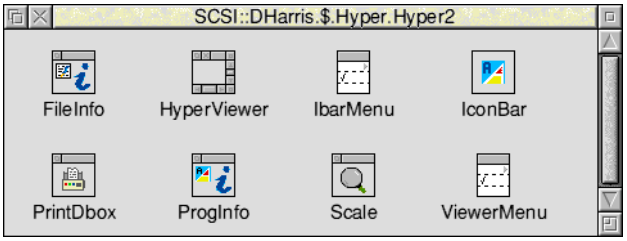
For example, the **Scale View** Menu entry properties box:



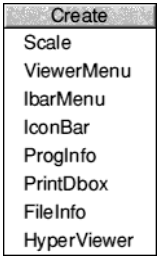

Having filled in all three menu entries, we then edited the HyperViewer window object template. We dragged `ViewerMenu` to the **Show menu** field, and filled in the other window properties boxes as appropriate.

Note that, to receive redraw events, we switched off the **Auto-redraw** flag in the Other properties dialogue in the HyperViewer window. This will affect the appearance in !ResTest and so, for the purposes of this demonstration, is left on.

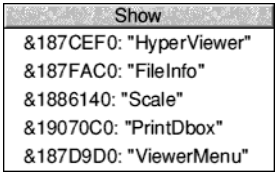
Our resource file display now looked like this:




After connecting them we dragged the resource file to !ResTest. Our icon appeared on the iconbar as before, but now when we pressed Menu over !ResTest's icon and looked at the Create submenu, we saw all the new object templates that we added.



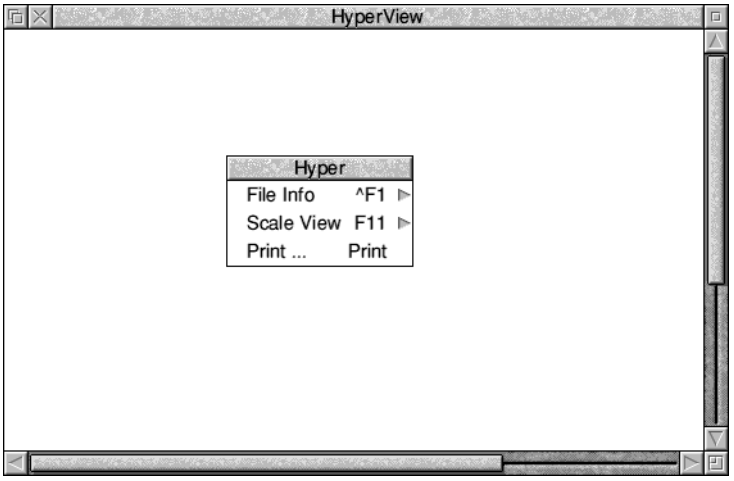

We then clicked on **HyperViewer** to create a viewer. This also unfaded the **Show** option and allowed us to go into the Show submenu and see all the object ids that had been created:



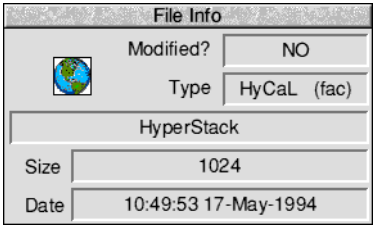

The Show submenu has three columns:

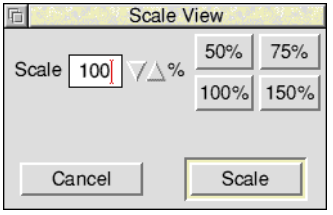
- the first indicates (via a tick) whether the object is showing
- the second is the unique identifier for a particular object – called the object id
- the third is the name of the template from which it was created.

When we clicked on the HyperViewer entry in the Show submenu the viewer was displayed on the screen. As a side effect of the creation the menu tree for the viewer was created as well. Pressing Menu over the viewer displayed the menu as one would expect:




Moving the pointer over the submenu arrows displayed the **File Info** and **Scale View** dialogue boxes:




Clicking on **Print ...** displayed the Print dialogue persistently:


The code to support these new features can be found in the C files under the !Hyper directory of the examples. As with the code fragments above, they take the form of registering a handler for a specific event in app_init (e.g. FileInfo_AboutToBeShown)

and then handling the event elsewhere. Note that the print code is an essentially standard print job/render loop, differing only in that it uses the DrawFile module to do the rendering. See print.c for more information on this.

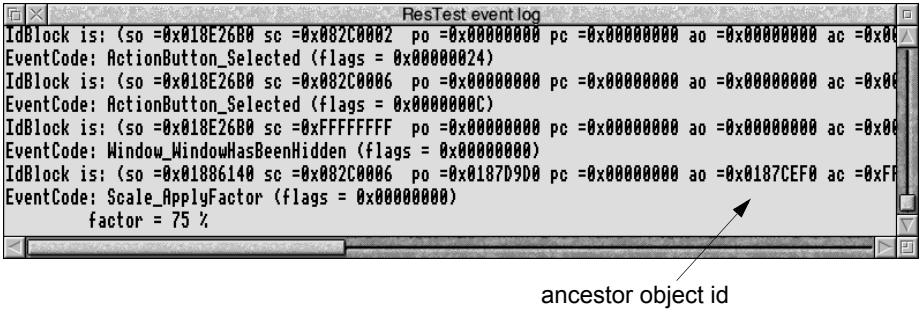
For the viewer (see view.c) we create a window object from a template (called HyperView, as seen in the !ResTest menu) and attach various handlers to cope with RedrawRequests and CloseWindow requests. Note that there is no need to register for OpenWindow requests as this is done on our behalf by the toolbox (as we set the AutoOpen bit of the window's template). We also register for mouse click events on the window. The relevant handler (click_viewer) sets input focus to the window and if applicable jumps to a new card.

**Redraw handler**

The redraw handler (in draw.c) is a standard Wimp redraw handler that uses the DrawFile module to render into the window. Note that the DrawFile module is a generic renderer (i.e. not Wimp specific) and so needs absolute coordinates and a transformation matrix. We use the latter in the simplest sense – just as a way of scaling the Draw files.

**Scaling**

The scaling is set whenever the user clicks scale on the Scale box. If you have the !ResTest Event log window open with the Resource file loaded, you will see that a 'Scale_ApplyFactor' event is generated. We use this in a handler (in draw.c) to adjust the transformation matrix.




The object id for the ancestor of the Scale_ApplyFactor event in this example is &187CEF0. This equates to the object id of HyperViewer (as shown in the Show submenu on page 52). This is because the viewer is the ancestor of this menu. The usefulness of this becomes apparent when more than one viewer object is shown.

### Implementing hotspots

To implement the hotspots on a view, we add gadgets (components of a Window Object) to our viewer window. We use the simplest gadget type, a button gadget, which is quite close in functionality to a Wimp icon (see `button.c`). Rather than hard code the definition of the gadget into the code, Window_ExtractGadgetInfo is used to get the basic gadget definition from a window template called 'Properties'.

### Linking the data structures

Not surprisingly, we link all the data structures for the loaded files together on a linked list. However, we do not need to search down this list every time an event happens: by using client handles (see `view.c`) we can attach the address of the relevant structure to an object. In this way, when we get a redraw event, we just find out the client handle of the viewer on which it happened and can determine what Draw files are to be rendered.

This also works for the menu tree; even though we are sharing the menu tree amongst all the open views, the IdBlock that initialised the toolbox is filled in with the ancestor of the tree. In Hyper, that will be a viewer (we set the Ancestor bit of the HyperView template). So, for example, when we receive a Scale_ApplyFactor event (as in *Scaling* on page 54), the ancestor is the viewer that leads to the scale object being shown. This also applies to PrintDboxes, even though they are shown persistently.

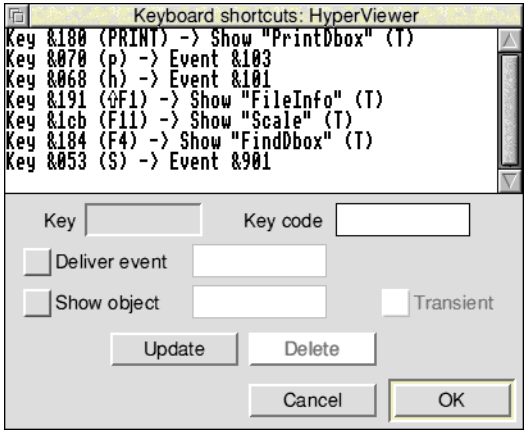
### Showing and hiding views

As we want a history of all views, we build a 'Views' submenu which will be off the icon bar menu. In common with other applications we want the ability to show a view and remove one from memory. In both cases the list of views is the same. This allows us to take advantage of shared objects again. We just need one menu that we build up entry by entry and make this a submenu of the 'Remove View' and 'Show View' entries that are added to the iconbar menu. When an event happens on this menu, we just need to find out the parent component (from the IdBlock) to determine whether we are removing or showing a view. We can also use another useful toolbox feature, in that it is the client that chooses the component ids. This means we can choose the address of the structure that defines a view as its component id – allowing very easy association between the menu entry and the view it refers to. Note that by having an about to be shown event enabled for the iconbar menu, it was possible to fade or unfade the 'Show view' and 'Remove view' entries as required (simply by checking whether our linked list was NULL).

### Adding keyboard short-cuts

With the interface beginning to stabilise, it was possible to start adding some of the keyboard short-cuts. These were generally decided by the Style Guide (e.g. F11 for scale), though some aspects of the interface required keys specific to Hyper (e.g.

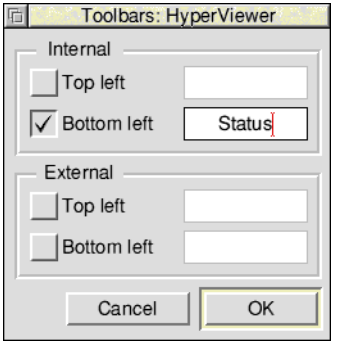
previous and home) to generate events. All this was handled through !ResEd (using the keyboard short-cuts option from the window object template menu) without any additional code requirement.




**Adding a status bar**

A status bar was also provided by creating a Toolbar containing a button gadget:


This Toolbar object template was then dragged to the Toolbars dialogue box from the HyperViewer window:

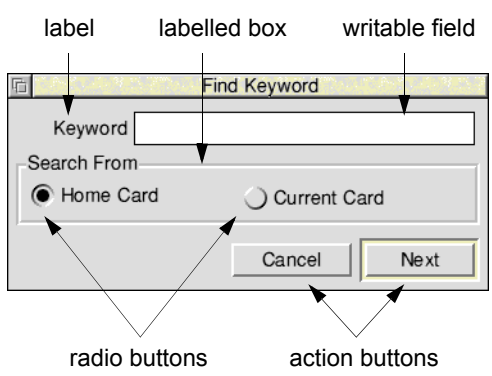


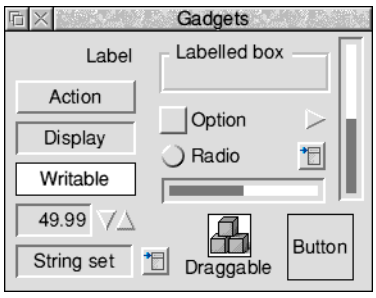

By using an internal bottom left toolbar, the parent window could be resized whilst still allowing the status to be visible. Previous and home action buttons were added (generating the same event codes as the keyboard short-cuts, so no additional code was required) as well.

To control the visibility of the status bar, a menu entry (and appropriate keyboard short-cut) was added that would tick according to whether the status was showing. The handler for this is in handler.c. Note that since the status is on a per-viewer basis, we need to know when the viewer menu is opened (and over what viewer) to determine whether the option should be ticked or not.

### Adding a find capability

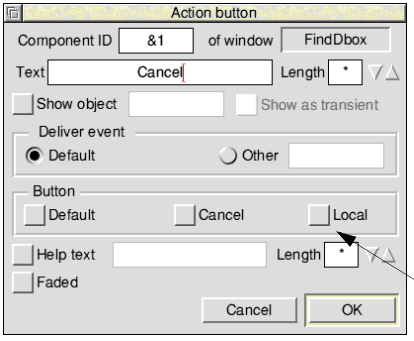
Finally, to provide a find capability, a custom dialogue was designed using !ResEd starting from a basic Window and adding gadgets from the gadgets window:

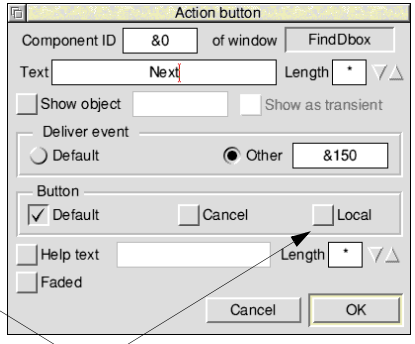



gadgets window

The properties dialogues for the two action buttons were:

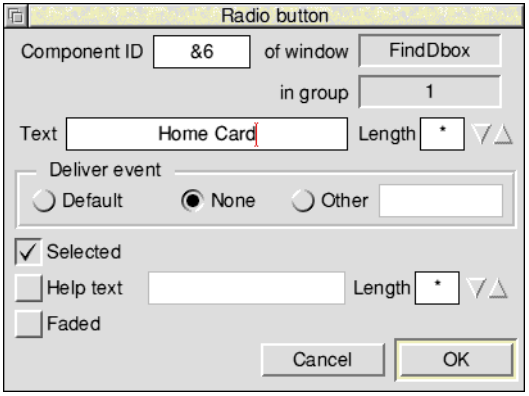



leaving the **Local** options switched off results in the Toolbox automatically closing the dialogue box when clicked on

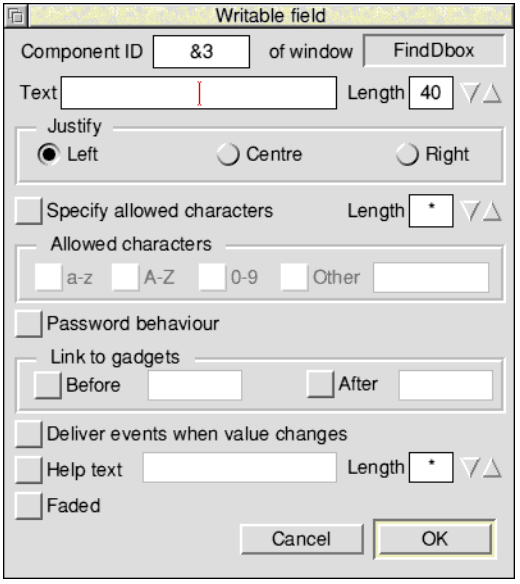
The **Next** action button was made the default and assigned a specific event code.

The **Home Card** radio button properties dialogue was filled in as follows (this radio button was specified as the selected radio button):




The **Current Card** radio button properties dialogue was edited to be similar to the Home Card radio button, except that it was not specified as the selected radio button.

The **Keyword** writable field properties dialogue was filled in as follows:




After choosing suitable components and event codes, the handler code can be written in a self contained unit.

## Modifying the interface

One of the original requirements was that it should be easy to modify the interface to !Hyper. By taking an event driven approach, it is possible to make significant changes to the User Interface, without altering the code. Alternatively, when adding new functionality, this can be done in a modular fashion by adding the required handlers and registering them when required.

### Adding an export DrawFile facility

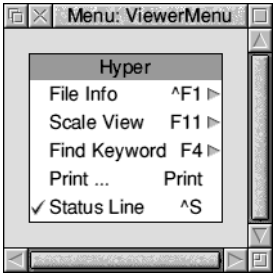
As an example, consider adding an export DrawFile facility. This would allow saving away the Draw files that make up the card on show in the viewer. The best way to implement this would be:

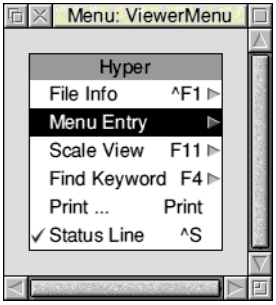
- add a new submenu to the main menu, and call this new submenu `File`
- create two menu entries in this submenu; the first entry will replace the FileInfo menu entry currently on the main menu; the second entry would provide an export facility (implemented using a simple SaveAs dialogue).

This can be achieved easily by some very simple editing of the resource file:

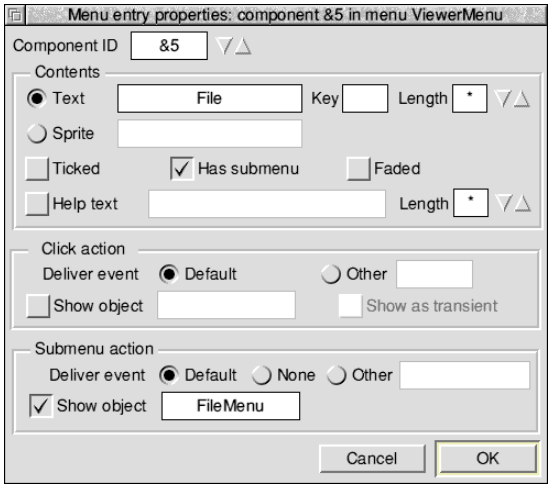
**1** Drag a Menu object template from the Object prototype window to the resource file, and rename the object template to `FileMenu`.

**2** Edit ViewerMenu and add a new menu entry to it:

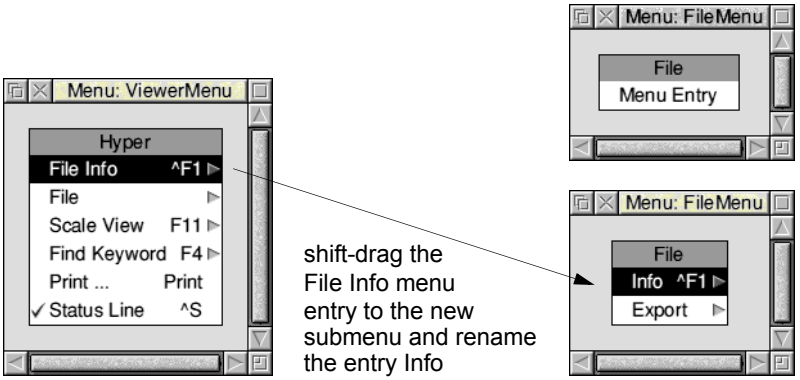


Now edit the new menu entry and rename it to `File`. Then drag the new menu object template `FileMenu` to the **Show object** option:

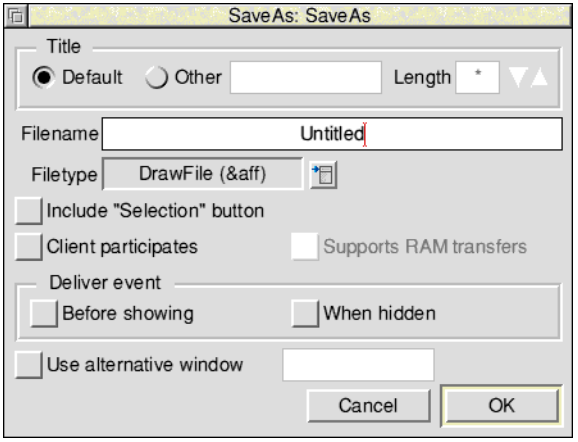



**3** Next double-click on the FileMenu object template. Rename the title `File`, and then Shift-drag the **File Info** menu entry from `ViewerMenu` to it. To make the copied menu entry Style Guide compliant rename it to **Info**:

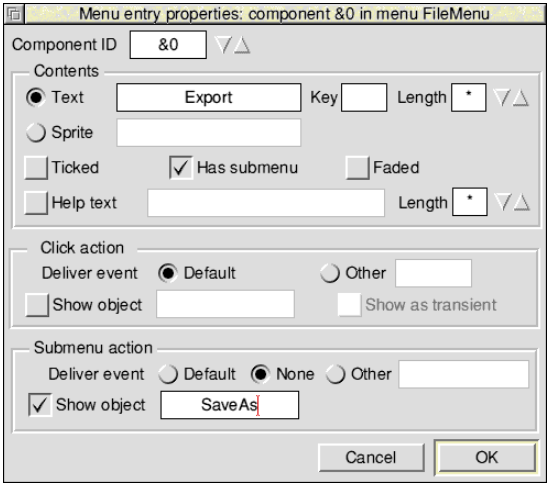



Moving the **File Info** menu entry from ViewerMenu to the new File submenu is a very simple way of relocating this menu option from one menu to another. As we rely on the FileInfo_AboutToBeShown event, it doesn't matter where it is in the interface; it will still work.

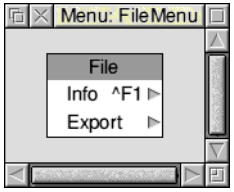
**4** Now drag a SaveAs object template from the Object prototype window to the resource file. Edit this object template to specify that the filetype should be DrawFile:




**5** Finally return to the File menu and create an Export menu entry (by renaming the default entry title `Menu Entry` to `Export`). Edit this entry and drag the SaveAs object template to the **Show object** option:

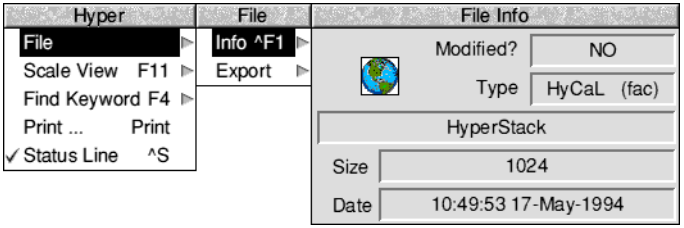


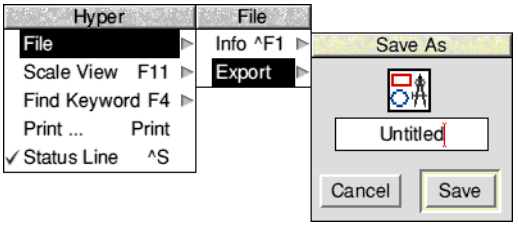
The final submenu should now appear as follows:




The code for the export facility would consist of registering for the various toolbox events and then handling them in a separate area of the code.

If you now dragged the resource file to ResTest, you would see:




**Other possible modifications**

By this time the viewer menu could begin to get cluttered. It would then be very easy to drag off some of the entries to a separate 'Utilities' submenu. Again, being event driven and remembering that the handlers operate on the Ancestor of the menu tree, they will continue to work without code alteration.

Making a demo version of Hyper could be achieved by removing or fading parts of the interface with !ResEd.

### Client Events

A number of events were used in Hyper that were 'Client specified'. These are listed here to help understand properties and output in !ResEd and !ResTest.

| Event number | Usage |
|---|---|
| &101 | Go to Home card |
| &103 | Go to previous card |
| &150 | Start find operation |
| &151 | Iconbar menu is about to be shown |
| &900 | Viewer menu is about to be shown |
| &901 | Toggle status bar |

Other standard events were enabled for dialogues being shown, Print etc.

### Summary

This chapter has demonstrated the following features of the toolbox:

| Toolbox feature | see section/file |
|---|---|
| shared objects and client handles | *Shared objects and client handles* on page 42 |
| About to be shown events | *AboutToBeShown events* on page 42 |
| adding and removing gadgets at run-time | button.c (see *Implementing hotspots* on page 55) |
| creating objects from a template | view.c (see page 54) |
| auto creation | *AboutToBeShown events* on page 42 |
| the Draw file renderer | draw.c (see page 54) |
| event handling with eventlib | *Coding* on page 47 |
| Menu handling | *Creating a basic resource file* on page 44 |
| keyboard short-cuts | *Adding keyboard short-cuts* on page 55 |
| client specified events and component ids | *Showing and hiding views* on page 55 |

## HyperCard Control Language

HyperCard Control Language (HCL) is used by !Hyper to control which draw files are displayed to the user and when jumps should be made to new cards. It is beyond the scope of this example to describe an editor, so the following section is provided to describe the commands that are used.

## HCL commands

All card definitions are enclosed within start and end directives:

```
!!start name
...
!!end
```

where name is cardXXXXXXXX, XXXXXXXX being an 8 digit hex number.

Other commands are as follows:

| Command | Action |
|---|---|
| `button` *`bbox name`* | sets up a hotspot at the given position and sets its behaviour to go to the named card when clicked on |
| `clear` | removes all buttons and Draw files from the viewer window |
| `colour` *`n`* | sets the background colour to the given decimal value |
| `gosub` *`name`* | allows 'inclusion' of common functionality |
| `goto` *`name`* | allows common ending of cards |
| `keyword` *`string`* | sets keyword(s) for this card – allows searching with the find dialogue box |
| `load` *`file`* | loads a file into the bottom layer – overlay will do this if it follows a clear |
| `overlay` *`file`* | loads a draw file into the next available layer |
| `stack` *`string`* | sets the name of this stack to the given string. This will appear in the iconbar menu |
| `status` *`string`* | changes the status line to the given string |
| `title` *`string`* | sets the title bar to the given string |

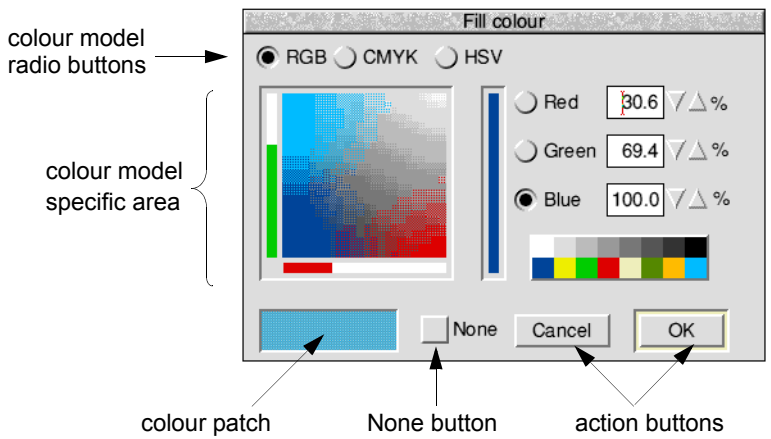
There are also a number of commands that are only used by an editor. These are not described here as they are not required by !Hyper.

# 3 Colour Dialogue box class

A Colour Dialogue box object allows the user to specify a colour using a variety of colour models.

## User interface

The colour selection window can be described as follows:




- At the top is a row of radio buttons – these select which colour model is being used.
- In the middle is an area defined by the current colour model – details of this are described overleaf.
- At the bottom of the window is the colour patch, an optional **None** button which controls transparency, and the window's action buttons.

# Application Program Interface

## Attributes

A Colour Dialogue object has the following attributes which are specified in its object template and can be manipulated at run-time by the client application:

| Attributes | Description | |
|---|---|---|
| flags | **Bit** | **Meaning** |
| | 0 | when set, this bit indicates that a ColourDbox_AboutTobeShown event should be raised when SWI Toolbox_ShowObject is called for this object. |
| | 1 | when set, this bit indicates that a ColourDbox_DialogueCompleted event should be raised when the Colour Dialogue object has been removed from the screen. |
| | 2 | when set, include a **None** button in the dialogue box |
| | 3 | when set, select the **None** button when the dialogue box is created |
| title | this gives an alternative string to use instead of the string 'Colour Choice' in the title bar of the dialogue box (0 means use default) | |
| max title length | this gives the maximum length in bytes of title text which will be used for this object | |
| colour | an RGB value for the initial colour value | |

Note that it is possible to set and read whether a Colour Dialogue has a **None** entry at run-time using the following methods (described on page 74):

ColourDbox_SetNoneAvailable
ColourDbox_GetNoneAvailable

## Manipulating a Colour Dialogue object

### Creating and deleting a Colour Dialogue object

A Colour Dialogue object is created using SWI Toolbox_CreateObject.

When this object is created it has no attached objects (see page 11).

A Colour Dialogue object is deleted using SWI Toolbox_DeleteObject.

The setting of the non-recursive delete bit does not have a meaning for Colour Dialogue objects.

### Showing a Colour Dialogue object

When a Colour Dialogue object is displayed on the screen using SWI Toolbox_ShowObject it has the following behaviour:

| Show type | Position | |
|---|---|---|
| 0 (default) | the underlying window is shown at the last place shown on the screen, or the coordinates given in its template, if it has not already been shown | |
| 2 (topleft) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |

For most applications it will not be necessary to make these calls explicitly, but instead to mark the templates with their auto-create bit set, so that a Colour Dialogue object is created on start-up.

### Before the dialogue box is shown

When the client calls Toolbox_ShowObject, a ColourDbox_AboutToBeShown Toolbox event is raised (if the appropriate flags bit is set), allowing the client to take any last minute action. Typically, a client will indicate which of the colours should be shown as the currently selected one, when it receives this event.

### Setting and reading the colour used in a Colour Dialogue box

It is possible for the colour which is currently selected in the dialogue box to be set by the client application. This is independent of the colour model being used, since the colour is specified as an RGB colour value. The client passes a ‘colour block’ to the Colour Dialogue module which has a one-word RGB value as its first word; the remainder of the block is intended to support any future colour models other than RGB, CMYK and HSV. It has a size field followed by colour-model-specific data. For clients not requiring this extensibility, the size field should be set to 0. The method for setting the colour thus used in a Colour Dialogue is ColourDbox_SetColour.

The current colour (and colour model data) can be read using the ColourDbox_GetColour method (described on page 71).

### Setting and reading the colour model used in a Colour Dialogue

The colour model used in a Colour Dialogue is normally chosen by the user by clicking on the appropriate radio button. The client can however set this at run-time using the ColourDbox_SetColourModel method, passing a colour number (RGB=0, CMYK=1, HSV=2). If any other colour model is required, then further colour-model-specific data must also be passed to this method (none are currently supported).

The current colour model used can be read using the ColourDbox_GetColourModel method.

**Reacting to colour selections**

When the user has found the correct colour he wants, he will click the **OK** button in the Colour Dialogue box. The Colour Dialogue module delivers a ColourDbox_ColourSelected Toolbox event to the client at this point giving the RGB value of the colour chosen.

**Completion of a Colour Dialogue**

When the Colour Dialogue module has hidden its dialogue box at the end of a dialogue, it delivers a ColourDbox_DialogueCompleted Toolbox event to the client, with an indication of whether a colour selection occurred during the dialogue.

## Colour Dialogue methods

The following methods are all invoked by calling SWI Toolbox_MiscOp with:

| | |
|---|---|
| R0 | holding a flags word |
| R1 | being a Colour Dialogue id |
| R2 | being the method code which distinguishes this method |
| R3-R9 | potentially holding method-specific data |

### ColourDbox_GetWimpHandle 0

**On entry**

R0 = flags
R1 = Colour Dbox object id
R2 = 0

**On exit**

R0 = Wimp window handle of underlying window

**Use**

This method returns the Wimp window handle of the window used by the underlying Colour Picker module to implement the Colour dialogue. The value returned is only valid when the Colour dialogue box is showing.

**C veneer**

```
extern _kernel_oserror *colourdbox_get_wimp_handle ( unsigned int flags,
                                                     ObjectId colourdbox,
                                                     int *wimp_handle
                                                   );
```

## ColourDbox_GetDialogueHandle 1

**On entry**

R0 = flags
R1 = Colour Dbox object id
R2 = 1

**On exit**

R0 = ColourPicker dialogue handle of underlying dialogue box

**Usage**

This method returns the handle of the dialogue box used by the underlying Colour Picker module to reference the Colour dialogue. The value returned is only valid when the Colour dialogue box is showing.

**C veneer**

```
extern _kernel_oserror *colourdbox_get_dialogue_handle ( unsigned int flags,
                                                         ObjectId colourdbox,
                                                         int *dialogue_handle
                                                       );
```

## ColourDbox_SetColour 2

**On entry**

R0 = flags
bit 0 set ⇒ select the **None** option
R1 = Colour Dbox object id
R2 = 2
R3 = pointer to colour block

**On exit**

R1-R9 preserved

**Use**

This method sets the colour currently displayed in the Colour Dialogue (adjusting the colour slice shown, the sliders, and the writable fields appropriately).

The colour block is defined as follows:

| | |
|---|---|
| +0 | 0 |
| +1 | blue value (0, ..., &FF) |
| +2 | green value |
| +3 | red value |
| +4 | size of the remainder of this block (which may be 0) |
| +8 | colour model number |
| +12... | other model-dependent data |

Currently there are no extra colour models supported, so the size field at byte offset 4 should be set to 0.

If bit 0 of the flags word is set (select the **None** option) then R3 may be 0.

**C veneer**

```
extern _kernel_oserror *colourdbox_set_colour ( unsigned int flags,
                                                ObjectId colourdbox,
                                                int *colour_block
                                              );
```

## ColourDbox_GetColour 3

**On entry**

R0 = flags
R1 = Colour Dbox object id
R2 = 3
R3 = pointer to buffer for colour block
R4 = size of buffer

**On exit**

if bit 0 of R0 is set ⇒ **None** is selected

R4 = size of buffer required (if R3 was 0)
(currently fixed because no extra colour models are supported)
else buffer pointed at by R3 contains colour information
R4 holds number of bytes written to buffer.

**Use**

This method returns the colour currently displayed in the Colour Dialogue.

The colour block is defined as follows:

| | |
|---|---|
| +0 | 0 |
| +1 | blue value (0, ..., &FF) |
| +2 | green value |
| +3 | red value |
| +4 | size of the remainder of this block (which may be 0) |
| +8 | colour model number |
| +12... | other model-dependent data |

**C veneer**

```
extern _kernel_oserror *colourdbox_get_colour ( unsigned int flags,
                                                ObjectId colourdbox,
                                                int *buffer,
                                                int buff_size,
                                                int *nbytes
                                              );
```

## ColourDbox_SetColourModel 4

**On entry**

R0 = flags
R1 = Colour Dbox object id
R2 = 4
R3 = pointer to colour model block

**On exit**

R1-R9 preserved

**Use**

This method sets the colour model currently used in the Colour Dialogue. The colour which is being displayed will now be shown using the new colour model, and the layout of the dialogue box will change accordingly.

The colour model block is defined as follows:

| | |
|---|---|
| +0 | size of the remainder of this block (currently only 4) |
| +4 | colour model number |
| +8... | other model-dependent data |

The current valid colour model numbers are:

| | |
|---|---|
| 0 | RGB |
| 1 | CMYK |
| 2 | HSV |

Currently there are no extra colour models supported, so the size field at byte offset 0 should be set to 4 (i.e. just a colour model number).

**C veneer**

```
extern _kernel_oserror *colourdbox_set_colour_model ( unsigned int flags,
                                                      ObjectId colourdbox,
                                                      int *colour_model_block
                                                    );
```

## ColourDbox_GetColourModel 5

**On entry**

R0 = flags
R1 = Colour Dbox object id
R2 = 5
R3 = pointer to buffer for colour block
R4 = size of buffer

**On exit**

R4 = size of buffer required (if R3 was 0)
(currently fixed because no extra colour models are supported)
else buffer pointed at by R3 contains colour information
R4 holds number of bytes written to buffer

**Use**

This method returns the number of the colour model currently used in the Colour Dialogue.

The colour model block is defined as follows:

| | |
|---|---|
| +0 | size of the remainder of this block |
| +4 | colour model number (currently: 0 = RGB, 1 = CMYK and 2 = HSV) |
| +8... | other model-dependent data |

**C veneer**

```
extern _kernel_oserror *colourdbox_get_colour_model ( unsigned int flags,
                                                      ObjectId colourdbox,
                                                      int *buffer,
                                                      int buff_size,
                                                      int *nbytes
                                                    );
```

## ColourDbox_SetNoneAvailable 6

**On entry**

R0 = flags
R1 = Colour Dbox object id
R2 = 6
R3 = non-zero means **None** is available

**On exit**

R1-R9 preserved

**Use**

This method sets whether a **None** option appears in the Colour Dialogue.

**C veneer**

```
extern _kernel_oserror *colourdbox_set_none_available ( unsigned int flags,
                                                        ObjectId colourdbox,
                                                        int none
                                                      );
```

## ColourDbox_GetNoneAvailable 7

**On entry**

R0 = flags
R1 = Colour Dbox object id
R2 = 7

**On exit**

if bit 0 of R0 is set, then **None** is available

**Use**

This method returns whether the **None** option appears in a Colour Dialogue.

**C veneer**

```
extern _kernel_oserror *colourdbox_get_none_available ( unsigned int flags,
                                                        ObjectId colourdbox,
                                                        int *out_flags
                                                      );
```

## Colour Dialogue events

There are a number of Toolbox events which are generated by the Colour Dialogue module:

### ColourDbox_AboutToBeShown (0x829c0)

**Block**

| | |
|---|---|
| + 8 | 0x829c0 |
| + 12 | flags (as passed in to Toolbox_ShowObject) |
| + 16 | value which will be passed in R2 to ToolBox_ShowObject |
| + 20... | block which will be passed in R3 to ToolBox_ShowObject for the underlying dialogue box. |

**Use**

This Toolbox event is raised when SWI Toolbox_ShowObject has been called for a Colour Dialogue object. It gives the application the opportunity to set fields in the dialogue box before it actually appears on the screen.

**C data type**

```
typedef struct
{
  ToolboxEventHeader  hdr;
  int                 show_type;
  union
  {
    TopLeft               pos;
    WindowShowObjectBlock full;
  } info;
} ColourDboxAboutToBeShownEvent;
```

## ColourDbox_DialogueCompleted (0x829c1)

**Block**

+ 8 0x829c1
+ 12 flags
bit 0 set means that a colour selection was done during this dialogue

**Use**

This Toolbox event is raised after the Colour Dialogue object has been hidden, either by a **Cancel** click, or after an **OK** click, or by the user pressing Escape. It allows the client to tidy up its own state associated with this dialogue.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
} ColourDboxDialogueCompletedEvent;
```

## ColourDbox_ColourSelected (0x829c2)

**Block**

+ 8 0x829c2
+ 12 flags bit 0 set means **None** was chosen
+ 16 colour block chosen

**Use**

This Toolbox event is raised when the user clicks **OK** in the dialogue box. The colour block has the same format shown in the ColourDbox_SetColour method.

Note that event if the **None** button is set, a colour value is still returned, reflecting the current state of the dialogue box.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  unsigned int       colour_block[(212/4)];
} ColourDboxColourSelectedEvent;
```

## Colour Dialogue templates

The layout of a Colour Dialogue template is shown below. Fields which have types MsgReference and StringReference are those which will require relocation when they are loaded from a resource file. If the template is being constructed in memory, then these fields should be real pointers (i.e. they do not require relocation).

For more details on relocation, see appendix *Resource File Formats* on page 499.

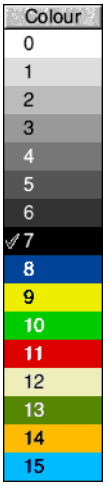
| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| title | 4 | MsgReference |
| max_title | 4 | word |
| colour | 4 | word |

# 4 Colour Menu class

A Colour Menu object is used to show a menu giving the 16 desktop colours (and an optional **None** entry), and to allow the user to select one of these colours by clicking on its menu entry.

## User interface

The Colour Menu allows the user to select from the set of available desktop colours (and an optional **None** entry which appears at the bottom). The menu is displayed showing the 16 desktop colours. Optionally any one of the colours can be shown as selected (with a tick against it).




When a hit is received for the Colour Menu, a Toolbox event is returned to the client. This contains the colour number of the selected colour. The selected colour is shown as ticked in the Colour Menu, when the menu is next shown (or immediately if Adjust is held down).

# Application Program Interface

## Attributes

A Colour Menu object has the following attributes which are specified in its object template and can be manipulated at run-time by the client application:

| Attribute | Description |
|---|---|
| flags word | **Bit Meaning**<br>0 when set, this bit indicates that a ColourMenu_AboutToBeShown event should be raised when SWI Toolbox_ShowObject is called for this Colour Menu<br>1 when set, this bit indicates that a ColourMenu_HasBeenHidden event should be raised when the Menu has been removed from the screen<br>2 when set, include a **None** entry in the menu (will appear with **None** as its last entry) |
| menu title | this gives an alternative string to use instead of the string 'Colour' in the title bar of the menu |
| max title length | this gives the maximum length in bytes of title text which will be used for this Colour Menu. |
| colour | this is an indication of which colour is selected when the Colour Menu is first created. Possible values are:<br>0-15 for the desktop colours<br>16 for 'None'<br>-1 to indicate that no colour should be selected |

## Manipulating a Colour Menu object

**Creating and deleting a Colour Menu**

A Colour Menu object is created using SWI Toolbox_CreateObject.

When this object is created it has no attached objects (see page 11).

A Colour Menu object is deleted using SWI Toolbox_DeleteObject.

The setting of the non-recursive delete bit does not have a meaning for Colour menus.

### Showing a Colour Menu

When a Colour menu is displayed on the screen using SWI Toolbox_ShowObject it has the following behaviour:

| Show type | Position |
|---|---|
| 0 (default) | 64 OS units to the left of the mouse pointer |
| 1 (full spec) | R3 + 0 gives x coordinate of top-left corner of Menu<br>R3 + 4 gives y coordinate of top-left corner of Menu |
| 2 (topleft) | R3 + 0 gives x coordinate of top-left corner of Menu<br>R3 + 4 gives y coordinate of top-left corner of Menu |

### Before the menu is shown

When the client calls Toolbox_ShowObject, a ColourMenu_AboutToBeShown Toolbox event is raised (if the appropriate flags bit is set), allowing the client to take any last minute action. Typically, a client will indicate which of the colours should be shown as the currently selected one, when it receives this event.

### Setting and getting the selected colour

For a Colour Menu, one of the colour entries can be designated the selected colour (indicated by a tick against it in the menu). Colours within the menu are numbered like the Wimp colours from 0-15 (with 16 meaning ‘None’, and -1 meaning ‘nothing selected’).

The currently selected colour entry can be set and read dynamically using the ColourMenu_SetColour/ColourMenu_GetColour methods.

Note that when the user clicks on a colour entry, that will become the selected colour automatically without calling ColourMenu_SetColour. As will be seen later, a user click results in a Toolbox event being delivered to the client, indicating which colour was selected.

The client can dynamically set whether a **None** entry is given, by using the ColourMenu_SetNoneAvailable method (and read whether it is available using the ColourMenu_GetNoneAvailable method).

### Processing a colour selection

Whenever the user clicks on a colour entry a ColourMenu_Selection Toolbox event is raised to indicate which colour was chosen (one of 0-15, or 16 to indicate ‘None’).

### Getting the underlying menu object id

The object id of the underlying menu object used to implement a Colour Menu can be obtained using the ColourMenu_GetMenuID method (normally you would not need to do this).

## Colour Menu methods

The following methods are all invoked by calling SWI Toolbox_MiscOp with:

| | |
|---|---|
| R0 | holding a flags word |
| R1 | being a Colour Menu id |
| R2 | being the method code which distinguishes this method |
| R3-R9 | potentially holding method-specific data. |

### ColourMenu_SetColour 0

**On entry**

R0 = flags
R1 = Colour Menu object id
R2 = 0
R3 = Wimp colour (0-15, or 16 for 'None', or -1 for 'nothing selected')

**On exit**

R1-R9 preserved

**Use**

This method selects a colour as being the currently selected one for this Colour Menu, and places a tick next to it. Note that this change will only be visible when the Colour Menu is next shown.

**C veneer**

```
extern _kernel_oserror *colourmenu_set_colour ( unsigned int flags,
                                                ObjectId colourmenu,
                                                int wimp_colour
                                              );
```

## ColourMenu_GetColour 1

**On entry**

R0 = flags
R1 = Colour Menu object id
R2 = 1

**Exit**

R0 = Wimp colour selected (0-15, or 16 for 'None', or -1 for 'nothing selected')

**Use**

This method returns the Wimp colour which is currently selected for this Colour Menu.

**C veneer**

```
extern _kernel_oserror *colourmenu_get_colour ( unsigned int flags,
                                                ObjectId colourmenu,
                                                int *wimp_colour
                                              );
```

## ColourMenu_SetNoneAvailable 2

**On entry**

R0 = flags
R1 = Colour Menu object id
R2 = 2
R3 = non-zero means allow a 'None' entry

**On exit**

R1-R9 preserved

**Use**

This method sets whether there is a 'None' entry for this Colour Menu.

**C veneer**

```
extern _kernel_oserror *colourmenu_set_none_available ( unsigned int flags,
                                                        ObjectId colourmenu,
                                                        int none
                                                      );
```

## ColourMenu_GetNoneAvailable 3

**On entry**

R0 = flags
R1 = Colour Menu object id
R2 = 3

**On exit**

R0 = non-zero means there is a ‘None’ entry

**Use**

This method returns whether this Colour Menu has a ‘None’ entry.

**C veneer**

```
extern _kernel_oserror *colourmenu_get_none_available ( unsigned int flags,
                                                        ObjectId colourmenu,
                                                        int *none
                                                      );
```

## ColourMenu_SetTitle 4

**On entry**

R0 = flags
R1 = Colour Menu object id
R2 = 4
R3 = pointer to text string to use

**Exit**

R1-R9 preserved

**Use**

This method sets the text which is to be used in the title bar of the given Colour Menu.

**C veneer**

```
extern _kernel_oserror *colourmenu_set_title ( unsigned int flags,
                                               ObjectId colourmenu,
                                               char *title
                                             );
```

## ColourMenu_GetTitle 5

**On entry**

R0 = flags
R1 = Colour Menu object id
R2 = 5
R3 = pointer to buffer to return the text in (or 0)
R4 = size of buffer

**Exit**

R4 = size of buffer required to hold the text (if R3 was 0)
else Buffer pointed to by R3 contains title text
R4 holds number of bytes written to buffer

**Use**

This method returns the text string used in a Colour Menu's title bar.

**C veneer**

```
extern _kernel_oserror *colourmenu_get_title ( unsigned int flags,
                                               ObjectId colourmenu,
                                               char *buffer,
                                               int buff_size,
                                               int *nbytes
                                             );
```

## Colour Menu events

There are a number of Toolbox Events which are generated by the Colour Menu module:

### ColourMenu_AboutToBeShown (0x82980)

**Block**

| | |
|---|---|
| + 8 | 0x82980 |
| + 12 | flags (as passed in to Toolbox_ShowObject) |
| + 16 | value which will be passed in R2 to ToolBox_ShowObject |
| + 20... | block which will be passed in R3 to ToolBox_ShowObject for the underlying Menu object |

**Use**

This Toolbox event is raised when SWI Toolbox_ShowObject has been called for a Colour Menu object. It gives the application the opportunity to set the selected colour before the menu actually appears on the screen.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                show_type;
  TopLeft            pos;
} ColourMenuAboutToBeShownEvent;
```

### ColourMenu_HasBeenHidden (0x82981)

**Block**

| | |
|---|---|
| + 8 | 0x82981 |

**Use**

This Toolbox Event is raised by the Toolbox when Toolbox_HideObject is called on a Colour Menu which has the appropriate bit set in its template flags word. It enables a client application to clear up after a menu has been closed. It is also raised when clicking outside a menu or hitting Escape.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
} ColourMenuHasBeenHiddenEvent;
```

### ColourMenu_Selection (0x82982)

**Block**

+ 8 0x82982
+ 16 Wimp colour selected (0-15, or 16 for 'None')

**Use**

This Toolbox event is raised when the user has clicked on one of the Colour entries in the Colour Menu. The colour value returned is in the range 0-15 for the desktop colours, or 16 for 'None'.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                colour;
} ColourMenuSelectionEvent;
```

## Colour Menu templates

The layout of a Colour Menu template is shown below. Fields which have types MsgReference and StringReference are those which will require relocation when they are loaded from a resource file. If the template is being constructed in memory, then these fields should be real pointers (i.e. they do not require relocation).

For more details on relocation, see appendix *Resource File Formats* on page 499.

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| title | 4 | MsgReference |
| max-title | 4 | word |
| colour | 4 | word |

## Colour Menu Wimp event handling

The Colour Menu class responds to certain Wimp events and takes the actions as described below:

| Wimp event | Action |
|---|---|
| Menu Selection | The colour number corresponding to the menu selection is sent back to the client via a ColourMenu_Selection event.<br>If Adjust is held down, then the currently open menu is re-opened in the same place. |
| User Msg | Message_HelpRequest<br>(while the pointer is over a Colour Menu object)<br>If a help message is attached to this Colour Menu, then a reply is sent on the application's behalf. |

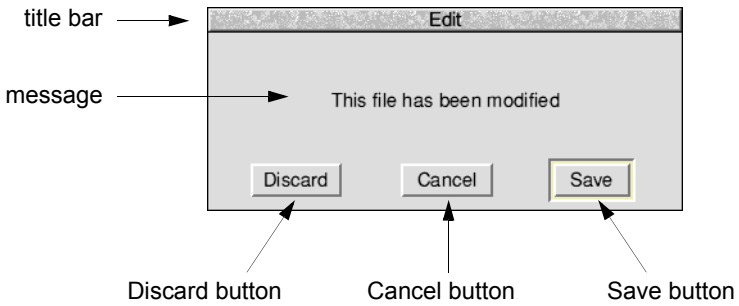
# 5 Discard/Cancel/Save Dialogue box class

A Discard/Cancel/Save (DCS) Dialogue box is used by the client application when the user attempts to close a window containing modified and unsaved data.

## User interface

A DCS dialogue object is used to allow the user to save data which has been modified, usually before a document window is closed.

The dialogue box which appears on the screen has a number of components:




- a title bar (by default containing the name of the application, i.e. the message whose tag is '_TaskName')
- a message stating (by default) that there is unsaved data
- three Action Buttons: **Discard**, **Cancel** and **Save** (default action button).

The user sees the following behaviour (note that a click with the adjust button is treated in the same way as a select click):

- if they click on **Discard**, the box is closed, the parent window is closed, and its (new) contents discarded
- if they click outside the dialogue box (and it was opened transiently, i.e. with Menu semantics), or click on **Cancel**, the box is closed, and the close on the parent window is cancelled

- if they click on **Save** or press Return, the box is closed, and either the data is saved without further interaction (if a suitable full pathname is available), or a SaveAs dialogue appears allowing an icon to be dragged to where the data should be saved. When the save is complete, the parent window is closed.

## Application Program Interface

When a DCS object is created, it has a number of optional components:

- an alternative title bar string instead of the client's name
- an alternative message to use in the dialogue box
- the name of an alternative template to use for the underlying Window object.

Just before the DCS dialogue box is shown on the screen, the client is delivered a DCS_AboutToBeShown Toolbox event if enabled by the flags word.

Once the dialogue box is displayed on the screen, the DCS module handles events for it, and raises a number of Toolbox Events to indicate what choice the user has made. These are DCS_Discard, DCS_Cancel and DCS_Save respectively. If the dialogue is closed, then the client receives a DCS_DialogueCompleted event if enabled by the appropriate bit in the flags word (see below).

### Attributes

A DCS object has the following attributes which are specified in its object template and can be manipulated at run-time by the client application:

| **Attributes** | **Description** | |
|---|---|---|
| flags | **Bit** | **Meaning** |
| | 0 | when set, this bit indicates that a DCS_AboutTobeShown event should be raised when SWI Toolbox_ShowObject is called for this object. |
| | 1 | when set, this bit indicates that a DCS_DialogueCompleted event should be raised when the DCS object has been removed from the screen. |
| DCS title | an alternative string for the title bar other than the client's name (0 means use application name) | |
| max title length | this gives the maximum length in bytes of title text which will be used for this object | |
| message | an alternative message to use in the DCS dialogue box (other than 'This file has been modified') | |

| Attributes | Description |
|---|---|
| max message length | this gives the maximum length in bytes of the message which will be used for this object |
| window | an alternative window template to use instead of the default one (o means use default) |

## Manipulating a DCS object

### Creating and deleting a DCS object

A DCS object is created using SWI Toolbox_CreateObject.

When this object is created it has no attached objects (see page 11).

A DCS object is deleted using SWI Toolbox_DeleteObject.

The setting of the non-recursive delete bit does not have a meaning for DCS objects.

### Showing a DCS object

When a DCS object is displayed on the screen using SWI Toolbox_ShowObject it has the following behaviour:

| Show type | Position | |
|---|---|---|
| 0 (default) | close to the pointer | |
| 1 (full spec) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |
| | R3 + 8 | visible area maximum x coordinate |
| | R3 + 12 | visible area maximum y coordinate |
| | R3 + 16 | scroll x offset relative to work area |
| | R3 + 20 | scroll y offset relative to work area |
| | R3 + 24 | Wimp window handle of window to open behind<br>-1 means top of stack<br>-2 means bottom of stack<br>-3 means the window behind the Wimp's backwindow |
| 2 (topleft) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |

### Changing the DCS dialogue's message

When a DCS dialogue object is created it has a default message warning the user that he has unsaved data which will be lost if he closes the window.

This can be set and read dynamically using the DCS_SetMessage and DCS_GetMessage methods (described on page 93).

**Getting the id of the underlying window for a DCS object**

The window object id of the Window object used to implement the DCS Dialogue can be obtained by using the DCS_GetWindowID method.

## DCS methods

The following methods are all invoked by calling SWI Toolbox_ObjectMiscOp with:

| | |
|---|---|
| R0 | holding a flags word (which is zero unless otherwise stated) |
| R1 | being a DCS Dialogue object id |
| R2 | being the method code which distinguishes this method |
| R3-R9 | potentially holding method-specific data |

### DCS_GetWindowID 0

**On entry**

R0 = flags
R1 = DCS object id
R2 = 0

**On exit**

R0 = Window object id for this DCS object

**Use**

This method returns the id of the underlying Window object used to implement this DCS object.

**C veneer**

```
extern _kernel_oserror *dcs_get_window_id ( unsigned int flags,
                                            ObjectId dcs,
                                            ObjectId *window
                                          );
```

## DCS_SetMessage 1

**On entry**

R0 = flags
R1 = DCS object id
R2 = 1
R3 = pointer to buffer holding new message (Ctrl-terminated)

**On exit**

R1-R9 preserved

**Use**

This method sets the message used in the DCS dialogue's window.

**C veneer**

```
extern _kernel_oserror *dcs_set_message ( unsigned int flags,
                                          ObjectId dcs,
                                          char *message
                                        );
```

## DCS_GetMessage 2

**On entry**

R0 = flags
R1 = DCS object id
R2 = 2
R3 = pointer to buffer to hold message
R4 = size of buffer to hold message

**On exit**

R4 = size of buffer required to hold message (if R3 was 0)
else buffer pointed at by R3 holds message
R4 holds number of bytes written to buffer

**Use**

This method returns the current message used in a DCS object.

**C veneer**

```
extern _kernel_oserror *dcs_get_message ( unsigned int flags,
                                          ObjectId dcs,
                                          char *buffer,
                                          int buff_size,
                                          int *nbytes
                                        );
```

## DCS_SetTitle 3

**On entry**

R0 = flags
R1 = DCS object id
R2 = 3
R3 = pointer to text string to use

**On exit**

R1-R9 preserved

**Use**

This method sets the text which is to be used in the title bar of the given DCS dialogue.

**C veneer**

```
extern _kernel_oserror *dcs_set_title ( unsigned int flags,
                                        ObjectId dcs,
                                        char *title
                                      );
```

## DCS_GetTitle 4

**On entry**

R0 = flags
R1 = DCS object id
R2 = 4
R3 = pointer to buffer to return the text in (or 0)
R4 = size of buffer

**On exit**

R4 = size of buffer required to hold the text (if R3 was 0)
else Buffer pointed to by R3 contains title text
R4 holds number of bytes written to buffer

**Use**

This method returns the text string used in a DCS dialogue's title bar.

**C veneer**

```
extern _kernel_oserror *dcs_get_title ( unsigned int flags,
                                        ObjectId dcs,
                                        char *buffer,
                                        int buff_size,
                                        int *nbytes
                                      );
```

## DCS events

The DCS module generates the following Toolbox events:

### DCS_AboutToBeShown (0x82a80)

**Block**

| | |
|---|---|
| + 8 | 0x82a80 |
| +12 | value which will be passed in R0 to Toolbox_ShowObject (i.e. show flags, such as 'Show as menu') |
| + 16 | value which will be passed in R2 to ToolBox_ShowObject |
| + 20... | block which will be passed in R3 to ToolBox_ShowObject for the underlying dialogue box. |

**Use**

This Toolbox event is raised just before the DCS module is going to show its underlying Window object.

**C data type**

```
typedef struct
{
ToolboxEventHeader hdr;
  int                show_type;
  union
  {
    TopLeft pos;
    WindowShowObjectBlock full;
  } info;

} DCSAboutToBeShownEvent;
```

## DCS_Discard (0x82a81)

**Block**

+ 8 0x82a81

**Use**

This Toolbox event is raised when the user clicks on the **Discard** button.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;

} DCSDiscardEvent;
```

## DCS_Save (0x82a82)

**Block**

+ 8 0x82a82

**Use**

This Toolbox event is raised when the user clicks on the **Save** Button or presses Return. It is then the client's responsibility to either save the data directly to file, or to display a SaveAs Dialogue object.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;

} DCSSaveEvent;
```

## DCS_DialogueCompleted (0x82a83)

**Block**

+ 8 0x82a83

**Use**

This Toolbox event is raised after the DCS object has been hidden, either by a Cancel click, a Save click or a Discard click, or by the user clicking outside the dialogue box (if opened transiently) or pressing Escape. It allows the client to tidy up its own state associated with this dialogue.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;

} DCSDialogueCompletedEvent;
```

## DCS_Cancel (0x82a84)

**Block**

+ 8 0x82a84

**Use**

This Toolbox event is raised when the user clicks on the Cancel button or presses the Escape key.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;

} DCSCancelEvent;
```

## DCS templates

The layout of a DCS template is shown below. Fields which have types MsgReference and StringReference are those which will require relocation when they are loaded from a resource file. If the template is being constructed in memory, then these fields should be real pointers (i.e. they do not require relocation).

For more details on relocation, see appendix *Resource File Formats* on page 499.

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| title | 4 | MsgReference |
| max_title | 4 | word |
| message | 4 | MsgReference |
| max_message | 4 | word |
| window | 4 | StringReference |

### Underlying window template

The window object used to implement a DCS dialogue, has the following characteristics. These must be reproduced if the Window is replaced by a client-specified alternative Window template:

Title bar must be indirected.

**Gadgets**

Component ids are derived by adding to 0x82a800

| Component id | Details |
|---|---|
| 0 | button gadget |
| 1 | action button (**Discard**) |
| 2 | action button (**Cancel**)<br>must be marked as a ‘Cancel’ action button |
| 3 | action button (**Save**)<br>must be marked as a ‘Default’ action button |

## DCS Wimp event handling

| Wimp event | Action |
|---|---|
| Mouse Click | on **Discard** button raise DCS_Discard Toolbox event, then a DCS_DialogueCompleted Toolbox event* |
| | on **Cancel** button raise DCS_Cancel Toolbox event, then a DCS_DialogueCompleted Toolbox event* |
| | on **Save** button raise DCS_Save Toolbox event, then a DCS_DialogueCompleted Toolbox event* |
| Key Pressed | on Return raise DCS_Save Toolbox event, then a DCS_DialogueCompleted Toolbox event* |
| | on Escape then act as if Cancel had been clicked. |

* if enabled

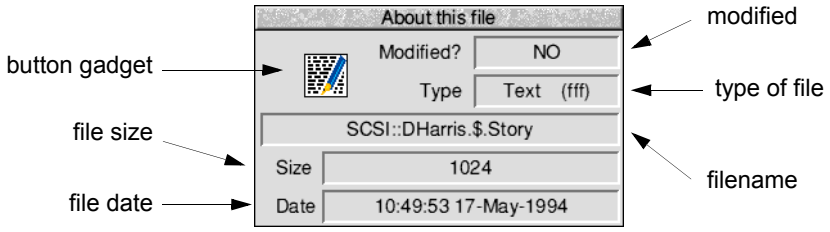
Note that if opened transiently, DCS_DialogueCompleted may be raised without any of DCS_Cancel, DCS_Discard or DCS_Save being raised. This could arise from the user clicking on the backdrop or opening a menu.

# 6 File Info Dialogue box class

A File Info dialogue object is used to display information about a file (or a directory or application) in a dialogue box.

## User interface

A File Info dialogue has the following information held in its dialogue box:




- an indication of whether the file is modified (a textual display field with the text ‘YES’ or ‘NO’)
- a sprite representing the file type (i.e. a sprite named file_xxx where xxx is the hex representation of the file type). If the filetype is 0x1000 a directory sprite is used, and if 0x2000 an application sprite is used.
- the type of the file (a textual display field with the textual filetype followed by its hex value in brackets)
- the full pathname of the file or ‘<untitled>’ (a display field)
- the size of the file in bytes (a display field giving the size of the file)
- the date the file was last written to (a textual display field showing the date in ‘*time’ format).

# Application Program Interface

## Attributes

A File Info object has the following attributes which are specified in its object template and can be manipulated at run-time by the client application:

| Attributes | Description | |
|---|---|---|
| flags | **Bit** | **Meaning** |
| | 0 | when set, this bit indicates that a FileInfo_AboutToBeShown event should be raised when SWI Toolbox_ShowObject is called for this object. |
| | 1 | when set, this bit indicates that a FileInfo_DialogueCompleted event should be raised when the File Info object has been removed from the screen. |
| File Info title | alternative title to use instead of 'About this file' (0 means use default title) | |
| max title length | this gives the maximum length in bytes of title text which will be used for this object | |
| modified | an indication as to whether the file is to be marked as modified from creation | |
| filetype | a word giving the RISC OS filetype | |
| filename | the initial filename to use in the dialogue box (if this field is 0, then the string '<untitled>' is used | |
| filesize | size of the file in bytes | |
| date | a 5-byte UTC time | |
| window | the name of an alternative window template to use instead of the default one (0 means use default) | |

## Manipulating a File Info object

### Creating and deleting a File Info object

A File Info object is created using SWI Toolbox_CreateObject.

When this object is created it has no attached objects (see page 11).

A File Info object is deleted using SWI Toolbox_DeleteObject.

The setting of the non-recursive delete bit does not have a meaning for File Info objects.

### Showing a File Info object

When a File Info object is displayed on the screen using SWI Toolbox_ShowObject it has the following behaviour:

| Show type | Position | |
|---|---|---|
| 0 (default) | the underlying window is shown at the last place shown on the screen, or the coordinates given in its template, if it has not already been shown | |
| 1 (full spec) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |
| | R3 + 8 | visible area maximum x coordinate |
| | R3 + 12 | visible area maximum y coordinate |
| | R3 + 16 | scroll x offset relative to work area |
| | R3 + 20 | scroll y offset relative to work area |
| | R3 + 24 | Wimp window handle of window to open behind<br>-1 means top of stack<br>-2 means bottom of stack<br>-3 means the window behind the Wimp's backwindow |
| 2 (topleft) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |

### Before the File Info dialogue box is shown

When SWI Toolbox_ShowObject is called, a FileInfo_AboutToBeShown Toolbox event is raised, if the appropriate bit is set in the File Info dialogue object's flags word. This enables the client to set any of the dialogue box's fields before it is displayed.

### Setting and reading the fields of the File Info dialogue

All of the display fields in a File Info dialogue can be set and read dynamically at run-time. The sprite displayed in the dialogue box depends on the value of the filetype field.

The methods used to do this are:

FileInfo_SetModifiedFileInfo_GetModified
FileInfo_SetFileTypeFileInfo_GetFileType
FileInfo_SetFileNameFileInfo_GetFileName
FileInfo_SetFileSizeFileInfo_GetFileSize
FileInfo_SetDateFileInfo_GetDate

## File Info methods

The following methods are all invoked by calling SWI Toolbox_ObjectMiscOp with:

| | |
|---|---|
| R0 | holding a flags word |
| R1 | being a File Info Dialogue object id |
| R2 | being the method code which distinguishes this method |
| R3-R9 | potentially holding method-specific data |

### FileInfo_GetWindowID 0

**On entry**

R0 = flags
R1 = File Info object id
R2 = 0

**On exit**

R0 = window object id for this File Info object

**Use**

This method returns the id of the underlying window object used to implement this File Info object.

**C veneer**

```
extern _kernel_oserror *fileinfo_get_window_id ( unsigned int flags,
                                                 ObjectId fileinfo,
                                                 ObjectId *window
                                               );
```

## FileInfo_SetModified 1

**On entry**

R0 = flags
R1 = File Info object id
R2 = 1
R3 = value

**On exit**

R1-R9 preserved

**Use**

This method sets whether the file is to be indicated as modified or not. If the value passed in R3 is 0, this indicates that the file is not modified; any other value in R3 means the file is modified.

**C veneer**

```
extern _kernel_oserror *fileinfo_set_modified ( unsigned int flags,
                                                ObjectId fileinfo,
                                                int modified
                                              );
```

## FileInfo_GetModified 2

**On entry**

R0 = flags
R1 = File Info object id
R2 = 2

**On exit**

R0 = modified state (0 ⇒ unmodified, non-0 ⇒ modified)

**Use**

This method returns whether the file is indicated as modified or not.

**C veneer**

```
extern _kernel_oserror *fileinfo_get_modified ( unsigned int flags,
                                                ObjectId fileinfo,
                                                int *modified
                                              );
```

## FileInfo_SetFileType 3

**On entry**

R0 = flags
R1 = File Info object id
R2 = 3
R3 = file type

**On exit**

R1-R9 preserved

**Use**

This method sets the file type to be indicated in the dialogue box.

**C veneer**

```
extern _kernel_oserror *fileinfo_set_file_type ( unsigned int flags,
                                                 ObjectId fileinfo,
                                                 int file_type
                                               );
```

## FileInfo_GetFileType 4

**On entry**

R0 = flags
R1 = File Info object id
R2 = 4

**On exit**

R0 = file type

**Use**

This method returns the file type shown in the dialogue box.

**C veneer**

```
extern _kernel_oserror *fileinfo_get_file_type ( unsigned int flags,
                                                 ObjectId fileinfo,
                                                 int *file_type
                                               );
```

## FileInfo_SetFileName 5

**On entry**

R0 = flags
R1 = File Info object id
R2 = 5
R3 = pointer to buffer holding filename

**On exit**

R1-R9 preserved

**Use**

This method sets the filename used in the File Info dialogue's Window. There is a limit of 256 characters on the filename length.

**C veneer**

```
extern _kernel_oserror *fileinfo_set_file_name ( unsigned int flags,
                                                 ObjectId fileinfo,
                                                 char *file_name
                                               );
```

## FileInfo_GetFileName 6

**On entry**

R0 = flags
R1 = File Info object id
R2 = 6
R3 = pointer to buffer to hold filename
R4 = size of buffer to hold filename

**On exit**

R4 = size of buffer required to hold filename (if R3 was 0)
else buffer pointed at by R3 holds filename
R4 holds number of bytes written to buffer

**Use**

This method returns the current filename used in a File Info object.

**C veneer**

```
extern _kernel_oserror *fileinfo_get_file_name ( unsigned int flags,
                                                 ObjectId fileinfo,
                                                 char *buffer,
                                                 int buff_size,
                                                 int *nbytes
                                               );
```

## FileInfo_SetFileSize 7

**On entry**

R0 = flags
R1 = File Info object id
R2 = 7
R3 = file size

**On exit**

R1-R9 preserved

**Use**

This method sets the file size to be indicated in the dialogue box.

**C veneer**

```
extern _kernel_oserror *fileinfo_set_file_size ( unsigned int flags,
                                                 ObjectId fileinfo,
                                                 int file_size
                                               );
```

## FileInfo_GetFileSize 8

**On entry**

R0 = flags
R1 = File Info object id
R2 = 8

**On exit**

R0 = file size

**Use**

This method returns the file size shown in the dialogue box.

**C veneer**

```
extern _kernel_oserror *fileinfo_get_file_size ( unsigned int flags,
                                                 ObjectId fileinfo,
                                                 int *file_size
                                               );
```

## FileInfo_SetDate 9

**On entry**

R0 = flags
R1 = File Info object id
R2 = 9
R3 = pointer to 5-byte UTC time

**On exit**

R1-R9 preserved

**Use**

This method sets the date string used in the File Info dialogue's window. The Territory Manager is used to convert the UTC time into a time string.

**C veneer**

```
extern _kernel_oserror *fileinfo_set_date ( unsigned int flags,
                                            ObjectId fileinfo,
                                            int *UTC
                                          );
```

## FileInfo_GetDate 10

**On entry**

R0 = flags
R1 = File Info object id
R2 = 10
R3 = pointer to buffer to hold 5-byte UTC time

**On exit**

R1-R9 preserved

**Use**

This method returns the current UTC time used in a File Info object.

**C veneer**

```
extern _kernel_oserror *fileinfo_get_date ( unsigned int flags,
                                            ObjectId fileinfo,
                                            int *UTC
                                          );
```

## FileInfo_SetTitle 11

**On entry**

R0 = flags
R1 = File Info object id
R2 = 11
R3 = pointer to text string to use

**On exit**

R1-R9 preserved

**Use**

This method sets the text which is to be used in the title bar of the given File Info dialogue.

**C veneer**

```
extern _kernel_oserror *fileinfo_set_title ( unsigned int flags,
                                             ObjectId fileinfo,
                                             char *title
                                           );
```

## FileInfo_GetTitle 12

**On entry**

R0 = flags
R1 = File Info object id
R2 = 12
R3 = pointer to buffer to return the text in (or 0)
R4 = size of buffer

**On exit**

R4 = size of buffer required to hold the text (if R3 was 0)
else Buffer pointed to by R3 contains title text
R4 holds number of bytes written to buffer

**Use**

This method returns the text string used in a File Info dialogue's title bar.

**C veneer**

```
extern _kernel_oserror *fileinfo_get_title ( unsigned int flags,
                                             ObjectId fileinfo,
                                             char *buffer,
                                             int buff_size,
                                             int *nbytes
                                           );
```

## File Info events

The File Info module generates the following Toolbox events:

### FileInfo_AboutToBeShown (0x82ac0)

**Block**

| | |
|---|---|
| + 8 | 0x82ac0 |
| + 12 | flags (as passed in to Toolbox_ShowObject |
| + 16 | value which will be passed in R2 to ToolBox_ShowObject |
| + 20... | block which will be passed in R3 to ToolBox_ShowObject for the underlying dialogue box |

**Use**

This Toolbox event is raised just before the File Info module is going to show its underlying Window object.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                show_type;
  union
  {
    TopLeft               pos;
    WindowShowObjectBlock full;
  } info;
} FileInfoAboutToBeShownEvent;
```

### FileInfo_DialogueCompleted (0x82ac1)

**Block**

| | |
|---|---|
| + 8 | 0x82ac1 |
| + 12 | flags<br>(none yet defined) |

**Use**

This Toolbox event is raised after the File Info object has been hidden, either by the user clicking outside the dialogue box or pressing Escape. It allows the client to tidy up its own state associated with this dialogue.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
} FileInfoDialogueCompletedEvent;
```

## File Info templates

The layout of a File Info template is shown below. Fields which have types MsgReference and StringReference are those which will require relocation when they are loaded from a resource file. If the template is being constructed in memory, then these fields should be real pointers (i.e. they do not require relocation).

For more details on relocation, see appendix *Resource File Formats* on page 499.

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| title | 4 | MsgReference |
| modified | 4 | word |
| filetype | 4 | word |
| filename | 4 | MsgReference |
| filesize | 4 | word |
| date | 8 | 2 words |
| window | 4 | StringReference |

### Underlying window template

The window object used to implement a File Info dialogue has the following characteristics. These must be reproduced if the Window is replaced by a client-specified alternative Window template:

Title bar must be indirected.

#### Gadgets

Component ids are derived by adding to 0x82ac00.

| Component id | Details |
|---|---|
| 0 | Display Field (date) |
| 1 | Display Field (size in bytes) |
| 2 | Display Field (filename) |
| 3 | Display Field (filetype) |
| 4 | Display Field (modified field) |
| 5 | Button gadget (indirected sprite used to display icon for file type) |
| 6 | Label (date) |
| 7 | Label (size) |
| 8 | Label (modified) |
| 9 | Label (type) |

## File Info Wimp event handling

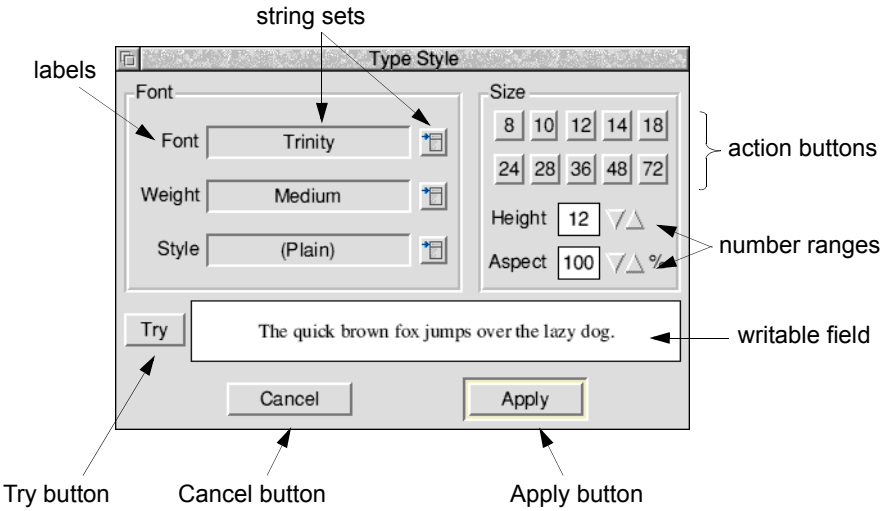
| Wimp event | Action |
|---|---|
| Open Window | Request show the dialogue box |
| Key Click | if Escape, then cancel this dialogue. |
| User Message | Window_HasBeenHidden<br>hide the dialogue box |

# 7 Font Dialogue box class

A Font Dialogue box shows font, weight and style of the currently selected font, together with a chosen height and aspect ratio. The dialogue box also has a writable field in which a test string in the chosen font is displayed.

## User interface

The Font Dialogue box can be broken down into the following components:




- A boxed area for setting the font, which contains three labels giving the font's name, weight and style; with three accompanying string sets (each string set contains a display field and a pop-up menu, which gives viable values for these fields, based on the list of currently available fonts). The pop-up menus are built and processed by the Toolbox, and do not require (or allow) any client intervention. The Toolbox deals with ensuring that only valid font id's are available to be chosen.
- Another boxed area, in which the user can set the height and aspect ratio used to plot the selected font. There are a number of standard sizes which can be chosen by clicking action buttons, and a number range into which a non-standard size can be entered. The aspect ratio used is specified by the contents of another number range.

- At the bottom of the dialogue box, there is a writable field which by default contains the string, ‘The quick brown fox jumps over the lazy dog’. When the user clicks on the **Try** button, this string is rendered in the selected font (and height and aspect ratio). The try string is limited to 64 characters long.
- The user can cancel the dialogue by clicking on the **Cancel** action button, or can apply the font selection by clicking on **Apply**.

Note that the strings which appear in the font, weight and style display fields may be localised for the current territory, but the strings used to communicate font selections between the client and the Toolbox are always the ‘real’ font id of the font (e.g. Corpus.Bold.Oblique).

## Application Program Interface

### Attributes

A Font Dialogue object has the following attributes which are specified in its object template and can be manipulated at run-time by the client application:

| **Attributes** | **Description** | |
|---|---|---|
| flags word | **Bit** | **Meaning** |
| | 0 | when set, this bit indicates that a FontDbox_AboutToBeShown event should be raised when SWI Toolbox_ShowObject is called for this object. |
| | 1 | when set, this bit indicates that a FontDbox_DialogueCompleted event should be raised when the Font Dialogue object has been removed from the screen. |
| | 2 | when set, include a **System font** entry in the list of fonts. |
| title | an alternative title for the dialogue box instead of ‘Type style’ (0 means use default title) | |
| max title length | the maximum length in bytes of title text which will be used for this object | |
| initial font | the font id to be displayed in the dialogue box as the selected font, on creation. If 0, the default is to display the first font in the list of currently available fonts. | |
| initial height | the initial height value when the dialogue box is created | |
| initial aspect | the initial aspect ratio value when the dialogue box is created | |

| Attributes | Description |
|---|---|
| try string | an alternative string to use in the **Try** writable field, instead of 'The quick brown fox jumps over the lazy dog' |
| window | an alternative window template to use instead of the default one. |

## Manipulating a Font Dialogue object

### Creating and deleting a Font Dialogue object

A Font Dialogue object is created using SWI Toolbox_CreateObject.

When this object is created it has no attached objects (see page 11).

A Font Dialogue object is deleted using SWI Toolbox_DeleteObject.

The setting of the non-recursive delete bit does not have a meaning for Font Dialogue objects.

### Showing a Font Dialogue object

When a Font Dialogue object is displayed on the screen using SWI Toolbox_ShowObject it has the following behaviour:

| Show type | Position | |
|---|---|---|
| 0 (default) | the underlying window is shown at the last place shown on the screen, or the coordinates given in its template, if it has not already been shown | |
| 1 (full spec) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |
| | R3 + 8 | visible area maximum x coordinate |
| | R3 + 12 | visible area maximum y coordinate |
| | R3 + 16 | scroll x offset relative to work area |
| | R3 + 20 | scroll y offset relative to work area |
| | R3 + 24 | Wimp window handle of window to open behind<br>-1 means top of stack<br>-2 means bottom of stack<br>-3 means the window behind the Wimp's backwindow |
| 2 (topleft) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |

**Before the Font Dialogue box is shown**

When the client calls Toolbox_ShowObject, a FontDbox_AboutToBeShown Toolbox event is raised (if the appropriate flags bit is set), allowing the client to take any last minute action. Typically, a client will indicate which of the fonts should be shown as the currently selected one, when it receives this event.

**Setting and getting the current selection**

The currently selected font id can be set and read at run-time using the FontDbox_SetFont and FontDbox_GetFont methods. These use a font id which assumes a <name>.<weight>.<style> structure (i.e. the first component appears in the **Font** field, the second in the **Weight** field, and the third in the **Style** field).

The size (both height and aspect ratio components) are set and read using the FontDbox_SetSize/FontDbox_GetSize methods respectively.

The **Try** string can be set and read using the FontDbox_SetTryString and FontDbox_GetTryString methods.

**Receiving a font selection**

When the user clicks the **Apply** button (or presses the Return key when the Font Dialogue box has the input focus), the client application is sent a FontDbox_ApplyFont Toolbox event. This event gives the font id of the currently selected font.

**Completing a Font Dialogue**

When the dialogue box is closed, either because **Apply** or **Cancel** has been clicked, or Escape has been pressed, a FontDbox_DialogueCompleted Toolbox event is raised for the client, with an indication of whether a font was selected during the dialogue.

## Font Dialogue methods

The following methods are all invoked by calling SWI Toolbox_MiscOp with:

| | |
|---|---|
| R0 | holding a flags word |
| R1 | being a Font Dialogue Box id |
| R2 | being the method code which distinguishes this method |
| R3-R9 | potentially holding method-specific data |

### FontDbox_GetWindowID 0

**On entry**

R0 = flags
R1 = FontDbox object id
R2 = 0

**On exit**

R0 = Window object id for this FontDbox object

**Use**

This method returns the id of the underlying Window object used to implement this FontDbox object.

**C veneer**

```
extern _kernel_oserror *fontdbox_get_window_id( unsigned int flags,
                                                ObjectId fontdbox,
                                                ObjectId *window
                                              );
```

## FontDbox_SetFont 1

**On entry**

R0 = flags
R1 = Font Dbox object id
R2 = 1
R3 = pointer to font id of font to select (0 means none)

**On exit**

R1-R9 preserved

**Use**

This method selects a font as being the currently selected one for this Font Dialogue box, and displays its name appropriately in the **Font**/**Weight**/**Style** display fields.

The special font id ‘SystemFont’ is used to indicate that the **System** entry should be selected.

**C veneer**

```
extern _kernel_oserror *fontdbox_set_font ( unsigned int flags,
                                            ObjectId fontdbox,
                                            char *font_id
                                          );
```

## FontDbox_GetFont 2

**On entry**

R0 = flags
R1 = Font Dbox object id
R2 = 2
R3 = pointer to buffer to hold font id
R4 = buffer size for font id

**On exit**

R4 = size of buffer required (if R3 was 0)
else buffer pointed at by R3 holds font id
R4 holds number of bytes written to buffer

**Use**

This method returns the font id for the font which was last specified in a FontDbox_SetFont call, or was last chosen by a user choice from a pop-up menu.

The special font id 'SystemFont' is used to indicate that the **System** entry is selected.

**C veneer**

```
extern _kernel_oserror *fontdbox_get_font ( unsigned int flags,
                                            ObjectId fontdbox,
                                            char *buffer,
                                            int buff_size,
                                            int *nbytes
                                          );
```

## FontDbox_SetSize 3

**On entry**

R0 = flags
    bit 0 set means change the height value
    bit 1 set means change the aspect ratio
R1 = Font Dbox object id
R2 = 3
R3 = height value
R4 = aspect ratio value

**On exit**

R1-R9 preserved

**Use**

This method sets the height value and/or the aspect ratio displayed in the Font Dialogue box.

**C veneer**

```
extern _kernel_oserror *fontdbox_set_size ( unsigned int flags,
                                            ObjectId fontdbox,
                                            int height,
                                            int aspect_ratio
                                          );
```

## FontDbox_GetSize 4

**On entry**

R0 = flags
R1 = Font Dbox object id
R2 = 4

**On exit**

R0 = height value
R1 = aspect ratio

**Use**

This method returns the height value and/or aspect ratio currently displayed in the Font Dialogue box.

**C veneer**

```
extern _kernel_oserror *fontdbox_get_size ( unsigned int flags,
                                            ObjectId fontdbox,
                                            int *height,
                                            int *aspect_ratio
                                          );
```

## FontDbox_SetTryString 5

**On entry**

R0 = flags
R1 = Font Dbox object id
R2 = 5
R3 = pointer to ‘try’ string to use

**On exit**

R1-R9 preserved

**Use**

This method sets the string used in the **Try** writable field of a Font Dialogue box. If the string is longer than 64 characters, an error is returned.

**C veneer**

```
extern _kernel_oserror *fontdbox_set_try_string ( unsigned int flags,
                                                  ObjectId fontdbox,
                                                  char *try_string
                                                );
```

## FontDbox_GetTryString 6

**On entry**

R0 = flags
R1 = Font Dbox object id
R2 = 6
R3 = pointer to buffer to hold try string
R4 = buffer size for try string

**On exit**

R4 = size of buffer required (if R3 was 0)
else buffer pointed at by R3 holds try string
R4 holds number of bytes written to buffer

**Use**

This method returns the string currently displayed in the **Try** writable field of the Font Dialogue box.

**C veneer**

```
extern _kernel_oserror *fontdbox_get_try_string ( unsigned int flags,
                                                  ObjectId fontdbox,
                                                  char *buffer,
                                                  int buff_size,
                                                  int *nbytes
                                                );
```

## FontDbox_SetTitle 7

**On entry**

R0 = flags
R1 = Font Dbox object id
R2 = 7
R3 = pointer to text string to use

**On exit**

R1-R9 preserved

**Use**

This method sets the text which is to be used in the title bar of the given Font dialogue box.

**C veneer**

```
extern _kernel_oserror *fontdbox_set_title ( unsigned int flags,
                                             ObjectId fontdbox,
                                             char *title
                                           );
```

## FontDbox_GetTitle 8

**On entry**

R0 = flags
R1 = Font Dbox object id
R2 = 8
R3 = pointer to buffer to return the text in (or 0)
R4 = size of buffer

**On exit**

R4 = the size of buffer required to hold the text (if R3 was 0)
else Buffer pointed to by R3 contains title text
R4 holds number of bytes written to buffer

**Use**

This method returns the text string used in a Font dialogue's title bar.

**C veneer**

```
extern _kernel_oserror *fontdbox_get_title ( unsigned int flags,
                                             ObjectId fontdbox,
                                             char *buffer,
                                             int buff_size,
                                             int *nbytes
                                           );
```

## Font Dialogue events

There are a number of Toolbox events which are generated by the Font Dialogue box module.

### FontDbox_AboutToBeShown (0x82a00)

**Block**

| | |
|---|---|
| + 8 | 0x82a00 |
| + 12 | flags (as passed in to Toolbox_ShowObject) |
| + 16 | value which will be passed in R2 to ToolBox_ShowObject |
| + 20... | block which will be passed in R3 to ToolBox_ShowObject for the underlying dialogue box |

**Use**

This Toolbox Event is raised when SWI Toolbox_ShowObject has been called for a Font Dialogue Box object. It gives the application the opportunity to set the selected font before the dialogue box actually appears on the screen.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                show_type;
  union
  {
    TopLeft               pos;
    WindowShowObjectBlock full;
  } info;
} FontDboxAboutToBeShownEvent;
```

## FontDbox_DialogueCompleted (0x82a01)

**Block**

+ 8 0x82a01
+ 12 flags

**Use**

This Toolbox Event is raised after the Font Dialogue object has been hidden, either by a Cancel click, or by a click on **Apply**. It allows the client to tidy up its own state associated with this dialogue.

Note that if the dialogue was cancelled, a font selection may still have been made, for example if the user clicked Adjust on **Apply**, and then cancelled the dialogue.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
} FontDboxDialogueCompletedEvent;
```

## FontDbox_ApplyFont (0x82a02)

**Block**

+ 8 0x82a02
+ 16 font height
+ 20 aspect ratio
+ 24... font id

**Use**

This Toolbox Event informs the client that a Font Dialogue box selection has been made.

The special font id SystemFont is used to indicate that the **System** entry is selected.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  unsigned int       height;
  unsigned int       aspect;
  char               font[208];
} FontDboxApplyFontEvent;
```

## Font Dialogue Templates

The layout of a Font Dialogue box template is shown below. Fields which have types MsgReference and StringReference are those which will require relocation when they are loaded from a resource file. If the template is being constructed in memory, then these fields should be real pointers (i.e. they do not require relocation).

For more details on relocation, see appendix *Resource File Formats* on page 499.

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| title | 4 | MsgReference |
| max_title | 4 | word |
| initial_font | 4 | StringReference |
| initial_height | 4 | word |
| initial_aspect | 4 | word |
| try_string | 4 | MsgReference |
| window | 4 | StringReference |

### Underlying Window template

The Window object used to implement a Font Dialogue has the following characteristics. These must be reproduced if the Window is replaced by a client-specified alternative Window template:

Title bar must be indirected.

**Gadgets**

Component ids are derived by adding to 0x82a000

| Component id | Details | |
|---|---|---|
| 0 | action button (**Apply**) | must be marked as the ‘default’ action button |
| 1 | action button (**Cancel**) | must be marked as the ‘cancel’ action button |
| 2 | action button (**Try**) | must be marked as a ‘local’ action button |
| 3 | writable field (**Try** string) | buffer must be 64 bytes |
| 4 | number range (**Aspect** ratio) | |
| 5 | number range (**Height**) | |

| Component id | Details | |
|---|---|---|
| 6-15 | action buttons (Standard sizes) | these should all be local action buttons containing the text 8, 10 12, 14, 18, 24, 28, 36, 48 72 respectively. |
| 16 | string set (**Style**) | non-writable, with pop-up menu |
| 17 | string set (**Weight**) | non-writable, with pop-up menu |
| 18 | string set (**Font**) | non-writable, with pop-up menu |
| 19 | label box (**Font**) | |
| 20 | label box (**Style**) | |
| 21 | label (**Height**) | |
| 22 | label (**Aspect**) | |
| 23 | label (**%**) | |
| 24 | label (**Font**) | |
| 25 | label (**Weight**) | |
| 26 | label (**Style**) | |

## Font Dialogue Wimp event handling

The Font Dialogue box class responds to certain Wimp events and takes the actions as described below:

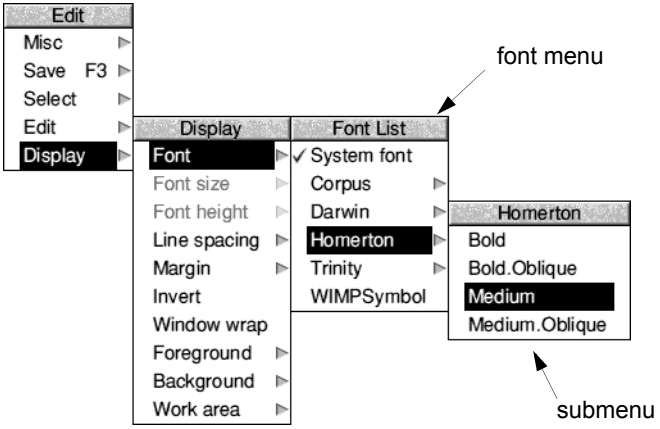
| Wimp event | Action |
|---|---|
| Mouse Click | on **Apply**, deliver a FontDbox_ApplyFont event |
| | on **Cancel**, deliver a FontDbox_DialogueCompleted event |
| | on one of the pop-up menu buttons, a menu is displayed |
| | on one of the ‘standard sizes’, this size is entered into the **Height** writable field |
| | on one of the arrow keys, increment/decrement the value of its associated writable field (either height or aspect ratio) |
| Key Pressed | if Return then act as if **Apply** button had been clicked |
| | if Escape, then act as if **Cancel** button had been clicked |

# 8 Font Menu class

A Font Menu is a menu which shows the currently selected font, and allows the user to set this from a list of font names, and submenus which give styles and weights.

## User interface

A typical Font Menu might look as follows:




When a hit is received for the Font Menu, it is decoded by the Font Menu module, and a Toolbox event is returned to the client. This contains the font id of the selected font (see SWI Font_DecodeMenu). The chosen font is shown as ticked in the font menu when the menu is next shown (or immediately if Adjust is held down).

## Application Program Interface

The RISC OS Font manager provides a facility of building a font menu from the current fontlist.

A Font Menu object is an abstraction on this facility. A Font Menu is built for the client using the Font manager.

### Attributes

A Font Menu object has the following attributes which are specified in its object template and can be manipulated at run-time by the client application:

| Attributes | Description | |
|---|---|---|
| flags word | **Bit** | **Meaning** |
| | 0 | when set, this bit indicates that a FontMenu_AboutToBeShown event should be raised when SWI Toolbox_ShowObject is called for this object |
| | 1 | when set, this bit indicates that a FontMenu_HasBeenHidden event should be raised when the Font Menu object has been removed from the screen |
| | 2 | when set, include a **System font** entry at head of menu |
| ticked_font | font id of the font to tick in the Font Menu when it is first created<br>The special font id 'SystemFont' is used to indicate that the **System** entry should be ticked. | |

### Manipulating a Font Menu object

**Creating and deleting a Font Menu object**

A Font Menu object is created using SWI Toolbox_CreateObject.

When this object is created it has no attached objects (see page 11).

A Font Menu object is deleted using SWI Toolbox_DeleteObject.

The setting of the non-recursive delete bit does not have a meaning for Font Menu objects.

### Showing a Font Menu object

When a Font Menu object is displayed on the screen using SWI Toolbox_ShowObject it has the following behaviour:

| Show type | Position |
|---|---|
| 0 (default) | 64 OS units to the left of the mouse pointer |
| 1 (full spec) | R3 + 0 gives x coordinate of top-left corner of Menu<br>R3 + 4 gives y coordinate of top-left corner of Menu |
| 2 (topleft) | R3 + 0 gives x coordinate of top-left corner of Menu<br>R3 + 4 gives y coordinate of top-left corner of Menu |

### Before the Font Menu is shown

When the client calls Toolbox_ShowObject, a FontMenu_AboutToBeShown Toolbox event is raised (if the appropriate flags bit is set), allowing the client to take any last minute action. Typically, a client will indicate which of the fonts should be shown as the currently selected one, when it receives this event.

### Selecting a font

The currently selected font is shown ticked in the Font Menu. The selected font can be set using FontMenu_SetFont, and can be read using FontMenu_GetFont. Note that the string passed to these methods is the font id, not the translated string.

### Receiving a font selection

When the user makes a Font selection from the Font Menu, a FontMenu_FontSelection Toolbox event is raised. This gives the font id of the font which has been chosen from the Font Menu.

## Font Menu methods

The following methods are all invoked by calling SWI Toolbox_MiscOp with:

| | |
|---|---|
| R0 | holding a flags word |
| R1 | being a Font Menu id |
| R2 | being the method code which distinguishes this method |
| R3-R9 | potentially holding method-specific data |

### FontMenu_SetFont 0

**On entry**

R0 = flags
R1 = Font Menu object id
R2 = 0
R3 = pointer to font id of font to select (0 means none)

**On exit**

R1-R9 preserved

**Use**

This method selects a font as being the currently selected one for this Font Menu, and places a tick next to it. The special font id ‘SystemFont’ is used to indicate that the **System** entry should be ticked.

**C veneer**

```
extern _kernel_oserror *fontmenu_set_font ( unsigned int flags,
                                            ObjectId fontmenu,
                                            char *font_id
                                          );
```

## FontMenu_GetFont 1

**On entry**

R0 = flags
R1 = Font Menu object id
R2 = 1
R3 = pointer to buffer to hold font id
R4 = buffer size for font id

**On exit**

R4 = size of buffer required (if R3 was 0)
else buffer pointed at by R3 holds font id
R4 holds number of bytes written to buffer

**Use**

This method returns the font id for the font which was last specified in a FontMenu_SetFont call, or was last chosen by a user mouse click (i.e. the one which is ticked). The special font id 'SystemFont' is used to indicate that the **System** entry was last chosen.

**C veneer**

```
extern _kernel_oserror *fontmenu_get_font ( unsigned int flags,
                                            ObjectId fontmenu,
                                            char *buffer,
                                            int buff_size,
                                            int *nbytes
                                          );
```

## Font Menu events

There are a number of Toolbox events which are generated by the Font Menu module:

### FontMenu_AboutToBeShown (0x82a40)

**Block**

+ 8 0x82a40
+ 12 flags (as passed in to Toolbox_ShowObject)
+ 16 value which will be passed in R2 to ToolBox_ShowObject
+ 20... block which will be passed in R3 to ToolBox_ShowObject for the underlying Menu Object

**Use**

This Toolbox event is raised when SWI Toolbox_ShowObject has been called for a Font Menu object. It gives the application the opportunity to set the selected font before the Menu actually appears on the screen.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                show_type;
  TopLeft            pos;
} FontMenuAboutToBeShownEvent;
```

### FontMenu_HasBeenHidden (0x82a41)

**Block**

+ 8 0x82a41

**Use**

This Toolbox Event is raised by the Toolbox when Toolbox_HideObject is called on a Font Menu which has the appropriate bit set in its template flags word. It enables a client application to clear up after a menu has been closed. It is also raised when clicking outside a menu or hitting Escape.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
} FontMenuHasBeenHiddenEvent;
```

### FontMenu_FontSelection (0x82a42)

**Block**

+ 8 0x82a42
+ 16... font id

**Use**

This Toolbox Event informs the client that a Font Menu selection has been made.

The special font id ‘SystemFont’ is used to indicate that the **System** entry was last chosen.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  char               font_id[216];
} FontMenuSelectionEvent;
```

## Font Menu templates

The layout of a Font Menu template is shown below. Fields which have types MsgReference and StringReference are those which will require relocation when they are loaded from a resource file. If the template is being constructed in memory, then these fields should be real pointers (i.e. they do not require relocation).

For more details on relocation, see appendix *Resource File Formats* on page 499.

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| ticked_font | 4 | StringReference |

## Font Menu Wimp event handling

The Font Menu class responds to certain Wimp events and takes the actions as described below:

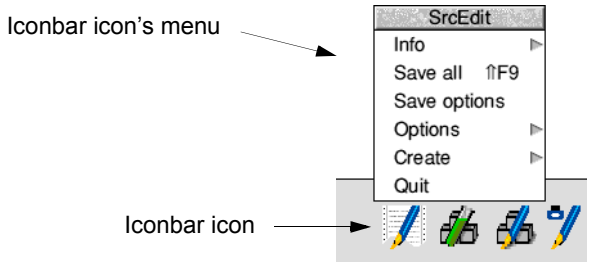
| Wimp event | Action |
|---|---|
| Menu Selection | The font id corresponding to the menu selection is sent back to the client via a FontMenu_FontSelection event.<br>If Adjust is held down, then the currently open Menu is re-opened in the same place. |
| User Msg | Message_HelpRequest (while the pointer is over a Font Menu object) A reply is sent on the application's behalf. |

# 9 Iconbar icon class

Objects of the Iconbar icon class are used to display an application icon on the Iconbar.

## User interface

An Iconbar object is normally used to show that an application is running, by placing an icon on the RISC OS Iconbar.




An Iconbar object can either be a sprite icon or a text&sprite icon. It does not appear on the Iconbar until the application has called Toolbox_ShowObject or if the auto-show bit has been set in its flags word. When the Toolbox places the icon on the Iconbar, it positions the icon in a Style Guide compliant manner, including placement of the text in a text&sprite icon. The bounding box used for the icon is taken from the sprite used for that icon, also taking into consideration the text used, if the iconbar object is text&sprite. If the application supports many icons on the Iconbar this can be achieved by creating many Iconbar objects.

The Toolbox supports handling of a Menu click over the icon, Select and Adjust clicks.

# Application Program Interface

## Attributes

An Iconbar icon object has the following attributes which are specified in its object template and can be manipulated at run-time by the client application:

| Attributes | Description | |
|---|---|---|
| flags | **Bit** | **Meaning** |
| | 0 | when set, generate an Iconbar_SelectAboutToBeShown event before the object which has been associated with a Select click is shown |
| | 1 | when set, generate an Iconbar_AdjustAboutToBeShown event before the object which has been associated with an Adjust click is shown |
| | 2 | when set, show the select_show object as a transient (i.e. with the semantics of Wimp_CreateMenu) |
| | 3 | when set, show the adjust_show object as a transient (i.e. with the semantics of Wimp_CreateMenu) |
| | 4 | reserved |
| | 5 | when set, generate an Iconbar_Clicked (or client-specified) event when Select is clicked |
| | 6 | when set, generate an Iconbar_Clicked (or client-specified) event when Adjust is clicked |
| position | a negative integer giving the position of the icon on the Iconbar (as specified in SWI Wimp_CreateIcon) | |
| priority | gives priority of this icon on the Iconbar (as specified in SWI Wimp_CreateIcon) | |
| sprite name | the name of the sprite to use for this Iconbar icon | |
| max sprite name | the maximum length of sprite name to be used | |
| text | an optional string which will be used for a Text&Sprite Iconbar icon (i.e. the text that will appear underneath the icon on the Iconbar) | |
| max text length | if the Iconbar icon has text, then this is a Text&Sprite Iconbar icon, and this field gives the maximum length of a text string which will be used for it | |

| Attributes | Description |
|---|---|
| menu | the name of the template to use to create a Menu object for this Iconbar icon |
| select event | the Toolbox Event code to be raised when the user clicks Select on the Iconbar icon<br>(if 0 then Iconbar_Clicked is raised) |
| adjust event | the Toolbox event code to be raised when the user clicks Adjust on the Iconbar icon<br>(if 0 then Iconbar_Clicked is raised) |
| select show | the name of a template to use to show an object when the user clicks Select on the Iconbar icon |
| adjust show | the name of a template to use to show an object when the user clicks Adjust on the Iconbar icon |
| help message | the message to respond to a help request with, instead of the default |
| max help | the maximum length of help message to be used |

## Manipulating an Iconbar icon object

### Creating and deleting an Iconbar icon object

An Iconbar icon object is created using SWI Toolbox_CreateObject.

When an Iconbar Icon Object is created, the following attached objects (see page 11) will be created (if specified):

- menu
- select show
- adjust show.

See the attributes table above for an explanation of what these objects are.

An Iconbar object is deleted using SWI Toolbox_DeleteObject. If it has any attached objects (see above), these are also deleted, unless the non-recursive bit is set for this SWI.

### Showing an Iconbar icon object

When a Iconbar icon object is displayed on the screen using SWI Toolbox_ShowObject it has the following behaviour:

| Show type | Position | |
|---|---|---|
| 0 (default) | display on the Iconbar in a place specified by the object's template's position and priority fields. | |
| 1 (full spec) | R3 + 0 | icon handle of icon to show icon to the left (-3) or right (-4) of its position. |

If the Iconbar icon's position is any other value than -3 or -4, then R3 should just be 0.

An Iconbar icon is hidden by using SWI Toolbox_HideObject.

### The Iconbar icon's position and priority

An Iconbar icon is created with a position and a priority. These are integer values as specified in SWI Wimp_CreateIcon. Note that these values are fixed at create-time, but are only used when the Iconbar icon is 'shown', either by explicitly calling Toolbox_ShowObject, or by setting the auto-show bit in the object template's flags.

The semantics of position and priority are as documented in Wimp_CreateIcon. Applications will mostly just use a position of -1 for the right of the iconbar.

Note that positions of -3 and -4 cannot be used in conjunction with the auto-show bit. Such an Iconbar icon must be explicitly shown using Toolbox_ShowObject to allow the client to pass the Wimp handle of the icon to whose left/right this icon should be placed.

An Iconbar icon's position and priority cannot be changed at run-time.

### The Iconbar icon's menu

Each Iconbar object can optionally have attached to it a Menu object. The Iconbar object holds the object id of this Menu object.

Whenever the user of the application presses the Menu mouse button over an Iconbar icon, the Iconbar class module opens its attached Menu object, by making a SWI Toolbox_ShowObject passing the attached Menu's id.

If the application wishes to perform some operations on the Menu before it is opened (ticking some entries for example), then by setting the appropriate bit in the Menu's flags word, the application can request that a special Toolbox event (Menu_AboutToBeShown) is delivered to it before the Menu is actually shown. The precise details of this Toolbox event are described on page 197. On receipt of such a Toolbox event, the client application is expected to make any changes it wants to the Menu object, and then return to its SWI Wimp_Poll loop.

When an Iconbar icon is created, if the client has specified the name of a Menu template for that Iconbar icon, then a Menu object is created from that template, and the id of that Menu is held in the Iconbar object. This id will be used to show the Menu when the user presses the Menu button over the Iconbar icon.

In most cases a Menu is attached to the Iconbar icon at resource editing time by entering the name of the template to use for this Iconbar icon's Menu. If the application wishes to dynamically attach and detach the Menu for a given Iconbar icon, then this can be done using the Iconbar_SetMenu method described on page 151.

The id of the Menu attached to an Iconbar icon can be read by using the Iconbar_GetMenu method.

**Select and Adjust click events**

The client application can specify a Toolbox event to be raised when the user clicks Select and/or one to be raised when the user clicks Adjust on the Iconbar icon.

This event will only be raised if the appropriate flags bits have been set for Select and Adjust clicks.

Normally this is specified in the application's resource file, but it can be set and read using the Iconbar_SetEvent/Iconbar_GetEvent methods.

**Help messages**

Each Iconbar object can optionally have attached to it a Help Message.

Whenever the Wimp delivers a HelpRequest message to the client application for this Iconbar icon, the attached Help Message is sent back automatically by the Toolbox.

In most cases a help message is attached to the Iconbar object at resource editing time. An Iconbar icon's Help Message can be set dynamically using the Iconbar_SetHelpMessage method described on page 156.

The text of the Help Message can be read using the Iconbar_GetHelpMessage method.

## Iconbar icon methods

The following methods are all invoked by calling SWI Toolbox_ObjectMiscOp with:

| | |
|---|---|
| R0 | holding a flags word |
| R1 | being an Iconbar object id |
| R2 | being the method code which distinguishes this method |
| R3-R9 | potentially holding method-specific data |

### Iconbar_GetIconHandle 0

**On entry**

R0 = flags
R1 = Iconbar object id
R2 = 0

**On exit**

R0 = Wimp icon handle for this Iconbar object

**Use**

This method returns the handle of the underlying Wimp icon used to implement this Iconbar object.

**C veneer**

```
extern _kernel_oserror *iconbar_get_icon_handle ( unsigned int flags,
                                                  ObjectId iconbar,
                                                  int *icon_handle
                                                );
```

## Iconbar_SetMenu 1

**On entry**

R0 = flags
R1 = Iconbar object id
R2 = 1
R3 = menu id

**On exit**

R1-R9 preserved

**Use**

This method is used to set the menu which will be displayed when the Menu button is pressed over this Iconbar object. The Toolbox handles opening the menu for you.

If R3 is 0, then the menu for this Iconbar object is detached.

**C veneer**

```
extern _kernel_oserror *iconbar_set_menu ( unsigned int flags,
                                           ObjectId iconbar,
                                           ObjectId menu_id
                                         );
```

## Iconbar_GetMenu 2

**On entry**

R0 = flags
R1 = Iconbar object id
R2 = 2

**On exit**

R0 = Menu id

**Use**

This method is used to get the id of the menu which will be displayed when the Menu button is pressed over this Iconbar object.

**C veneer**

```
extern _kernel_oserror *iconbar_get_menu ( unsigned int flags,
                                           ObjectId iconbar,
                                           ObjectId *menu_id
                                         );
```

## Iconbar_SetEvent 3

**On entry**

R0 = flags
    bit 0 set means raise the event code specified in R3 when Select is clicked
    bit 1 set means raise the event code specified in R4 when Adjust is clicked
R1 = Iconbar object id
R2 = 3
R3 = Toolbox Event code to raise for Select
R4 = Toolbox Event code to raise for Adjust

**On exit**

R1-R9 preserved

**Use**

This method specifies a Toolbox event to be raised when the user clicks Select and/or Adjust on the Iconbar icon.

If R3 or R4 is 0, then an IconBar_Clicked Toolbox event will be raised instead.

**C veneer**

```
extern _kernel_oserror *iconbar_set_event ( unsigned int flags,
                                            ObjectId iconbar,
                                            int select_event,
                                            int adjust_event
                                          );
```

## Iconbar_GetEvent 4

**On entry**

R0 = flags
    bit 0 set means return the event code which will be raised
        when Select is clicked
    bit 1 set means return the event code which will be raised
        when Adjust is clicked
R1 = Iconbar object id
R2 = 4

**On exit**

R0 = Toolbox event code raised when Select is clicked on the Iconbar icon
R1 = Toolbox event code raised when Adjust is clicked on the Iconbar icon

**Use**

This method reads the Toolbox Event to be raised when the user clicks Select or Adjust on the Iconbar icon.

**C veneer**

```
extern _kernel_oserror *iconbar_get_event ( unsigned int flags,
                                            ObjectId iconbar,
                                            int *select_event,
                                            int *adjust_event
                                          );
```

## Iconbar_SetShow 5

**On entry**

R0 = flags
    bit 0 set means show the object whose id is given in R3
        when Select is clicked
    bit 1 set means show the object whose id is given in R4
        when Adjust is clicked
R1 = Iconbar object id
R2 = 5
R3 = id of object to show for Select
R4 = id of object to show for Adjust

**On exit**

R1-R9 preserved

**Use**

This method specifies an object to be shown when the user clicks Select and/or Adjust on the Iconbar icon.

If R3 or R4 is 0, then no object will be shown.

**C veneer**

```
extern _kernel_oserror *iconbar_set_show ( unsigned int flags,
                                           ObjectId iconbar,
                                           ObjectId select,
                                           ObjectId adjust
                                         );
```

## Iconbar_GetShow 6

**On entry**

R0 = flags
bit 0 set means return the id of the object which will be shown when Select is clicked
bit 1 set means return the id of the object which will be shown when Adjust is clicked
R1 = Iconbar object id
R2 = 6

**On exit**

R0 = id of object which will be shown when Select is clicked on the Iconbar icon. R1 = id of object which will be shown when Adjust is clicked on the Iconbar icon

**Use**

This method reads the ids of the objects to be shown when the user clicks Select or Adjust on the Iconbar icon.

**C veneer**

```
extern _kernel_oserror *iconbar_get_show ( unsigned int flags,
                                           ObjectId iconbar,
                                           ObjectId *select,
                                           ObjectId *adjust
                                         );
```

## Iconbar_SetHelpMessage 7

**On entry**

R0 = flags
R1 = Iconbar object id
R2 = 7
R3 = pointer to message text

**On exit**

R1-R9 preserved

**Use**

This method is used to set the help message which will be returned when a Help Request message is received for this Iconbar object. The Toolbox handles the reply message for you.

If R3 is 0, then the Help Message for this Iconbar object is detached.

**C veneer**

```
extern _kernel_oserror *iconbar_set_help_message ( unsigned int flags,
                                                   ObjectId iconbar,
                                                   char *message_text
                                                 );
```

## Iconbar_GetHelpMessage 8

**On entry**

R0 = flags
R1 = Iconbar object id
R2 = 8
R3 = pointer to buffer (or 0)
R4 = size of buffer to hold message text

**On exit**

R4 = holds size of buffer required for message text (if R3 was 0)
else Buffer pointed at by R3 holds message text
R4 holds number of bytes written to buffer

**Use**

This method is used to read the help message which will be returned when a Help Request message is received for this Iconbar object.

**C veneer**

```
extern _kernel_oserror *iconbar_get_help_message ( unsigned int flags,
                                                   ObjectId iconbar,
                                                   char *buffer,
                                                   int buff_size,
                                                   int *nbytes
                                                 );
```

## Iconbar_SetText 9

**On entry**

R0 = flags
R1 = Iconbar object id
R2 = 9
R3 = pointer to text string to use

**On exit**

R1-R9 preserved

**Use**

This method sets the text which is to be used in a text&sprite Iconbar object. If the text is longer than the maximum size specified when the Iconbar icon was created, then an error is returned.

**C veneer**

```
extern _kernel_oserror *iconbar_set_text ( unsigned int flags,
                                           ObjectId iconbar,
                                           char *text
                                         );
```

## Iconbar_GetText 10

**On entry**

R0 = flags
R1 = Iconbar object id
R2 = 10
R3 = pointer to buffer to return the text in (or 0)
R4 = size of buffer

**On exit**

R4 = the size of buffer required to hold the text (if R3 was 0)
else Buffer pointed to by R3 contains icon's text
R4 holds number of bytes written to buffer

**Use**

This method is used for a text&sprite Iconbar object. It returns the text string displayed for that object.

**C veneer**

```
extern _kernel_oserror *iconbar_get_text ( unsigned int flags,
                                           ObjectId iconbar,
                                           char *buffer,
                                           int buff_size,
                                           int *nbytes
                                         );
```

## Iconbar_SetSprite 11

**On entry**

R0 = flags
R1 = Iconbar object id
R2 = 11
R3 = pointer to name of sprite to use

**On exit**

R1-R9 preserved

**Use**

This method sets the sprite which is to be used in the Iconbar object.

**C veneer**

```
extern _kernel_oserror *iconbar_set_sprite ( unsigned int flags,
                                             ObjectId iconbar,
                                             char *sprite_name
                                           );
```

## Iconbar_GetSprite 12

**On entry**

R0 = flags
R1 = Iconbar object id
R2 = 12
R3 = pointer to buffer to return the sprite name in (or 0)
R4 = size of buffer

**On exit**

R4 = holds size of buffer required for sprite name (if R3 was 0)
else Buffer pointed at by R3 holds sprite name
R4 holds number of bytes written to buffer

**Use**

This method returns the name of the sprite used for the Iconbar object.

**C veneer**

```
extern _kernel_oserror *iconbar_get_sprite ( unsigned int flags,
                                             ObjectId iconbar,
                                             char *buffer,
                                             int buff_len,
                                             int *nbytes
                                           );
```

# Iconbar icon events

## Iconbar_Clicked (0x82900)

**Block**

+ 8 0x82900
+ 12 flags
bits 0, 1 and 2 show how the activation was done:
bit 0 set means Adjust was clicked
bit 1 reserved
bit 2 set means Select was clicked

**Use**

This Toolbox event is raised when the user clicks Select or Adjust on an Iconbar object, and the client application has not associated any other Toolbox event with this event.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
} IconbarClickedEvent;
```

## Iconbar_SelectAboutToBeShown (0x82901)

**Block**

+ 8 0x82901
+ 16 object id of the object which will be shown
(note that the 'self' field in the id block will be for the Iconbar object).

**Use**

This Toolbox event is raised just before Toolbox_ShowObject is called for the object to be shown on a Select click. Note that on receipt of this event, the client could call Iconbar_SetShow to give the object id of a different object to be shown.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  ObjectId           id;
} IconbarAboutToBeShownEvent;
```

### Iconbar_AdjustAboutToBeShown (0x82902)

**Block**

+ 8 0x82902
+ 16 object id of the object which will be shown
(note that the 'self' field in the id block will be for the Iconbar object).

**Use**

This Toolbox event is raised just before Toolbox_ShowObject is called for the object to be shown on a Adjust click. Note that on receipt of this event, the client could call Iconbar_SetShow to give the object id of a different object to be shown.

Note: This event and the Iconbar_SelectAboutToBeShown event both share the same typedef.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  ObjectId           id;
} IconbarAboutToBeShownEvent;
```

## Iconbar icon templates

The layout of an Iconbar icon template is shown below. Fields which have types MsgReference and StringReference are those which will require relocation when they are loaded from a resource file. If the template is being constructed in memory, then these fields should be real pointers (i.e. they do not require relocation).

For more details on relocation, see appendix *Resource File Formats* on page 499.

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| position | 4 | word |
| priority | 4 | word |
| sprite_name | 4 | StringReference |
| max_sprite_name | 4 | word |
| text | 4 | MsgReference |
| max_text_len | 4 | word |
| menu | 4 | StringReference |
| select_event | 4 | word |
| adjust_event | 4 | word |

| Field | Size in bytes | Type |
|---|---|---|
| select_show | 4 | StringReference |
| adjust_show | 4 | StringReference |
| help_message | 4 | MsgReference |
| max_help | 4 | word |

## Iconbar icon Wimp event handling

Certain Wimp events for an Iconbar icon are fielded by the Iconbar class, and either acted upon for the client, or result in a Toolbox event being raised. Such events are listed below:

| Wimp event | Action |
|---|---|
| Mouse Click | If the Menu button has been pressed, and there is a Menu object attached to this Iconbar icon, then the Menu is shown using Toolbox_ShowObject.<br>If the Select or Adjust buttons have been pressed and this Iconbar icon has a Toolbox event associated with this, then that Toolbox event is raised, and any attached object is also shown using Toolbox_ShowObject. |
| User Msg | Message_HelpRequest (for this Iconbar icon)<br>If a help message is attached to this Iconbar icon, then a reply is sent on the application's behalf. |

# 10 Menu class

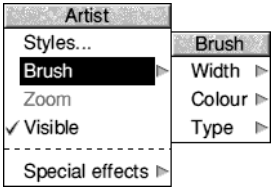
A menu allows the user to select an item from a list of choices using the mouse pointer.

## User interface

A menu should appear on the screen either when the user clicks the Menu mouse button, or clicks on a Pop-up menu button. The menu will disappear again when the user clicks outside the menu or presses Escape (or the client application hides it or the user opens another menu).

When the user clicks on a menu entry the client application will typically perform some task. The menu will then disappear, unless the selection was made using the Adjust button in which case it will persist on the screen.

- A menu has a title bar with black (Wimp colour 7) text on a grey (Wimp colour 2) background.
- Menu entries which contain text are black (7) on a white (0) background; a menu entry may alternatively contain a sprite.
- Menu entries may optionally be separated by a dotted line, to group related items.
- A menu entry may lead to further menus, or a dialogue box, in which case a submenu arrow is displayed at the righthand edge of the entry. When a menu entry is unavailable it is displayed as ‘shaded’ (i.e. its text is displayed in light grey).



## Application Program Interface

When a Menu object is created, the Toolbox deals with ensuring that the colours used for the Menu are Style Guide compliant. Each menu entry is set with a height of 44 OS units (or 68 if it has a dotted line separator), and the width of the menu is calculated from details of its entries on the application's behalf.

The Menu module deals with keeping the menu tree displayed when a selection is made with Adjust.

### Attributes

**Menu attributes**

A Menu object has the following attributes which are specified in its object template and can be manipulated at run-time by the client application:

| Attribute | Description | |
|---|---|---|
| flags word | **Bit** | **Meaning** |
| | 0 | when set, this bit indicates that an event should be raised when SWI Toolbox_ShowObject is called for this Menu. |
| | 1 | when set, this bit indicates that an event should be raised when the Menu has been removed from the screen. |
| menu title | gives a text string which will appear in the menu's title bar (0 means no title, an empty string means no titlebar) | |
| max title length | gives the maximum length in bytes of title text which will be used for this Menu. | |
| help message | when a HelpRequest message is received on this menu, then this text message is sent in a HelpReply message. Note that this help message is only sent if the menu entry for which the request was received has not got a help message of its own. | |
| max help length | gives the maximum length in bytes of help text which will be used for this Menu. | |
| show event | this is a Toolbox event code which will be raised when SWI Toolbox_ShowObject is called for this menu. If its value is -1, then the default Menu_AboutToBeShown event is raised. An event is only raised if the appropriate bit is set in the menu's flags word. | |

| Attribute | Description |
|---|---|
| hide event | this is a Toolbox event code which will be raised when this menu has been removed from the screen (either as a result of an explicit call to SWI Toolbox_HideObject or because the Wimp has removed the menu).<br>If its value is -1, then the default Menu_HasBeenHidden event is raised. An event is only raised if the appropriate bit is set in the menu's flags word. |

**Menu entry attributes**

A Menu also has a list of ‘entries’. Each entry has its own component id which uniquely identifies it within this menu. An entry has the following attributes:

| Attribute | Description |
|---|---|
| flags | **Bit** **Meaning**<br>0 when set, this entry is ticked.<br>1 when set, this entry has a dotted line immediately after it.<br>2-7 must be 0.<br>8 when set, this entry is faded.<br>9 when set, this entry is a sprite (default is a text menu entry).<br>10 when set, this entry has a submenu (ie a submenu arrow appears next to the entry).<br>11 when set, an event (either Menu_SubMenu or client-specified) is raised when the user traverses this entry's submenu arrow with the mouse pointer (if bit 10 is set).<br>12 when set, if there is an object to be shown when this entry is selected, then it will be shown with Wimp_CreateMenu semantics. The default is to show persistently. |
| component id | identifies this entry uniquely within this menu.<br>-1 and -2 are invalid component ids |
| text | depending on whether this is a text or sprite entry (as indicated by bit 9 of the flags word), this is either:<br>● a text string which will appear in the menu entry<br>● the name of the sprite which will appear in the Menu entry |

| Attribute | Description |
|---|---|
| max length | gives the maximum length in bytes of entry text or sprite name |
| click show | the name of the template for an object to show, when the user clicks on this entry.<br>0 means there is no object to be shown |
| submenu show | the name of the template for an object to show, when the user moves the pointer over the submenu arrow (if the entry has a submenu).<br>0 means there is no object to be shown |
| submenu event | a Toolbox event code which will be raised when the user moves the pointer over the submenu arrow (if the entry has a submenu and bit 11 of the flags word is set)<br>if its value is 0 then the default Menu_Submenu event is raised |
| click event | a Toolbox event code which will be raised when the user clicks on this entry<br>if its value is 0 then the default Menu_Selection event is raised |
| help message | when a HelpRequest message is received on this entry of this menu then this text string is sent in a HelpReply message<br>0 means that the help message for the menu will be sent (if such exists) |
| max help length | gives the maximum length in bytes of the entry's help message |

## Manipulating a Menu object

Since there can only be one Menu visible on the screen at any one time, it is usual for the client application to mark Menu templates as ‘shared’ so that only one copy will exist in memory. The application receives a Menu_AboutToBeShown Toolbox event just before the Menu is shown, to allow it to set any attributes like ticks and fades, which may differ depending on where the Menu is being shown; for example, in a multi-document editor a single menu can be maintained for all document Windows; when the Toolbox receives a Menu button click event from the Wimp, it will show the Menu associated with the Window over which the mouse click occurred; when the application receives the Menu_AboutToBeShown Toolbox event, it can tick and fade entries in the Menu depending on the state of the document Window.

Another alternative for supporting multi-document editors is to create a Menu object for each Window object. In this case it will not be necessary to use the Menu_AboutToBeShown Toolbox event to make last minute changes to the menu, since these can be made on a per-window basis as the changes occur. Whether this method is used, or the above ‘shared’ scheme is really one of personal taste, and memory usage.

It is possible to associate a client handle with a Menu using the Toolbox_SetClientHandle method, but normally an application will simply wish to use the client handle of the object to which a Menu is attached (via the parent_id or the ancestor_id in the id block).

### Creating and deleting a menu

A Menu object is created using SWI Toolbox_CreateObject.

When a Menu object is created, the following attached objects (see page 11) are also created for each menu entry for which they are defined:

- submenu show
- click show.

The *Menu entry attributes* table on page 167 describes these objects.

Attached objects are also created when a menu entry is added to the Menu, if they are referenced by the menu entry (and deleted when the menu entry is removed).

A Menu object is deleted using SWI Toolbox_DeleteObject. If it has any attached objects these are also deleted, unless the non-recursive bit is set for this SWI.

Note: Menus must not be mutually recursive (i.e. in a menu hierarchy, a menu entry may not have, as a submenu, a menu further up the hierarchy). The menu module does not check for such a case, so it is the client application’s responsibility to check for correctness.

### Showing a menu

When a menu is displayed on the screen using SWI Toolbox_ShowObject it has the following behaviour:

| Show type | Position |
|---|---|
| 0 (default) | 64 OS units to the left of the mouse pointer |
| 1 (full spec) | R3 + 0 gives x coordinate of top-left corner of Menu<br>R3 + 4 gives y coordinate of top-left corner of Menu |
| 2 (topleft) | R3 + 0 gives x coordinate of top-left corner of Menu<br>R3 + 4 gives y coordinate of top-left corner of Menu |

The client application should not need to make this call, since it is made automatically by the Window and Iconbar modules for objects which have a Menu attached to them.The Window module will display the menu in its default place when the Menu button is clicked, or in the case of a pop-up menu directly to the right of the pop-up icon; the Iconbar module displays the menu with its base 96 OS units from the bottom of the screen, and 64 OS units to the right of the mouse pointer.

### Adding and removing menu entries

Normally the set of entries in a Menu will be specified in the application's resource file. If, however, the application wishes to add and remove Menu entries dynamically at run-time, this is done using the Menu_AddEntry and Menu_RemoveEntry methods.

### Changing a Menu entry

A given Menu entry can either contain text or a sprite. Normally these will be fixed when the menu is created, but they can be set and read dynamically using the Menu_SetEntryText, Menu_GetEntryText, Menu_SetEntrySprite, and Menu_GetEntrySprite methods.

### Ticking or fading a Menu entry

Each Menu entry can be optionally 'ticked' (i.e. have a tick displayed to the left of it), and/or 'faded' (i.e. displayed in light grey, and unselectable).

A given Menu entry can be ticked/unticked, faded/unfaded using the Menu_SetTick/Menu_SetFade methods.

The client can determine the state of a particular entry using the Menu_GetTick/Menu_GetFade methods.

### Attaching a submenu dynamically

Normally an application's Menu structure is fully specified statically in its resource file, but occasionally an application may wish to build a submenu at run-time, and attach it at a particular point in the Menu tree.

This is achieved by creating the submenu object, and using the Menu_SetSubMenuShow method already mentioned (and detailed on page 181).

### Dealing with Menu hits

Each Menu entry can have a specified Toolbox event which will be raised when a menu selection is made on that entry (i.e. the Wimp has returned a Menu Selection event to the application).

Normally this Toolbox event is specified in the client application's resource file, but it can be read and set dynamically using the Menu_SetClickEvent and Menu_GetClickEvent methods.

The client can also specify the name of a template of an object which should be shown when the menu hit happens. The main use for this is to supply the name of the template of a persistent dialogue box, on a Menu entry with an ellipsis (...). The object is only shown after the 'Menu hit event' has been delivered to the client. The **show type** value passed in R2 to Toolbox_ShowObject will be 0 (default place).

It is possible to specify at run-time the object id of an object which should be shown when a Menu hit happens, using the Menu_SetClickShow method (and the object id can be read using the Menu_GetClickShow method).

If neither of the above is specified, then the Toolbox raises the Menu_Selection Toolbox event, as described on page 198. This Toolbox event reports which entry was selected.

### Dealing with Adjust clicks on a Menu

When the user of the client application clicks Adjust on a Menu entry or on a Gadget in a dialogue box which has been opened from a Menu, it is conventional for the Menu tree to remain on the screen.

The Toolbox handles this automatically on behalf of the application, so the client does not have to look for Adjust clicks; the client's code just responds to the Toolbox events raised by the user's interaction with the Menu.

Note that the Toolbox 're-shows' the Menu when the application next calls SWI Wimp_Poll, after the Menu selection, so any ticking/fading etc of Menu entries, must be done in response to the Toolbox event which was raised when a menu selection was made.

**Dealing with traversal of a submenu arrow**

Each Menu entry can have a specified Toolbox event which will be raised when the user moves the mouse pointer over the submenu arrow, which is displayed on all Menu entries which have a submenu.

Normally this Toolbox event is specified in the client application's resource file, but it can be read and set dynamically using the Menu_SetSubMenuEvent and Menu_GetSubMenuEvent methods.

The client can also specify the name of a template of an object which should be shown when the user moves the mouse pointer over the submenu arrow. The main use for this is to supply the name of the template of a transient dialogue box or a submenu. The object is only shown after the Menu_SubMenu event has been delivered to the client.

It is possible to specify at run-time the object id of an object which should be shown when the user moves the pointer over the submenu arrow, using the Menu_SetSubMenuShow method (and the object id can be read using the Menu_GetSubMenuShow method).

If neither of the above is specified, then the Toolbox raises the Menu_SubMenu Toolbox event. This Toolbox event reports the entry over which the mouse pointer has moved.

**Interactive help on Menus**

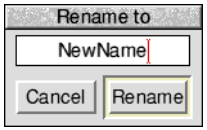
Each Menu has an optional Help Message associated with it. When the client application receives a HelpRequest for the Menu, the Toolbox replies automatically with this Help Message.

Normally the Menu's Help Message will be specified in the application's resource file, however the client can set and read the message dynamically using the Menu_SetHelpMessage/Menu_GetHelpMessage methods.

Each Menu entry can also have a Help Message. If no such message is specified, then the Toolbox will return the Menu's Help Message instead. Normally, again, an entry's Help Message will have been specified in the resource file, but it can be read and set using the Menu_SetEntryHelpMessage and Menu_GetEntryHelpMessage methods (described on page 191).

**Writable menu entries**

Writable menu entries as seen in older applications are not supported by the Toolbox as these are not Style Guide compliant. Instead you should use small dialogues. For example:




## Menu methods

The following methods are all invoked by calling SWI Toolbox_MiscOp with:

| | |
|---|---|
| R0 | holding a flags word |
| R1 | being a Menu id |
| R2 | being the method code which distinguishes this method |
| R3-R9 | potentially holding method-specific data |

### Menu_SetTick 0

**On entry**

R0 = flags
R1 = Menu object id
R2 = 0
R3 = component id of entry

R4 = value
0 means ‘untick’
non-zero means ‘tick’

**On exit**

R1-R9 preserved

**Use**

This method affects the tick state of a Menu entry.

**C veneer**

```
extern _kernel_oserror *menu_set_tick ( unsigned int flags,
                                        ObjectId menu,
                                        ComponentId entry,
                                        int tick
                                      );
```

## Menu_GetTick 1

**On entry**

R0 = flags
R1 = Menu object id
R2 = 0
R3 = component id of entry

**On exit**

R0 = tick state
non-zero means ticked
0 means unticked

**Use**

This method returns the tick state of a Menu entry.

**C veneer**

```
extern _kernel_oserror *menu_get_tick ( unsigned int flags,
                                        ObjectId menu,
                                        ComponentId entry,
                                        int *ticked
                                      );
```

## Menu_SetFade 2

**On entry**

R0 = flags
R1 = Menu object id
R2 = 2
R3 = component id of entry
R4 = value
    0 means unfade
    non-zero means fade

**On exit**

R1-R9 preserved

**Use**

This method affects the fade state of a Menu entry.

**C veneer**

```
extern _kernel_oserror *menu_set_fade ( unsigned int flags,
                                        ObjectId menu,
                                        ComponentId entry,
                                        int fade
                                      );
```

## Menu_GetFade 3

**On entry**

R0 = flags
R1 = Menu object id
R2 = 3
R3 = component id of entry

**On exit**

R0 = fade state
0 means unfaded
non-zero means faded

**Use**

This method returns the fade state of a Menu entry.

**C veneer**

```
extern _kernel_oserror *menu_get_fade ( unsigned int flags,
                                        ObjectId menu,
                                        ComponentId entry,
                                        int *faded
                                      );
```

## Menu_SetEntryText 4

**On entry**

R0 = flags
R1 = Menu object id
R2 = 4
R3 = component id of entry
R4 = pointer to text string to use

**On exit**

R1-R9 preserved

**Use**

This method sets the text which is to be used in the named text Menu entry.

An error is returned if the entry's text buffer is not large enough to hold the supplied text.

An error is returned if this SWI is called on an entry which is a sprite.

**C veneer**

```
extern _kernel_oserror *menu_set_entry_text ( unsigned int flags,
                                              ObjectId menu,
                                              ComponentId entry,
                                              char *text
                                            );
```

## Menu_GetEntryText 5

**On entry**

R0 = flags
R1 = Menu object id
R2 = 5
R3 = component id of entry
R4 = pointer to buffer to return the text in (or 0)
R5 = size of buffer

**On exit**

R5 = the size of buffer required to hold the text (if R4 was 0)
else Buffer pointed to by R4 contains entry text
R5 holds number of bytes written to buffer

**Use**

This method is used for a text Menu entry. It returns the text string displayed for that entry.

**C veneer**

```
extern _kernel_oserror *menu_get_entry_text ( unsigned int flags,
                                              ObjectId menu,
                                              ComponentId entry,
                                              char *buffer,
                                              int buff_size,
                                              int *nbytes
                                            );
```

## Menu_SetEntrySprite 6

**On entry**

R0 = flags
R1 = Menu object id
R2 = 6
R3 = component id of entry
R4 = pointer to name of sprite to use

**On exit**

R1-R9 preserved

**Use**

This method sets the sprite which is to be used in the named sprite Menu entry.

An error is returned if the entry's sprite name buffer is not large enough to hold the supplied sprite name.

An error is returned if this SWI is called on a text entry.

**C veneer**

```
extern _kernel_oserror *menu_set_entry_sprite ( unsigned int flags,
                                                ObjectId menu,
                                                ComponentId entry,
                                                char *sprite_name
                                              );
```

## Menu_GetEntrySprite 7

**On entry**

R0 = flags
R1 = Menu object id
R2 = 7
R3 = component id of entry
R4 = pointer to buffer to return the sprite name in (or 0)
R5 = size of buffer

**On exit**

R5 = the size of buffer required to hold the sprite name (if R4 was 0)
else Buffer pointed to by R4 contains sprite name
R5 holds number of bytes written to buffer

**Use**

This method is used for a sprite Menu entry. It returns the name of the sprite displayed for that entry.

**C veneer**

```
extern _kernel_oserror *menu_get_entry_sprite ( unsigned int flags,
                                                ObjectId menu,
                                                ComponentId entry,
                                                char *buffer,
                                                int buff_size,
                                                int *nbytes
                                              );
```

## Menu_SetSubMenuShow 8

**On entry**

R0 = flags
R1 = Menu object id
R2 = 8
R3 = component id of entry where submenu should be attached
R4 = object id of the submenu (or 0)

**On exit**

R1-R9 preserved

**Use**

This method allows the client to specify the object id of an object to show when the user moves the pointer over the submenu arrow.

If R4 is 0, then no object should be shown.

Calling this SWI also causes the submenu to be shown or hidden as appropriate.

**C veneer**

```
extern _kernel_oserror *menu_set_sub_menu_show ( unsigned int flags,
                                                 ObjectId menu,
                                                 ComponentId entry,
                                                 ObjectId sub_menu
                                               );
```

## Menu_GetSubMenuShow 9

**On entry**

R0 = flags
R1 = Menu object id
R2 = 9
R3 = component id

**On exit**

R0 = id of object to be shown

**Use**

This method returns the object id of the object which will be shown when the user moves the pointer over the submenu arrow.

**C veneer**

```
extern _kernel_oserror *menu_get_sub_menu_show ( unsigned int flags,
                                                 ObjectId menu,
                                                 ComponentId entry,
                                                 ObjectId *sub_menu
                                               );
```

## Menu_SetSubMenuEvent 10

**On entry**

R0 = flags
R1 = Menu object id
R2 = 10
R3 = component id of entry
R4 = Toolbox event code to raise

**On exit**

R1-R9 preserved

**Use**

This method specifies a Toolbox event to be raised when the user moves the mouse over this entry's submenu arrow.

If R4 is 0, then a Menu_SubMenu Toolbox event will be raised instead.

Calling this SWI also causes the submenu to be shown or hidden as appropriate.

**C veneer**

```
extern _kernel_oserror *menu_set_sub_menu_event ( unsigned int flags,
                                                  ObjectId menu,
                                                  ComponentId entry,
                                                  int toolbox_event
                                                );
```

## Menu_GetSubMenuEvent 11

**On entry**

R0 = flags
R1 = Menu object id
R2 = 11
R3 = component id of entry

**On exit**

R4 = Toolbox event code

**Use**

This method reads the Toolbox event to be raised when the user moves the mouse over this entry's submenu arrow.

If no event has been specified, then 0 is returned.

**C veneer**

```
extern _kernel_oserror *menu_get_sub_menu_event ( unsigned int flags,
                                                  ObjectId menu,
                                                  ComponentId entry,
                                                  int *toolbox_event
                                                );
```

## Menu_SetClickShow 12

**On entry**

R0 = flags
R1 = Menu object id
R2 = 12
R3 = component id of entry
R4 = object id to show
R5 = show flags: bit 0
    if clear show persistently
    if set show transiently

**On exit**

R1-R9 preserved

**Use**

This method allows the client to specify the object id of an object to show when the user selects this Menu entry. By setting bit 0 of R5 it is possible to control whether the show is persistent or not.

If R4 is 0, then no object should be shown.

**C veneer**

```
extern _kernel_oserror *menu_set_click_show ( unsigned int flags,
                                              ObjectId menu,
                                              ComponentId entry,
                                              ObjectId object,
                                              int show_flags
                                            );
```

## Menu_GetClickShow 13

**On entry**

R0 = flags
R1 = Menu object id
R2 = 13
R3 = component id

**On exit**

R0 = id of object to be shown
R1 = show flags

**Use**

This method returns the object id of the object which will be shown when the user selects this Menu entry. If bit 0 of R1 is set on exit, it means that the object will be shown transiently.

If no object has been specified, then 0 is returned in R0.

**C veneer**

```
extern _kernel_oserror *menu_get_click_show ( unsigned int flags,
                                              ObjectId menu,
                                              ComponentId entry,
                                              ObjectId *object,
                                              int *show_flags
                                            );
```

## Menu_SetClickEvent 14

**On entry**

R0 = flags
R1 = Menu object id
R2 = 14
R3 = component id of entry
R4 = Toolbox event code to raise

**On exit**

R1-R9 preserved

**Use**

This method specifies a Toolbox event to be raised when the user selects the given Menu entry.

If R4 is 0, then a Menu_Selection Toolbox event will be raised instead.

**C veneer**

```
extern _kernel_oserror *menu_set_click_event ( unsigned int flags,
                                               ObjectId menu,
                                               ComponentId entry,
                                               int toolbox_event
                                             );
```

## Menu_GetClickEvent 15

**On entry**

R0 = flags
R1 = Menu object id
R2 = 15
R3 = component id of entry

**On exit**

R4 = Toolbox event code

**Use**

This method reads the Toolbox event to be raised when the user selects the given Menu entry.

If no event has been specified, then 0 is returned.

**C veneer**

```
extern _kernel_oserror *menu_get_click_event ( unsigned int flags,
                                               ObjectId menu,
                                               ComponentId entry,
                                               int *toolbox_event
                                             );
```

## Menu_SetHelpMessage 16

**On entry**

R0 = flags
R1 = Menu object id
R2 = 16
R3 = pointer to message text

**On exit**

R1-R9 preserved

**Use**

This method is used to set the help message which will be returned when a Help Request message is received for this Menu object. The Toolbox handles the reply message for you.

If R3 is 0, then the Help Message for this Menu is detached.

**C veneer**

```
extern _kernel_oserror *menu_set_help_message ( unsigned int flags,
                                                ObjectId menu,
                                                char *help_message
                                              );
```

## Menu_GetHelpMessage 17

**On entry**

R1 = Menu object id
R2 = 17
R3 = pointer to buffer
R4 = size of buffer to hold message text

**On exit**

R4 = size of buffer required for message text (if R3 was 0)
else Buffer pointed at by R3 holds message text
R4 holds number of bytes written to buffer

**Use**

This method is used to read the help message which will be returned when a Help Request message is received for this Menu object.

**C veneer**

```
extern _kernel_oserror *menu_get_help_message ( unsigned int flags,
                                                ObjectId menu,
                                                char *buffer,
                                                int buff_size,
                                                int *nbytes
                                              );
```

## Menu_SetEntryHelpMessage 18

**On entry**

R0 = flags
R1 = Menu object id
R2 = 18
R3 = component id of entry
R4 = pointer to message text

**On exit**

R1-R9 preserved

**Use**

This method is used to set the help message which will be returned when a Help Request message is received for this Menu entry. The Toolbox handles the reply message for you.

If R4 is 0, then the Help Message for this Menu entry is detached.

**C veneer**

```
extern _kernel_oserror *menu_set_entry_help_message ( unsigned int flags,
                                                      ObjectId menu,
                                                      ComponentId entry,
                                                      char *help_message
                                                    );
```

## Menu_GetEntryHelpMessage 19

**On entry**

R0 = flags
R1 = Menu object id
R2 = 19
R3 = component id of entry
R4 = pointer to buffer
R5 = size of buffer to hold message text

**On exit**

R5 = size of buffer required for message text (if R4 was 0)
else Buffer pointed at by R4 holds message text
R5 holds number of bytes written to buffer

**Use**

This method is used to read the help message which will be returned when a Help Request message is received for this Menu object.

**C veneer**

```
extern _kernel_oserror *menu_get_entry_help_message ( unsigned int flags,
                                                      ObjectId menu,
                                                      ComponentId entry,
                                                      char *buffer,
                                                      int buff_size,
                                                      int *nbytes
                                                    );
```

## Menu_AddEntry 20

**On entry**

R0 = flags (bit 0 set means add the entry before the specified entry)
R1 = Menu object id
R2 = 20
R3 = component id of entry after/before which to add this entry
(or -1 to mean at the beginning, -2 to mean at the end)
R4 = pointer to buffer containing a description of the new entry

**On exit**

R0 = component id of added entry
R1-R9 preserved

**Use**

This method adds a new Menu entry at the specified place in the Menu. The description of the Menu entry should have a format as specified under the Menu Templates section.

By default the entry is added after the specified entry whose id is passed in R3, but the client can specify that it is added before that entry, by setting bit 0 of the flags word.

If the component id in the template of the Menu entry was specified as -1, then the Toolbox uses the lowest numbered component id available for this Menu.

**C veneer**

```
extern _kernel_oserror *menu_add_entry ( unsigned int flags,
                                         ObjectId menu,
                                         ComponentId at_entry,
                                         char *entry_description,
                                         ComponentId *new_entry
                                       );
```

## Menu_RemoveEntry 21

**On entry**

R0 = flags
R1 = Menu object id
R2 = 21
R3 = component id of the entry

**On exit**

R1-R9 preserved

**Use**

This method removes a Menu entry

**C veneer**

```
extern _kernel_oserror *menu_remove_entry ( unsigned int flags,
                                            ObjectId menu,
                                            ComponentId entry
                                          );
```

## Menu_GetHeight 22

**On entry**

R0 = flags
R1 = Menu object id
R2 = 22

**On exit**

R0 = height of menu work area in OS Units
R1-R9 preserved

**Use**

This method returns the height of the work area of the given Menu (in OS Units). It takes into account whether items in the Menu have dashed line separators. This can be used to accurately position the Menu in a call to Toolbox_ShowObject.

**C veneer**

```
extern _kernel_oserror *menu_get_height ( unsigned int flags,
                                          ObjectId menu,
                                          int *height
                                        );
```

## Menu_GetWidth 23

**On entry**

R0 = flags
R1 = Menu object id
R2 = 23

**On exit**

R0 = width of menu work area in OS Units
R1-R9 preserved

**Use**

This method returns the width of the work area of the given Menu (in OS Units).

**C veneer**

```
extern _kernel_oserror *menu_get_width ( unsigned int flags,
                                         ObjectId menu,
                                         int *width
                                       );
```

## Menu_SetTitle 24

**On entry**

R0 = flags
R1 = Menu object id
R2 = 24
R3 = pointer to text string to use

**On exit**

R1-R9 preserved

**Use**

This method sets the text which is to be used in the title bar of the given Menu. Note that this has no immediate effect if the Menu is currently being displayed.

**C veneer**

```
extern _kernel_oserror *menu_set_title ( unsigned int flags,
                                         ObjectId menu,
                                         char *title
                                       );
```

## Menu_GetTitle 25

**On entry**

R0 = flags
R1 = Menu object id
R2 = 25
R3 = pointer to buffer to return the text in (or 0)
R4 = size of buffer

**On exit**

R4 = the size of buffer required to hold the text (if R3 was 0)
else Buffer pointed to by R3 contains title text
R4 holds number of bytes written to buffer

**Use**

This method returns the text string used in a Menu's title bar.

**C veneer**

```
extern _kernel_oserror *menu_get_title ( unsigned int flags,
                                         ObjectId menu,
                                         char *buffer,
                                         int buff_size,
                                         int *nbytes
                                       );
```

## Menu events

### Menu_AboutToBeShown (0x828c0)

**Block**

| | |
|---|---|
| + 8 | 0x828c0 (or client specified event – see *Menu Templates* on page 199) |
| + 12 | flags (as passed in to Toolbox_ShowObject) |
| + 16 | value as passed in R2 to ToolBox_ShowObject |
| + 20... | block as passed in R3 to ToolBox_ShowObject |

**Use**

This Toolbox event is raised due to a call to SWI Toolbox_ShowObject on a Menu object which has bit 0 of its flags word set. It gives the application the opportunity to tick, fade or change the text/sprite of any Menu entries before the Menu actually appears on the screen.

This is useful where a shared Menu is being used by many Window objects, each of which has a state which is reflected in the Menu state.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                show_type;
  TopLeft            pos;
} MenuAboutToBeShownEvent;
```

### Menu_HasBeenHidden (0x828c1)

**Block**

| | |
|---|---|
| + 8 | 0x828c1 (or client specified event – see *Menu Templates* on page 199) |

**Use**

This Toolbox event is raised by the Toolbox when Toolbox_HideObject is called on a Menu which has the appropriate bit set in its template flags word. It enables a client application to clear up after a menu has been closed. It is also raised when clicking outside a menu or hitting Escape.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
} MenuHasBeenHiddenEvent;
```

## Menu_SubMenu (0x828c2)

**Block**

+ 8 0x828c2
+ 16 x coordinate where the submenu will be shown
+ 20 y coordinate where the submenu will be shown

**Use**

This Toolbox event is raised when the user moves the mouse over a Menu entry's submenu arrow, and the client application has not associated any other Toolbox event with this event. The event is only delivered if the appropriate bit is set in the menu entry's flags word.

This Toolbox event is raised by the Menu class.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  TopLeft            pos;
} MenuSubMenuEvent;
```

## Menu_Selection (0x828c3)

**Block:**

+ 8 0x828c3

**Use**

This Toolbox event is raised when the user makes a selection on a Menu object, and the client application has not associated any other Toolbox event with this event.

This Toolbox event is raised by the Menu class.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
} MenuSelectionEvent;
```

## Menu Templates

The layout of a Menu template is shown below. Fields which have types MsgReference and StringReference are those which will require relocation when they are loaded from a resource file. If the template is being constructed in memory, then these fields should be real pointers (i.e. they do not require relocation).

The current version for Menu templates is 102.

For more details on relocation, see appendix *Resource File Formats* on page 499.

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| title | 4 | MsgReference |
| max_title | 4 | word |
| help_message | 4 | MsgReference |
| max_help | 4 | word |
| show_event | 4 | word |
| hide_event | 4 | word |
| num_entries | 4 | word |

Followed by a list of menu entries, where each entry is:

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| component_id | 4 | word |
| text | 4 | MsgReference or StringReference |
| max_text | 4 | word |
| click_show | 4 | StringReference |
| submenu_show | 4 | StringReference |
| submenu_event | 4 | word |
| click_event | 4 | word |
| help_message | 4 | MsgReference |
| max_entry_help | 4 | word |

## Menu Wimp event handling

The Menu class responds to certain Wimp events and takes the actions as described below:

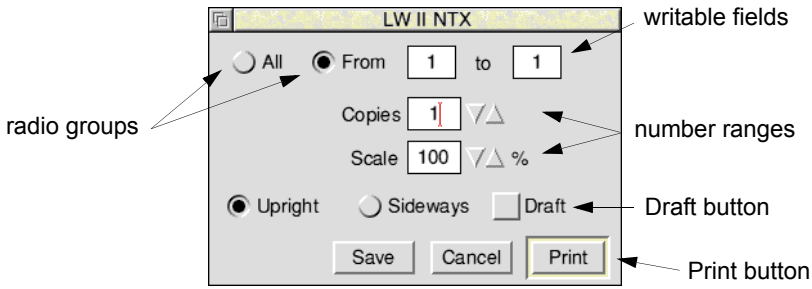
| Wimp event | Action |
|---|---|
| Menu Selection | If there is a click event associated with the given Menu entry, then that Toolbox event is raised;<br>if there is an object to be shown for this entry then show it;<br>if neither of the above then the Menu_Selection Toolbox event is raised.<br>If Adjust is held down, then the currently open Menu is re-opened in the same place. |
| Mouse Click | (on a dialogue box attached to the Menu)<br>If Adjust is held down, then the currently open Menu is re-opened in the same place. |
| User Msg | Message_HelpRequest<br>(while the pointer is over a Menu object) If a help message is attached to this Menu or Menu entry, then a reply is sent on the application's behalf.<br>Message_MenuWarning<br>If a submenu event is associated with the given Menu entry, then this Toolbox event is raised;<br>if a submenu object has been specified for this Menu entry, then it is shown by the Toolbox.<br>if neither of the above, then a Menu_SubMenu Toolbox event is raised.<br>Message_MenusDeleted<br>The Menu which was being shown is marked as hidden (as if Toolbox_HideObject had been called). |

# 11 Print Dialogue box class

A Print dialogue object is used to allow the user to set a number of print options (e.g. number of pages, number of copies etc), and then to request that a document be printed given these options.

## User interface

When a Print dialogue is created, it has the following components:




- a set of buttons and writable fields giving a page range to print (optional)
- a number range giving the number of copies to print (optional)
- a radio group consisting of two buttons, indicating whether the printing is to be done **Upright** or **Sideways** (optional).
- an action button **Save** which saves the current print options (optional)
- an action button **Set Up...** which brings up a dialogue box allowing further print options to be set (optional)
- an action button **Cancel** which closes the dialogue box without printing
- a default action button **Print** which causes a print operation to take place using these print options
- an option button **Draft** indicating that draft standard printing is to be used
- a number range giving a percentage scale factor to apply during printing (optional).

Pressing Escape cancels the dialogue (as well as clicking on the **Cancel** button).

The title bar of the dialogue box displays the name of the currently selected printer or ‘Unknown printer’ if there is no such printer.

## Application Program Interface

All processing of the dialogue box is handled by the Print module, and the client is informed of any user actions via Toolbox events (PrintDbox_Print, PrintDbox_SetUp, PrintDbox_DialogueCompleted and PrintDbox_Save).

### Attributes

A Print Dialogue object has the following attributes which are specified in its object template and can be manipulated at run-time by the client application:

| **Attributes** | **Description** | |
|---|---|---|
| flags word | **Bit** | **Meaning** |
| | 0 | when set, this bit indicates that a PrintDbox_AboutToBeShown event should be raised when SWI Toolbox_ShowObject is called for this object. |
| | 1 | when set, this bit indicates that a PrintDbox_DialogueCompleted event should be raised when the Print Dialogue object has been removed from the screen. |
| | 2 | when set, this bit indicates generate PrintDbox_SetUpAboutToBeShown event before the underlying SetUp object is shown |
| | 3 | when set, dialogue box has the **All/From/To Page Range** options |
| | 4 | when set, dialogue box has the **Copies** writable field |
| | 5 | when set, dialogue box has the **Scale** writable field |
| | 6 | when set, dialogue box has the Orientation options (i.e. **Upright** and **Sideways**) |
| | 7 | when set, dialogue box has **Save** action button |
| | 8 | when set, dialogue box has **Set Up ...** action button |
| | 9 | when set, dialogue box has **Draft** option button |
| | 10 | when set, dialogue box has **From/to** set from **All/From/to** |
| | 11 | when set, dialogue box has **Sideways** (and not **Upright**) selected |
| | 12 | when set, dialogue box has **Draft** selected |
| from | initial value to put in the **From** writable field | |
| to | initial value to put in the **to** writable field | |
| copies | initial value to put in the **Copies** number range | |

| Attributes | Description |
|---|---|
| scale | initial value to put in the **Scale** number range |
| further options | name of the template for a Window object to be displayed when **Setup...** is clicked |
| window | name of the template for an alternative window to use instead of the default one (0 means use default) |

## Manipulating a Print Dialogue object

### Creating and deleting a Print Dialogue object

A Print Dialogue object is created using SWI Toolbox_CreateObject.

When a Print Dialogue object is created, the following attached object (see page 11) will be created (if specified):

- further options.

A Print Dialogue object is deleted using SWI Toolbox_DeleteObject. If it has any attached objects (see above), these are also deleted, unless the non-recursive bit is set for this SWI.

The setting of the non-recursive delete bit means that the SetUp dialogue box will not be deleted.

### Showing a Print Dialogue object

When a Print Dialogue object is displayed on the screen using SWI Toolbox_ShowObject it has the following behaviour:

| Show type | Position | |
|---|---|---|
| 0 (default) | the underlying window is shown at the last place shown on the screen, or the coordinates given in its template, if it has not already been shown | |
| 1 (full spec) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |
| | R3 + 8 | visible area maximum x coordinate |
| | R3 + 12 | visible area maximum y coordinate |
| | R3 + 16 | scroll x offset relative to work area |
| | R3 + 20 | scroll y offset relative to work area |
| | R3 + 24 | Wimp window handle of window to open behind |
| | | -1 means top of stack |
| | | -2 means bottom of stack |
| | | -3 means the window behind the Wimp's backwindow |

| Show type | Position | |
|---|---|---|
| 2 (topleft) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |

**Before the Print Dialogue box is shown**

When the client (or the Toolbox) calls Toolbox_ShowObject on a Print Dialogue object, a PrintDbox_AboutToBeShown Toolbox event is raised before the dialogue box becomes visible on the screen (if the appropriate flags bit is set).

This allows the client to set up the contents of the dialogue box appropriately.

**Getting and setting printing options**

A Print dialogue box contains many fields which are either options or writable fields. These are:

- page range
- number of copies
- scale factor
- orientation
- draft.

Each of these components can be read and set dynamically using the following methods:

PrintDbox_SetPageRangePrintDbox_GetPageRange
PrintDbox_SetCopiesPrintDbox_GetCopies
PrintDbox_SetScalePrintDbox_GetScale
PrintDbox_SetOrientationPrintDbox_GetOrientation
PrintDbox_SetDraftPrintDbox_GetDraft

**Responding to action button clicks**

When the user clicks a particular action button (or presses Return or Escape), the client receives one of the following Toolbox events:

- PrintDbox_Save if **Save** has been clicked.
- PrintDbox_Print if **Print** has been clicked or Return has been pressed.
- PrintDbox_SetUp if **Set Up...** has been clicked and there is no specified Window to be shown.

**Getting the Print Dialogue's title**

The string appearing in the Print Dialogue's title bar is the currently selected printer (or 'unknown printer' if there is no such printer). This string can be read using the PrintDbox_GetTitle method.

If the Print Dialogue is persistent, and the currently selected Printer is changed, then the Title Bar will change to reflect this.

**Getting the id of the underlying Window object**

The object id of the Window used to implement a Print Dialogue can be obtained using the PrintDbox_GetWindowID method.

**The SetUp Window**

It is possible to specify the name of a template to be used for showing an object when the **SetUp...** button is pressed. This object is shown in its default place persistently.

## Print Dialogue Methods

The following methods are all invoked by calling SWI Toolbox_ObjectMiscOp with:

| | |
|---|---|
| R0 | holding a flags word |
| R1 | being a Print Dialogue object id |
| R2 | being the method code which distinguishes this method |
| R3-R9 | potentially holding method-specific data |

### PrintDbox_GetWindowID 0

**On entry**

R0 = flags
R1 = Print Dbox object id
R2 = 0

**On exit**

R0 = Window object id for this Print object

**Use**

This method returns the id of the underlying Window object used to implement this Print object.

**C veneer**

```
extern _kernel_oserror *printdbox_get_window_id ( unsigned int flags,
                                                  ObjectId printdbox,
                                                  ObjectId *window
                                                );
```

## PrintDbox_SetPageRange 1

**On entry**

R0 = flags
R1 = Print Dbox object id
R2 = 1
R3 = start of page range
R4 = end of page range

**On exit**

R1-R9 preserved

**Use**

This method is used to set the page range for a Print Dialogue.
A 'start' value of -1 means 'All'.

**C veneer**

```
extern _kernel_oserror *printdbox_set_page_range ( unsigned int flags,
                                                   ObjectId printdbox,
                                                   int start,
                                                   int end
                                                 );
```

## PrintDbox_GetPageRange 2

**On entry**

R0 = flags
R1 = Print Dbox object id
R2 = 2

**On exit**

R0 = start of page range (a 'start' value of -1 means 'All')
R1 = end of page range

**Use**

This method is used to return the page range for a Print Dialogue.

**C veneer**

```
extern _kernel_oserror *printdbox_get_page_range ( unsigned int flags,
                                                   ObjectId printdbox,
                                                   int *start,
                                                   int *end
                                                 );
```

## PrintDbox_SetCopies 3

**On entry**

R0 = flags
R1 = Print Dbox object id
R2 = 3
R3 = number of copies

**On exit**

R1-R9 preserved

**Use**

This method is used to set the number of copies field for a Print Dialogue.

**C veneer**

```
extern _kernel_oserror *printdbox_set_copies ( unsigned int flags,
                                               ObjectId printdbox,
                                               int copies
                                             );
```

## PrintDbox_GetCopies 4

**On entry**

R0 = flags
R1 = Print Dbox object id
R2 = 4

**On exit**

R0 = number of copies to be printed

**Use**

This method returns the value of the **Copies** field for a Print Dialogue.

**C veneer**

```
extern _kernel_oserror *printdbox_get_copies ( unsigned int flags,
                                               ObjectId printdbox,
                                               int *copies
                                             );
```

## PrintDbox_SetScale 5

**On entry**

R0 = flags
R1 = Print Dbox object id
R2 = 5
R3 = percentage value to scale by

**On exit**

R1-R9 preserved

**Use**

This method is used to set the scale factor for a Print Dialogue.

**C veneer**

```
extern _kernel_oserror *printdbox_set_scale ( unsigned int flags,
                                              ObjectId printdbox,
                                              int scale_factor
                                            );
```

## PrintDbox_GetScale 6

**On entry**

R0 = flags
R1 = Print Dbox object id
R2 = 6

**On exit**

R0 = percentage scale factor

**Use**

This method returns the value of the scale factor for a Print Dialogue.

**C veneer**

```
extern _kernel_oserror *printdbox_get_scale ( unsigned int flags,
                                              ObjectId printdbox,
                                              int *scale_factor
                                            );
```

## PrintDbox_SetOrientation 7

**On entry**

R0 = flags
R1 = Print Dbox object id
R2 = 7
R3 = non-zero means Sideways, 0 means Upright

**On exit**

R1-R9 preserved

**Use**

This method is used to set the orientation for a Print Dialogue.

**C veneer**

```
extern _kernel_oserror *printdbox_set_orientation ( unsigned int flags,
                                                    ObjectId printdbox,
                                                    int orientation
                                                  );
```

## PrintDbox_GetOrientation 8

**On entry**

R0 = flags
R1 = Print Dbox object id
R2 = 8

**On exit**

R0 = orientation non-zero means Sideways, 0 means Upright

**Use**

This method returns the orientation for a Print Dialogue.

**C veneer**

```
extern _kernel_oserror *printdbox_get_orientation ( unsigned int flags,
                                                    ObjectId printdbox,
                                                    int *orientation
                                                  );
```

## PrintDbox_GetTitle 9

**On entry**

R0 = flags
R1 = Print Dbox object id
R2 = 9
R3 = pointer to buffer to hold title string
R4 = size of buffer to hold title string

**On exit**

R4 = size of buffer required to hold title string (if R3 was 0)
else buffer pointed at by R3 holds title string
R4 holds number of bytes written to buffer

**Use**

This method returns the current string used in a Print object's title bar.

**C veneer**

```
extern _kernel_oserror *printdbox_get_title ( unsigned int flags,
                                              ObjectId printdbox,
                                              char *buffer,
                                              int buff_size,
                                              int *nbytes
                                            );
```

## PrintDbox_SetDraft 10

**On entry**

R0 = flags
R1 = Print Dbox object id
R2 = 10
R3 = non-zero means Draft, 0 means ‘non-draft’

**On exit**

R1-R9 preserved

**Use**

This method is used to set whether draft printing is used for a Print Dialogue.

**C veneer**

```
extern _kernel_oserror *printdbox_set_draft ( unsigned int flags,
                                              ObjectId printdbox,
                                              int draft
                                            );
```

## PrintDbox_GetDraft 11

**On entry**

R0 = flags
R1 = Print Dbox object id
R2 = 11

**On exit**

R0 = draft non-zero means Draft, 0 means ‘non-draft’

**Use**

This method returns whether draft printing is used for a Print Dialogue.

**C veneer**

```
extern _kernel_oserror *printdbox_get_draft ( unsigned int flags,
                                              ObjectId printdbox,
                                              int *draft
                                            );
```

## Print Dialogue events

The Print module generates the following Toolbox events:

### PrintDbox_AboutToBeShown (0x82b00)

**Block**

| | |
|---|---|
| + 8 | 0x82b00 |
| + 12 | flags (as passed in to Toolbox_ShowObject) |
| + 16 | value which will be passed in R2 to ToolBox_ShowObject |
| + 20... | block which will be passed in R3 to ToolBox_ShowObject for the underlying dialogue box |

**Use**

This Toolbox event is raised just before the Print module is going to show its underlying Window object.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                show_type;
  union
  {
    TopLeft               pos;
    WindowShowObjectBlock full;
  } info;
} PrintDboxAboutToBeShownEvent;
```

## PrintDbox_DialogueCompleted (0x82b01)

**Block**

+ 8 0x82b01
+ 12 flags

**Use**

This Toolbox event is raised after the Print object has been hidden, either by a Cancel click, or after a successful print, or by the user clicking outside the dialogue box (if it is transient) or pressing Escape. It allows the client to tidy up its own state associated with this dialogue.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
} PrintDboxDialogueCompletedEvent;
```

## PrintDbox_SetUpAboutToBeShown (0x82b02)

**Block**

| | |
|---|---|
| + 8 | 0x82b02 |
| + 16 | object id of the object about to be shown<br>(note that the 'self' id in the id block will be for the Print Dialogue object, not the object which will be shown) |
| + 20 | value which will be passed in R2 to ToolBox_ShowObject |
| + 24... | block which will be passed in R3 to ToolBox_ShowObject for the underlying dialogue box |

**Use**

This Toolbox event is raised just before the Print module is going to show its underlying Window object.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  ObjectId           object_id;
  int                show_type;
  union
  {
    TopLeft               pos;
    WindowShowObjectBlock full;
  } info;
} PrintDboxSetUpAboutToBeShownEvent;
```

## PrintDbox_Save (0x82b03)

**Block**

| | |
|---|---|
| + 8 | 0x82b03 |
| + 12 | flags<br>bit 0 set means print Sideways (default is Upright)<br>bit 1 set means print Draft (default is non-draft) |
| + 16 | page range start (-1 means All) |
| + 20 | page range end |
| + 24 | number of copies |
| + 28 | value to scale by (a percentage) |

**Use**

This Toolbox event is raised when the user clicks on the **Save** button. The client should save any options associated with this Print Dialogue (usually in a document which is being edited).

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                start_page;
  int                finish_page;
  int                copies;
  int                scale_factor;
} PrintDboxSaveEvent;
```

## PrintDbox_SetUp (0x82b04)

**Block**

| | |
|---|---|
| + 8 | 0x82b04 |

**Use**

This Toolbox event is raised when the user clicks on the **Set Up...** button, if there is no dialogue box associated with this button.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
} PrintDboxSetUpEvent;
```

## PrintDbox_Print (0x82b05)

**Block**

| | |
|---|---|
| + 8 | 0x82b05 |
| + 12 | flags<br>bit 0 set means print Sideways (default is Upright)<br>bit 1 set means print Draft (default is non-draft) |
| + 16 | page range start (-1 means All) |
| + 20 | page range end |
| + 24 | number of copies |
| + 28 | value to scale by (a percentage) |

**Use**

This Toolbox event is raised when the user clicks on the Print button or presses Return.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                start_page;
  int                finish_page;
  int                copies;
  int                scale_factor;
} PrintDboxPrintEvent;
```

## Print Dialogue templates

The layout of a Print template is shown below. Fields which have types MsgReference and StringReference are those which will require relocation when they are loaded from a resource file. If the template is being constructed in memory, then these fields should be real pointers (i.e. they do not require relocation).

For more details on relocation, see appendix *Resource File Formats* on page 499.

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| from | 4 | word |
| to | 4 | word |
| copies | 4 | word |
| scale | 4 | word |
| further_options | 4 | StringReference |
| window | 4 | StringReference |

### Underlying window template

The Window object used to implement a Print dialogue, has the following characteristics. These must be reproduced if the Window is replaced by a client-specified alternative Window template:

Title bar must be indirected.

#### Gadgets

Component ids are derived by adding to 0x82b000.

| Component id | Details | |
|---|---|---|
| 0 | action button (**Print**) | this should be marked as the 'default' action button |
| 1 | action button (**Save**) | this should be marked as a 'local' action button |
| 2 | action button (**Cancel**) | this should be marked as the 'cancel' action button |
| 3 | radio button (**From**/**To**) | this is selected to allow page ranges to be printed |
| 4 | radio button (**All**) | selected for all page print |
| 5 & 6 | writable field (**From**)<br>writable field (**To**) | these are used by the user to enter a page range |
| 7 | number range (**Copies**) | these are used by the user to enter the number of copies |

| Component id | Details | |
|---|---|---|
| 8 | number range (**Scale**) | these are used by the user to specify a scale |
| 9 | radio button (**Upright**) | selected for portrait |
| 10 | radio button (**Sideways**) | selected for landscape |
| 11 | option button (**Draft**) | selected for draft |
| 12 | action button (**SetUp...**) | this is used to bring up a Window of further options |
| 13 | label (**To**) | |
| 14 | label (**Copies**) | |
| 15 | label (**Scale**) | |
| 16 | label (**%**) | |

## Print Dialogue Wimp event handling

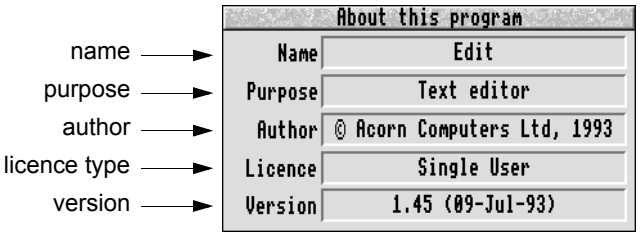
| Wimp event | Action |
|---|---|
| Mouse Click | on **Print** button then raise PrintDbox_Print Toolbox event |
| | on **Cancel** button then raise PrintDbox_DialogueCompleted Toolbox event |
| | on **Save** button then raise PrintDbox_Save Toolbox event |
| | on **Setup...** then raise a PrintDbox_SetUpAboutToBeShown, then show the specified Window object, or raise a PrintDbox_SetUp Toolbox event if there is no such Window |
| | on **All** (pages) and All is off then set All on, set From off and shade the writable fields |
| | on **From** and From is off then set From on, set All to off and unshade the writable fields |
| | on **Copies** or **Scale** up/down arrows then increment/decrement values |
| | on **Upright** then set Upright on and Sideways off |
| | on **Sideways** then set Sideways on and Upright off |
| | on **Draft** then toggle state of option button |
| Key Pressed | if key is Return raise PrintDbox_Print Toolbox event |
| | if key is Escape act as if Cancel has been clicked |
| User Message | Window_HasBeenHidden Toolbox event<br>hide the dialogue box, and raise a PrintDbox_DialogueCompleted Toolbox event |
| | Message_HelpRequest<br>return help message to sender |

# 12 Prog Info Dialogue box class

A Prog Info dialogue object is used to display information about the client application in a dialogue box.

## User interface

A Prog Info Dialogue has the following information held in its dialogue box:




- the name of the application (taken from the message whose tag is '_TaskName')
- the purpose of the application
- the author of the application
- the licence type of the application (optional)
- the version of the application.

All of the above are display field gadgets.

The last of these fields can be set dynamically by the client at run-time.

This gives the simplest of Prog Info Dialogue boxes. If the client wishes to use further fields, or wishes to customise the dialogue box, then there is a facility for including the name of a different template to use rather than the standard Prog Info one.

# Application Program Interface

## Attributes

A Prog Info object has the following attributes which are specified in its object template and can be manipulated at run-time by the client application:

| Attributes | Description | |
|---|---|---|
| flags word | **Bit** | **Meaning** |
| | 0 | when set, this bit indicates that a ProgInfo_AboutTobeShown event should be raised when SWI Toolbox_ShowObject is called for this object. |
| | 1 | when set, this bit indicates that a ProgInfo_DialogueCompleted event should be raised when the ProgInfo object has been removed from the screen. |
| | 2 | when set, include a licence type field in the dialogue box |
| title | alternative title bar string to ‘About this program’ (0 means use default title) | |
| max title length | this gives the maximum length in bytes of title text which will be used for this Prog Info dialogue’s title bar | |
| purpose | a string giving the purpose of this application | |
| author | a string giving the author of this application | |
| licence type | an integer giving the licence type of the application | |
| version | a string giving version information for this application | |
| window | the name of an alternative window template to use instead of the default one (0 means use default) | |

## Manipulating a Prog Info object

### Creating and deleting a Prog Info object

A Prog Info object is created using SWI Toolbox_CreateObject.

When this object is created it has no attached objects (see page 11).

A Prog Info object is deleted using SWI Toolbox_DeleteObject.

The setting of the non-recursive delete bit does not have a meaning for Prog Info objects.

### Showing a Prog Info object

When a Prog Info object is displayed on the screen using SWI Toolbox_ShowObject it has the following behaviour:

| Show type | Position |
|---|---|
| 0 (default) | the underlying window is shown at the last place shown on the screen, or the coordinates given in its template, if it has not already been shown |
| 1 (full spec) | R3 + 0 visible area minimum x coordinate<br>R3 + 4 visible area minimum y coordinate<br>R3 + 8 visible area maximum x coordinate<br>R3 + 12 visible area maximum y coordinate<br>R3 + 16 scroll x offset relative to work area<br>R3 + 20 scroll y offset relative to work area<br>R3 + 24 Wimp window handle of window to open behind<br>-1 means top of stack<br>-2 means bottom of stack<br>-3 means the window behind the Wimp's backwindow |
| 2 (topleft) | R3 + 0 visible area minimum x coordinate<br>R3 + 4 visible area minimum y coordinate |

### Changing the version string

Most of the fields in a Prog Info object will remain unchanged at run-time.

The client may wish to set and read the version string field at run-time. This is done using the ProgInfo_SetVersion/ProgInfo_GetVersion methods.

### Setting the licence type

If the client wishes to set and read the licence type displayed in the Prog Info dialogue box, then it can use the ProgInfo_SetLicenceType and ProgInfo_GetLicenceType methods (described on page 227).

Licence types are one of:

- public domain
- single user
- single machine
- site
- network
- authority.

## Prog Info methods

The following methods are all invoked by calling SWI Toolbox_ObjectMiscOp with:

| | |
|---|---|
| R0 | holding a flags word |
| R1 | being a Prog Info Dialogue object id |
| R2 | being the method code which distinguishes this method |
| R3-R9 | potentially holding method-specific data |

### ProgInfo_GetWindowID 0

**On entry**

R0 = flags
R1 = Prog Info object id
R2 = 0

**On exit**

R0 = Window object id for this Prog Info object

**Use**

This method returns the id of the underlying Window object used to implement this Prog Info object.

**C veneer**

```
extern _kernel_oserror *proginfo_get_window_id ( unsigned int flags,
                                                 ObjectId proginfo,
                                                 ObjectId *window
                                               );
```

## ProgInfo_SetVersion 1

**On entry**

R0 = flags
R1 = Prog Info object id
R2 = 1
R3 = pointer to buffer holding version string (Ctrl-terminated)

**On exit**

R1-R9 preserved

**Use**

This method sets the version string used in the Prog Info Dialogue's Window.

**C veneer**

```
extern _kernel_oserror *proginfo_set_version ( unsigned int flags,
                                               ObjectId proginfo,
                                               char *version_string
                                             );
```

## ProgInfo_GetVersion 2

**On entry**

R0 = flags
R1 = Prog Info object id
R2 = 2
R3 = pointer to buffer to hold version string
R4 = size of buffer to hold version string

**On exit**

R4 = size of buffer required to hold version string (if R3 was 0)
else buffer pointed at by R3 holds version string
R4 holds number of bytes written to buffer

**Use**

This method returns the current version string used in a Prog Info object.

**C veneer**

```
extern _kernel_oserror *proginfo_get_version ( unsigned int flags,
                                               ObjectId proginfo,
                                               char *buffer,
                                               int buff_size,
                                               int *nbytes
                                             );
```

## ProgInfo_SetLicenceType 3

**On entry**

R0 = flags
R1 = Prog Info object id
R2 = 3
R3 = licence type
- 0 ⇒ public domain
- 1 ⇒ single user
- 2 ⇒ single machine
- 3 ⇒ site
- 4 ⇒ network
- 5 ⇒ authority

**On exit**

R1-R9 preserved

**Use**

This method sets the licence type used in the Prog Info Dialogue's Window.

**C veneer**

```
extern _kernel_oserror *proginfo_set_licence_type ( unsigned int flags,
                                                    ObjectId proginfo,
                                                    int licence_type
                                                  );
```

## ProgInfo_GetLicenceType 4

**On entry**

R0 = flags
R1 = Prog Info object id
R2 = 4

**On exit**

R0 = licence type of application
- 0 ⇒ public domain
- 1 ⇒ single user
- 2 ⇒ single machine
- 3 ⇒ site
- 4 ⇒ network
- 5 ⇒ authority

**Use**

This method returns the current licence type used in a Prog Info object.

**C veneer**

```
extern _kernel_oserror *proginfo_get_licence_type ( unsigned int flags,
                                                    ObjectId proginfo,
                                                    int *licence_type
                                                  );
```

## ProgInfo_SetTitle 5

**On entry**

R0 = flags
R1 = Prog Info object id
R2 = 5
R3 = pointer to text string to use

**On exit**

R1-R9 preserved

**Use**

This method sets the text which is to be used in the title bar of the given Prog Info dialogue.

**C veneer**

```
extern _kernel_oserror *proginfo_set_title ( unsigned int flags,
                                             ObjectId proginfo,
                                             char *title
                                           );
```

## ProgInfo_GetTitle 6

**On entry**

R0 = flags 6
R3 = pointer to buffer to return the text in (or 0)
R4 = size of buffer

**On exit**

R4 = size of buffer required to hold the text (if R3 was 0)
else Buffer pointed to by R3 contains title text
R4 holds number of bytes written to buffer

**Use**

This method returns the text string used in a Prog Info dialogue's title bar.

**C veneer**

```
extern _kernel_oserror *proginfo_get_title ( unsigned int flags,
                                             ObjectId proginfo,
                                             char *buffer,
                                             int buff_size,
                                             int *nbytes
                                           );
```

## Prog Info events

The Prog Info module generates the following Toolbox events:

### ProgInfo_AboutToBeShown (0x82b40)

**Block**

| | |
|---|---|
| + 8 | 0x82b40 |
| + 12 | flags (as passed in to Toolbox_ShowObject) |
| + 16 | value which will be passed in R2 to ToolBox_ShowObject |
| + 20... | block which will be passed in R3 to ToolBox_ShowObject for the underlying dialogue box |

**Use**

This Toolbox event is raised just before the Prog Info module is going to show its underlying Window object.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                show_type;
  union
  {
    TopLeft               pos;
    WindowShowObjectBlock full;
  } info;
} ProgInfoAboutToBeShownEvent;
```

### ProgInfo_DialogueCompleted (0x82b41)

**Block**

| | |
|---|---|
| + 8 | 0x82b41 |
| + 12 | flags<br>(none yet defined) |

**Use**

This Toolbox event is raised after the Prog Info object has been hidden, either by the user clicking outside the dialogue box or pressing Escape. It allows the client to tidy up its own state associated with this dialogue.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
} ProgInfoDialogueCompletedEvent;
```

## Prog Info templates

The layout of a Prog Info template is shown below. Fields which have types MsgReference and StringReference are those which will require relocation when they are loaded from a resource file. If the template is being constructed in memory, then these fields should be real pointers (i.e. they do not require relocation).

For more details on relocation, see appendix *Resource File Formats* on page 499.

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| title | 4 | MsgReference |
| max-title | 4 | word |
| purpose | 4 | MsgReference |
| author | 4 | MsgReference |
| licence_type | 4 | word |
| version | 4 | MsgReference |
| window | 4 | StringReference |

### Underlying window template

The Window object used to implement a Prog Info dialogue, has the following characteristics. These must be reproduced if the Window is replaced by a client-specified alternative Window template.

Title bar must be indirected.

**Gadgets**

Component ids are derived by adding to 0x82b400.

| Component id | Details |
|---|---|
| 0 | display field (Name of Application) |
| 1 | display field (Purpose) |
| 2 | display field (Author) |
| 3 | display field (Licence Type) |
| 4 | display field (Version) |
| 5 | label (name) |
| 6 | label (purpose) |
| 7 | label (author) |
| 8 | label (licence) |
| 9 | label (version) |

## Prog Info Wimp event handling

| Wimp event | Action |
|---|---|
| Open Window | request show the dialogue box |
| Key Click | if Escape then cancel dialogue |
| User Message | Message_MenusDeleted<br>hide the dialogue box |

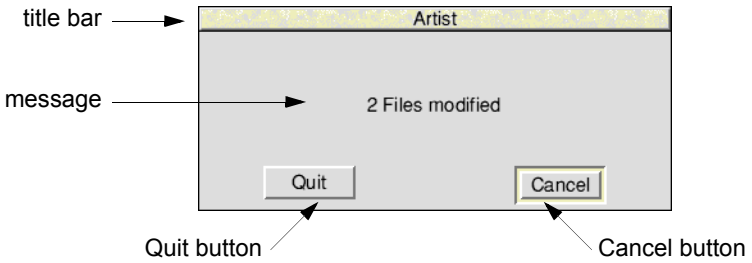
# 13 Quit Dialogue box class

A Quit Dialogue box is used by the client application when the user attempts to quit the application or shut down the computer whilst there is still unsaved data.

## User interface

A Quit Dialogue object is used to warn the user of quitting without saving unsaved data.

The dialogue box which appears on the screen has a number of components:




- a title bar (by default containing the name of the application, i.e. the message whose tag is '_TaskName')
- a message stating (by default) that there is unsaved data
- two action buttons:
  - a Cancel button (default action button)
  - a Quit button.

The user sees the following behaviour:

- if they click on Quit, the application quits
- if they click on Cancel (or press Return or Escape), the application returns to normal operation.

## Application Program Interface

When a Quit object is created, it has a number of optional components:

- an alternative title bar string instead of the client's name

- an alternative message to use in the dialogue box
- the name of an alternative template to use for the underlying Window object.

If the dialogue box is opened as a transient dialogue box, then it closes when the user clicks outside the box.

Just before the Quit dialogue box is shown on the screen, the client is delivered a Quit_AboutToBeShown Toolbox event (if enabled by the appropriate bit in the flags).

Once the dialogue box is displayed on the screen, the Quit module handles events for it, and raises a number of Toolbox events to indicate what choice the user has made. These are Quit_DialogueCompleted, Quit_Cancel and Quit_Quit (respectively).

## Attributes

A Quit object has the following attributes which are specified in its object template and can be manipulated at run-time by the client application:

| **Attributes** | **Description** | |
|---|---|---|
| flags word | **Bit** | **Meaning** |
| | 0 | when set, this bit indicates that a Quit_AboutToBeShown event should be raised when SWI Toolbox_ShowObject is called for this object. |
| | 1 | when set, this bit indicates that a Quit_DialogueCompleted event should be raised when the Quit object has been removed from the screen. |
| title | alternative title to use instead of client's name (0 means default title) | |
| max title length | this gives the maximum length in bytes of title text which will be used for this object | |
| message | the string to use as the message in the Quit dialogue box (0 means default message) | |
| max message | maximum length of string used in dialogue's message field | |
| window | alternative window template to use instead of the default one | |

## Manipulating a Quit object

### Creating and deleting a Quit object

A Quit object is created using SWI Toolbox_CreateObject.

When this object is created it has no attached objects (see page 11).

A Quit object is deleted using SWI Toolbox_DeleteObject.

The setting of the non-recursive delete bit does not have a meaning for Quit objects.

### Showing a Quit object

When a Quit object is displayed on the screen using SWI Toolbox_ShowObject it has the following behaviour:

| Show type | Position | |
|---|---|---|
| 0 (default) | the underlying window is shown at the last place shown on the screen, or the coordinates given in its template, if it has not already been shown | |
| 1 (full spec) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |
| | R3 + 8 | visible area maximum x coordinate |
| | R3 + 12 | visible area maximum y coordinate |
| | R3 + 16 | scroll x offset relative to work area |
| | R3 + 20 | scroll y offset relative to work area |
| | R3 + 24 | Wimp window handle of window to open behind<br>-1 means top of stack<br>-2 means bottom of stack<br>-3 means the window behind the Wimp's backwindow |
| 2 (topleft) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |

### Changing the Quit Dialogue's message

When a Quit Dialogue object is created it has a default message warning the user that he has unsaved data which will be lost if he quits the application.

This can be set and read dynamically using the Quit_SetMessage and Quit_GetMessage methods.

### Getting the id of the underlying window for a Quit Dialogue

The Window object id of the Window object used to implement the Quit Dialogue can be obtained by using the Quit_GetWindowID method.

## Quit methods

The following methods are all invoked by calling SWI Toolbox_ObjectMiscOp with:

| | |
|---|---|
| R0 | holding a flags word (which is zero unless otherwise stated) |
| R1 | being a Quit Dialogue object id |
| R2 | being the method code which distinguishes this method |
| R3-R9 | potentially holding method-specific data |

### Quit_GetWindowID 0

**On entry**

R0 = flags
R1 = Quit object id
R2 = 0

**On exit**

R0 = Window object id for this Quit object

**Use**

This method returns the id of the underlying Window object used to implement this Quit object.

**C veneer**

```
extern _kernel_oserror *quit_get_window_id ( unsigned int flags,
                                             ObjectId quit,
                                             ObjectId *window
                                           );
```

### Quit_SetMessage 1

**On entry**

R0 = flags
R1 = Quit object id
R2 = 1
R3 = pointer to buffer holding new message (Ctrl-terminated)

**On exit**

R1-R9 preserved

**Use**

This method sets the message used in the Quit Dialogue's Window.

### C veneer

```
extern _kernel_oserror *quit_set_message ( unsigned int flags,
                                           ObjectId quit,
                                           char *message
                                         );
```

## Quit_GetMessage 2

**On entry**

R0 = flags
R1 = Quit object id
R2 = 2
R3 = pointer to buffer to hold message
R4 = size of buffer to hold message

**On exit**

R4 = size of buffer required to hold message (if R3 was 0)
else buffer pointed at by R3 holds message
R4 holds number of bytes written to buffer

**Use**

This method returns the current message used in a Quit object.

**C veneer**

```
extern _kernel_oserror *quit_get_message ( unsigned int flags,
                                           ObjectId quit,
                                           char *buffer,
                                           int buff_size,
                                           int *nbytes
                                         );
```

## Quit_SetTitle 3

**On entry**

R0 = flags
R1 = Quit object id
R2 = 3
R3 = pointer to text string to use

**On exit**

R1-R9 preserved

**Use**

This method sets the text which is to be used in the title bar of the given Quit dialogue.

**C veneer**

```
extern _kernel_oserror *quit_set_title ( unsigned int flags,
                                         ObjectId quit,
                                         char *title
                                       );
```

## Quit_GetTitle 4

**On entry**

R0 = flags
R1 = Quit object id
R2 = 4
R3 = pointer to buffer to return the text in (or 0)
R4 = size of buffer

**On exit**

R4 = size of buffer required to hold the text (if R3 was 0)
else Buffer pointed to by R3 contains title text
R4 holds number of bytes written to buffer

**Use**

This method returns the text string used in a Quit dialogue's title bar.

**C veneer**

```
extern _kernel_oserror *quit_get_title ( unsigned int flags,
                                         ObjectId quit,
                                         char *buffer,
                                         int buff_size,
                                         int *nbytes
                                       );
```

## Quit events

The Quit module generates the following Toolbox events:

### Quit_AboutToBeShown (0x82a90)

**Block**

| | |
|---|---|
| + 8 | 0x82a90 |
| + 12 | flags (as passed in to Toolbox_ShowObject) |
| + 16 | value which will be passed in R2 to ToolBox_ShowObject |
| + 20... | block which will be passed in R3 to ToolBox_ShowObject for the underlying dialogue box |

**Use**

This Toolbox event is raised just before the Quit module is going to show its underlying Window object.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                show_type;
  union
  {
    TopLeft               pos;
    WindowShowObjectBlock full;
  } info;

} QuitAboutToBeShownEvent;
```

## Quit_Quit (0x82a91)

**Block**

+ 8 0x82a91

**Use**

This Toolbox event is raised when the user clicks on the Quit Button.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;

} QuitQuitEvent;
```

## Quit_DialogueCompleted (0x82a92)

**Block**

+ 8 0x82a92
+ 12 flags
(none yet defined)

**Use**

This Toolbox event is raised after the Quit object has been hidden, either by a Cancel click, or a Quit click, or by the user clicking outside the dialogue box (if it was opened transiently) or pressing Escape. It allows the client to tidy up its own state associated with this dialogue.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;

} QuitDialogueCompletedEvent;
```

### Quit_Cancel (0x82a93)

**Block**

+ 8 0x82a93

**Use**

This Toolbox event is raised when the user clicks on the Cancel button or presses Return or Escape.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;

} QuitCancelEvent;
```

## Quit templates

The layout of a Quit template is shown below. Fields which have types MsgReference and StringReference are those which will require relocation when they are loaded from a resource file. If the template is being constructed in memory, then these fields should be real pointers (i.e. they do not require relocation).

For more details on relocation, see appendix *Resource File Formats* on page 499.

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| title | 4 | MsgReference |
| max_title | 4 | word |
| message | 4 | MsgReference |
| max_message | 4 | word |
| window | 4 | StringReference |

### Underlying window template

The Window object used to implement a Quit Dialogue, has the following characteristics. These must be reproduced if the Window is replaced by a client-specified alternative Window template.

Title bar must be indirected.

**Gadgets**

Component Ids are derived by adding 0x82a900:

| Component id | Details | |
|---|---|---|
| 0 | button | |
| 1 | action button (Quit) | |
| 2 | action button (Cancel) | must be marked as default and Cancel action button |

## Quit Wimp event handling

| Wimp event | Action |
|---|---|
| Mouse Click | on **Quit** button raise Quit_Quit and Quit_DialogueCompleted (if enabled) Toolbox event |
| | on **Cancel** button raise Quit_Cancel and Quit_DialogueCompleted (if enabled) Toolbox event |
| Key Pressed | if key is Return raise Quit_Cancel Toolbox event |
| | if key is Escape act as if Cancel had been pressed |

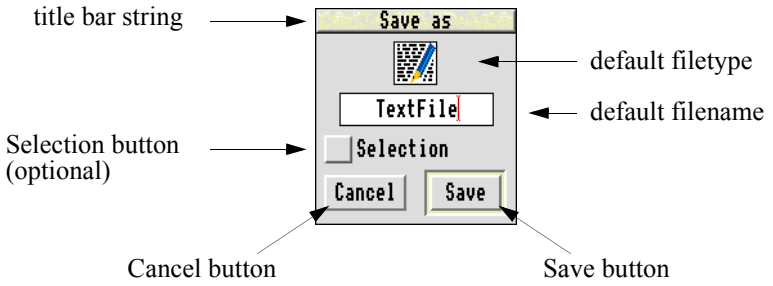
# 14 SaveAs Dialogue box class

Objects of the Save As Dialogue class are used to display a standard (or customised) Save As dialogue box, and to handle the drag of the ‘file icon’ to its destination, and to request the client application to do the save operation. Most of the Wimp message protocol is hidden from the client.

## User interface

A Save As Dialogue object is used to allow the user to drag an icon representing a document from a dialogue box to another application or to a directory display.

When a Save As Dialogue object is created, it has a number of components:




It is possible to specify the following:

- a default filename to use in the Save As dialogue box
- a default filetype to use in the Save As dialogue box
- a string to use in the dialogue box’s title bar, instead of ‘Save as’.
- the name of a Window template to use instead of the Save As module’s internal Window template.

The default Save As dialogue box, has a draggable sprite to represent the data to be saved, a writable field giving the name to save the data under, a **Save** (default) action button, a **Cancel** action button, and an option button saying whether the whole data or just a selection should be saved. If the client wishes to customise the dialogue box, then the above components must be present in that dialogue box, and must have the same component ids.

If the dialogue box is opened as a transient dialogue box, then it closes when the user clicks outside the box.

The user can interact with the Save As dialogue box in the following ways:

- clicking Cancel or pressing Escape will close the dialogue box, and cancel the Save.
- clicking Save (or pressing Return) will save the data in a file whose name is given by the contents of the Writable Field (if it is a full pathname).
- dragging the sprite to its destination will save the data to that destination, with the 'leaf' part of its name.

When the **Selection** option button is clicked on, then the filename will change to the string 'Selection'.

## Application Program Interface

Once the Save As dialogue box is on display, the Save As module handles much of the messaging protocols associated with saving to another application or to a directory display. The client no longer deals in the normal Wimp protocols for data transfer, but instead responds to Toolbox events raised by the Save As module. In fact in the very simplest of cases, the client does no more than just provide a pointer to the data to be saved, and leaves the rest up to the Save As module.

### Attributes

A Save As object has the following attributes which are specified in its object template and can be manipulated at run-time by the client application:

| **Attributes** | **Description** | |
|---|---|---|
| flags | **Bit** | **Meaning** |
| | 0 | when set, this bit indicates that a SaveAs_AboutToBeShown event should be raised when SWI Toolbox_ShowObject is called for this object. |
| | 1 | when set, this bit indicates that a SaveAs_DialogueCompleted event should be raised when the Save As object has been removed from the screen. |
| | 2 | when set, do not include the **Selection** option button in the dialogue box. This is used by clients where there is no concept of a current selection. |
| | 3 | when set, handle the SaveAs operation entirely in the SaveAs module, from the supplied buffer |
| | 4 | when set, client is willing to support RAM transfers |

| Attributes | Description |
|---|---|
| filename | a message string which gives the default filename to use in the writable field |
| filetype | an integer giving the RISC OS type of the file being saved |
| title | a string to use for the Save As dialogue box title bar, instead of 'Save as' (0 means use the default string) |
| max title length | this gives the maximum length in bytes of title text which will be used for this object |
| window | an alternative window template to use instead of the default one (null implies default) |

## Manipulating a SaveAs object

### Creating and deleting a SaveAs object

A SaveAs object is created using SWI Toolbox_CreateObject.

When this object is created it has no attached objects (see page 11).

A SaveAs object is deleted using SWI Toolbox_DeleteObject.

The setting of the non-recursive delete bit does not have a meaning for SaveAs objects.

### Showing a SaveAs object

When a SaveAs object is displayed on the screen using SWI Toolbox_ShowObject it has the following behaviour:

| Show type | Position | |
|---|---|---|
| 0 (default) | the underlying window is shown at the last place shown on the screen, or the coordinates given in its template, if it has not already been shown | |
| 1 (full spec) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |
| | R3 + 8 | visible area maximum x coordinate |
| | R3 + 12 | visible area maximum y coordinate |
| | R3 + 16 | scroll x offset relative to work area |
| | R3 + 20 | scroll y offset relative to work area |
| | R3 + 24 | Wimp window handle of window to open behind<br>-1 means top of stack<br>-2 means bottom of stack<br>-3 means the window behind the Wimp's backwindow |
| 2 (topleft) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |

### Setting the SaveAs Dialogue box's filename and filetype

When a SaveAs Dialogue object is created, it is given the filename from its template to use in its writable field, and a filetype which will be used to look up and use a sprite (from the Wimp sprite pool) whose name is `file_`*`HHH`*, where *HHH* is a 3-digit hex representation of the filetype. If such a sprite does not exist then a sprite called `file_xxx` is used instead. For saving directories and applications the filetype values 0x1000 and 0x2000 should be used. In the latter case, the standard 'App' sprite is used.

Both of these attributes can be set and read dynamically using the SaveAs_SetFileName/SaveAs_GetFileName and SaveAs_SetFileType/ SaveAs_GetFileType methods.

### Summary of how to save data from a Toolbox client

There are essentially three sorts of application:

- Type 1 – an application which will allow the Toolbox to do data saving entirely on its behalf.
- Type 2 – an application which needs to do the data saving itself, but is not willing to support RAM transfers.
- Type 3 – an application which needs to do the data saving itself, and is willing to support RAM transfers.

Let us look at how a client should react to each Toolbox event which it will receive. Notice that these are the only events which the client needs to watch for to achieve the SaveAs operation; there is no need to watch for user drags and window events, and no need to watch for Message_RAMFetch events. The following is some pseudo-C showing how a client might process Toolbox events delivered to it:

### Type 1

```
switch(toolbox_event_code)
{
    case SaveAs_AboutToBeShown:
        /* call SaveAs_SetFileSize, SaveAs_SetFileName, SaveAs_SetFileType
           and SaveAs_SelectionAvailable if necessary.
           Also call SaveAs_SetDataAddress to tell the Toolbox
           the address and size of data to be saved.
        */
        break;

    case SaveAs_SaveCompleted:
        /* maybe mark a document as 'unmodified' */
        break;

    case SaveAs_DialogueCompleted:
        /* do any tidying up
           maybe delete the SaveAs object if desired
        */
        break;
```

```
    default:
        break;
}
```

### Type 2

```
switch(toolbox_event_code)
{
    case SaveAs_AboutToBeShown:
        /* call SaveAs_SetFileSize, SaveAs_SetFileName, SaveAs_SetFileType
           and SaveAs_SelectionAvailable if necessary
        */
        break;

    case SaveAs_SaveToFile:
        /* save the data to the given filename
           and call SaveAs_FileSaveCompleted
        */
        break;

    case SaveAs_SaveCompleted:
        /* maybe mark a document as 'unmodified' */
        break;

    case SaveAs_DialogueCompleted:
        /* do any tidying up
           maybe delete the SaveAs object if desired
        */
        break;

    default:
        break;
}
```

### Type 3

```
switch(toolbox_event_code)
{
    case SaveAs_AboutToBeShown:
        /* SaveAs_SetFileSize, call SaveAs_SetFileName, SaveAs_SetFileType
           and SaveAs_SelectionAvailable if necessary
        */
        break;

    case SaveAs_SaveToFile:
        /* save the data to the given filename
           and call SaveAs_FileSaveCompleted
        */
        break;
```

```
    case SaveAs_FillBuffer:
        /* if (address of buffer == 0)
               allocate a buffer for RAM transfer
           if (more data to go)
           {
               fill buffer with data
               call SaveAs_BufferFilled
           }
        */
        break;

    case SaveAs_SaveCompleted:
        /* maybe mark a document as 'unmodified' */
        break;

    case SaveAs_DialogueCompleted:
        /* do any tidying up
           maybe delete the SaveAs object if desired
        */
        break;

    default:
        break;
}
```

**Setting the File Size for the SaveAs Dialogue**

In the file transfer protocol under RISC OS, the sender of a file must specify an estimated size in bytes of the file being saved. This should be set using the SaveAs_SetFileSize method, and can be read using the SaveAs_GetFileSize method. This value will be used in the initial Message_DataSave message which will be sent by the SaveAs module when the file icon is dragged to its destination.

**Enabling/disabling the Selection option button**

In the dialogue box used to implement the SaveAs Dialogue object, there is an option button which is used to show whether the Save operation is to be done on the whole file or just a selection. Handling this button is done entirely by the SaveAs module. It is, however, the responsibility of the client to either enable or disable this option button, depending on whether there is a selection currently in existence. This will cause the button to appear greyed out when no selection exists.

The SaveAs module provides the method SaveAs_SelectionAvailable for this use. The client should typically use this method in response to the SaveAs_AboutToBeShown Toolbox event.

**Before the SaveAs Dialogue box is shown**

Once a SaveAs dialogue has been started by using Toolbox_ShowObject on a SaveAs Dialogue object, a **SaveAs** dialogue box will appear on the screen. By setting an appropriate bit in the SaveAs Dialogue object's flags word, the client will be sent a SaveAs_AboutToBeShown Toolbox event before the dialogue box appears. This allows the client to set any relevant state like a different filename, or filetype etc.

**Cancelling the dialogue**

If the user clicks on the **Cancel** button or presses Escape (or clicks outside the SaveAs dialogue box if it was transient), then the SaveAs module delivers a SaveAs_DialogueCompleted Toolbox event to the client application (if enabled). This allows the client to update any of its data structures and to clean up any state associated with this dialogue.

**Saving handled entirely by the SaveAs module**

If the client is able to supply the data to be saved in a contiguous block of memory (i.e. client type 1), then by setting bit 3 in the SaveAs object's flags word, the client can request that the SaveAs module handles the entire Save operation itself. To do this, the client must supply the address of the data (and its size), using the SaveAs_SetDataAddress method. Typically the client will do this when it receives the SaveAs_AboutToBeShown Toolbox event.

The SaveAs module will then conduct the rest of the dialogue. If it receives a Message_RAMFetch message from the receiver, it will do a RAM transfer on behalf of the client; otherwise it will do a scrap transfer (or save directly to file if the destination is a filing system). All of this is transparent to the client if bit 3 is set in the SaveAs object's flags word.

**Saving to a file**

If bit 3 of the SaveAs object's flags word is not set (thus indicating that the Toolbox cannot do a save operation on the client's behalf), then when the SaveAs module wants the application to save to a file, it will deliver a SaveAs_SaveToFile Toolbox event. On receipt of this event, the client (type 2 always and type 3 when necessary) should save its data into the file whose name is given in the event block. The client should then use the SaveAs_FileSaveCompleted method to inform the SaveAs module whether the Save was successful or not. This **must** be done before the next call to SWI Wimp_Poll, since the SaveAs module will assume this.

The SaveAs_SaveToFile event will be delivered if

- the user clicks on **Save**
- a Wimp$Scrap transfer is being used
- the user has dragged the file icon onto a directory display.

### Saving via RAM transfer

If bit 3 of the SaveAs object's flags word is not set (thus indicating that the Toolbox cannot do a save operation on the client's behalf), then the client (type 3 only) may wish to help support RAM transfers if they are requested by the receiving task. This is indicated by setting bit 4 of the SaveAs object's flags word.

The client must supply a buffer, into which it places data ready for transmission to the receiving task.

The SaveAs module will deal with all subsequent RAMFetch requests, and will call SWI Wimp_TransferBlock to do the data transfer, and will reply to the receiver using Message_RAMTransmit.

The client will receive SaveAs_FillBuffer Toolbox events when the buffer has been transmitted, and on receipt of such events should fill the buffer and call the SaveAs_BufferFilled method. If the field in the SaveAs_FillBuffer event giving the address of the buffer is 0, then the client has not yet supplied a buffer, and they should allocate one. Each SaveAs_FillBuffer Toolbox event contains an indication of how many bytes have been transmitted so far during the transfer. As soon as the number of bytes which the client writes into the buffer is less than the size of the buffer, the SaveAs module assumes that the transfer is complete.

### Successful completion of a Save operation

When a Save operation has been successfully completed (i.e. the data has been saved), the SaveAs module will send a SaveAs_SaveCompleted Toolbox event to the client, and will hide the SaveAs object, unless the user has clicked Adjust on the **Save** button.

One field in the event block passed back to the client is a one-word indication of whether the destination was a 'safe' place (like a filing system) or 'unsafe' (like another application). The client may choose to use this value to decide whether to mark the data as 'un-modified', if the client is an editor.

If the original save operation was started by the user dragging the file icon from the SaveAs dialogue box, then the SaveAs_SaveCompleted event block also contains the Wimp message reference number of the Message_DataSave sent by the SaveAs module, to allow the client to use in conjunction with any Message_DataSaved replies.

### Completion of the SaveAs dialogue

When the SaveAs module has hidden its dialogue box at the end of a dialogue, it delivers a SaveAs_DialogueCompleted Toolbox event to the client, with an indication of whether a successful save occurred during the dialogue.

**Error handling**

Any errors referring to the SaveAs dialogue box itself will be reported to the user by the SaveAs module. For example, if there is only a leafname in the writable field, and the user clicks on **Save**, then the SaveAs module will display an error box saying 'To save, drag the icon to a directory display'.

The SaveAs module will also report any errors which occur while it is carrying out a Save operation.

The client should report (via SWI Wimp_ReportError), any errors which occur if it is requested to save to a given filename.

## Save As methods

The following methods are all invoked by calling SWI Toolbox_ObjectMiscOp with:

| | |
|---|---|
| R0 | holding a flags word |
| R1 | being a Save As Dialogue object id |
| R2 | being the method code which distinguishes this method |
| R3-R9 | potentially holding method-specific data |

### SaveAs_GetWindowID 0

**On entry**

R0 = flags
R1 = Save As object id
R2 = 0

**On exit**

R0 = Window object id for this Save As object

**Use**

This method returns the id of the underlying Window object used to implement this Save As object.

**C veneer**

```
extern _kernel_oserror *saveas_get_window_id ( unsigned int flags,
                                               ObjectId saveas,
                                               ObjectId *window
                                             );
```

## SaveAs_SetTitle 1

**On entry**

R0 = flags
R1 = Save As object id
R2 = 1
R3 = pointer to text string to use

**On exit**

R1-R9 preserved

**Use**

This method sets the text which is to be used in the title bar of the given Save As dialogue.

**C veneer**

```
extern _kernel_oserror *saveas_set_title ( unsigned int flags,
                                           ObjectId saveas,
                                           char *title
                                         );
```

## SaveAs_GetTitle 2

**On entry**

R0 = flags
R1 = Save As object id
R2 = 2
R3 = pointer to buffer to return the text in (or 0)
R4 = size of buffer

**On exit**

R4 = size of buffer required to hold the text (if R3 was 0)
else Buffer pointed to by R3 contains title text
R4 holds number of bytes written to buffer

**Use**

This method returns the text string used in a Save As dialogue's title bar.

**C veneer**

```
extern _kernel_oserror *saveas_get_title ( unsigned int flags,
                                           ObjectId saveas,
                                           char *buffer,
                                           int buff_size,
                                           int *nbytes
                                         );
```

## SaveAs_SetFileName 3

**On entry**

R0 = flags
R1 = Save As object id
R2 = 3
R3 = pointer to filename to use in writable field

**On exit**

R1-R9 preserved

**Use**

This method sets the filename which is to be used in the Save As object's writable field.

**C veneer**

```
extern _kernel_oserror *saveas_set_file_name ( unsigned int flags,
                                               ObjectId saveas,
                                               char *file_name
                                             );
```

## SaveAs_GetFileName 4

**On entry**

R0 = flags
R1 = Save As object id
R2 = 4
R3 = pointer to buffer to return the filename in (or 0) R4 =size of buffer

**On exit**

R4 = size of buffer required to hold the filename (if R3 was 0)
else Buffer pointed to by R3 contains filename
R4 holds number of bytes written to buffer

**Use**

This method returns the filename displayed in this Save As object's writable field.

**C veneer**

```
extern _kernel_oserror *saveas_get_file_name ( unsigned int flags,
                                               ObjectId saveas,
                                               char *buffer,
                                               int buff_size,
                                               int *nbytes
                                             );
```

## SaveAs_SetFileType 5

**On entry**

R0 = flags
R1 = Save As object id
R2 = 5
R3 = filetype

**On exit**

R1-R9 preserved

**Use**

This method is used to set the filetype for this Save As object, and hence the sprite which will be displayed in the dialogue box.

**C veneer**

```
extern _kernel_oserror *saveas_set_file_type ( unsigned int flags,
                                               ObjectId saveas,
                                               int file_type
                                             );
```

## SaveAs_GetFileType 6

**On entry**

R0 = flags
R1 = Save As object id
R2 = 6

**On exit**

R0 = filetype

**Use**

This method is used to get the filetype of this Save As object.

**C veneer**

```
extern _kernel_oserror *saveas_get_file_type ( unsigned int flags,
                                               ObjectId saveas,
                                               int *file_type
                                             );
```

## SaveAs_SetFileSize 7

**On entry**

R0 = flags
R1 = Save As object id
R2 = 7
R3 = file size in bytes

**On exit**

R1-R9 preserved

**Use**

This method is used to set the estimated file size in bytes for this Save As Dialogue. This will be used in a Message_DataSave message when the file icon is dragged to its destination.

**C veneer**

```
extern _kernel_oserror *saveas_set_file_size ( unsigned int flags,
                                               ObjectId saveas,
                                               int file_size
                                             );
```

## SaveAs_GetFileSize 8

**On entry**

R0 = flags
R1 = Save As object id
R2 = 8

**On exit**

R0 = file size

**Use**

This method is used to get the file size of this Save As object.

**C veneer**

```
extern _kernel_oserror *saveas_get_file_size ( unsigned int flags,
                                               ObjectId saveas,
                                               int *file_size
                                             );
```

## SaveAs_SelectionAvailable 9

**On entry**

R0 = flags
R1 = Save As object id
R2 = 9
R3 = non-zero means selection is available, otherwise it is not available

**On exit**

R1-R9 preserved

**Use**

This method is used to indicate to the Save As module whether there is a current selection in existence. If there is a selection, then the **Selection** option button will be enabled (i.e. the user can click on it), if not the **Selection** option button will be greyed out.

If the Save As object has no **Selection** option button then an error is returned.

**C veneer**

```
extern _kernel_oserror *saveas_selection_available ( unsigned int flags,
                                                     ObjectId saveas,
                                                     int selection
                                                   );
```

## SaveAs_SetDataAddress 10

**On entry**

R0 = flags
R1 = Save As object id
R2 = 10
R3 = address of contiguous block of data which is to be saved
R4 = size of data
R5 = address of contiguous block of data, which is the current selection
R6 = size of selection

**On exit**

R1-R9 preserved

**Use**

This method indicates to the Save As module the address of a contiguous block of memory containing the data to be saved. It is used if the client wishes the entire Save operation to be carried out by the Save As module. It is typically called in response to a SaveAs_SaveAboutToBeShown Toolbox event. If there is a current selection, then its address and size should also be passed to this method.

Note: This method is only suitable for Type 1 clients.

**C veneer**

```
extern _kernel_oserror *saveas_set_data_address ( unsigned int flags,
                                                  ObjectId saveas,
                                                  void *data,
                                                  int data_size,
                                                  void *selection,
                                                  int selection_size
                                                );
```

## SaveAs_BufferFilled 11

**On entry**

R0 = flags
R1 = Save As object id
R2 = 11
R3 = address of buffer which has been filled
R4 = number of bytes written into buffer

**On exit**

R1-R9 preserved

**Use**

This method is used to respond to a SaveAs_FillBuffer Toolbox event; it confirms that the requested buffer fill has taken place, and states the number of bytes written to the buffer.

**C veneer**

```
extern _kernel_oserror *saveas_buffer_filled ( unsigned int flags,
                                               ObjectId saveas,
                                               void *buffer,
                                               int bytes_written
                                             );
```

## SaveAs_FileSaveCompleted 12

**On entry**

R0 = flags bit 0 set means that the save was successful
R1 = Save As object id
R2 = 12
R3 = filename where the client tried to save the data

**On exit**

R1-R9 preserved

**Use**

This method is used by the client to report whether an attempt to save the data to file as a result of a SaveAs_SaveToFile Toolbox event was successful or not.

If this SWI is called with bit 0 of R0 clear, then it will return an error.

Note: This method is only suitable for Type 2 and Type 3 clients.

**C veneer**

```
extern _kernel_oserror *saveas_file_save_completed ( unsigned int flags,
                                                     ObjectId saveas,
                                                     char *filename
                                                   );
```

## Save As events

The Save As module generates the following Toolbox events:

### SaveAs_AboutToBeShown (0x82bc0)

**Block**

| | |
|---|---|
| + 8 | 0x82bc0 |
| + 12 | flags (as passed in to Toolbox_ShowObject) |
| + 16 | value which will be passed in R2 to ToolBox_ShowObject |
| + 20... | block which will be passed in R3 to ToolBox_ShowObject for the underlying dialogue box |

**Use**

This Toolbox event is raised just before the Save As module is going to show its underlying Window object, to enable the client to set its filename and filetype appropriately.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                show_type;
  union
  {
    TopLeft               pos;
    WindowShowObjectBlock full;
  } info;
} SaveAsAboutToBeShownEvent;
```

## SaveAs_DialogueCompleted (0x82bc1)

**Block**

+ 8 0x82bc1
+ 12 flags
bit 0 set means that a successful save was done during this dialogue

**Use**

This Toolbox event is raised after the Save As object has been hidden, either by a Cancel click, or after a successful save, or by the user clicking outside the dialogue box or pressing Escape. It allows the client to tidy up its own state associated with this dialogue.

Note that if the dialogue was cancelled, a successful save may still have been done, for example if the user clicked Adjust on Save, and then cancelled the dialogue.

**C data type**

```
typedef struct
{
  ToolboxEventHeader  hdr;
} SaveAsDialogueCompletedEvent;
```

## SaveAs_SaveToFile (0x82bc2)

**Block**

+ 8 0x82bc2
+ 12 flags bit 0 set means save only the current selection
+ 16... nul-terminated filename to which the data should be saved

**Use**

This Toolbox event is raised by the Save As module to request that the client should save its data to the given filename. If bit 0 of the flags word is set, then only the current selection should be saved.

**C data type**

```
typedef struct
{
  ToolboxEventHeader  hdr;
  char                filename [212];
} SaveAsSaveToFileEvent;
```

## SaveAs_FillBuffer (0x82bc3)

**Block**

| | |
|---|---|
| + 8 | 0x82bc3 |
| + 12 | flags<br>bit 0 set means a selection is being saved |
| + 16 | size of buffer being used |
| + 20 | address of buffer |
| + 24 | number of bytes already transmitted |

**Use**

This Toolbox event is raised by the Save As module to request that the client should fill the given buffer (which is the one which the client will have allocated).

If the address returned by this event is 0, then the client application needs to do one of the following:

- reserve memory for buffering and return its address using SWI BufferFilled
- maintain a pointer to the current location in the data to be transferred.

**C data type**

```
typedef struct
{
  ToolboxEventHeader  hdr;
  int                 size;
  char                *address;
  int                 no_bytes;
} SaveAsFillBufferEvent;
```

### SaveAs_SaveCompleted (0x82bc4)

**Block**

| | |
|---|---|
| + 8 | 0x82bc4 |
| + 12 | flags<br>bit 0 set means a selection was saved<br>bit 1 set means the destination was safe (e.g. a filing system) |
| + 16 | Wimp message number of original Message_DataSave<br>(or 0 if the save operation was not started via a drag) |
| + 20... | if bit 1 is set in the flags word (i.e. safe save), then this field indicates the full pathname of the place where the save was done. |

**Use**

This Toolbox event is raised when the Save is successfully completed. Bit 0 of the flags word indicates whether just a selection was saved; bit 1 means that the Save was to a place where the data is safe (e.g. it is in a real file, on a filing system).

**C data type**

```
typedef struct
{
  ToolboxEventHeader  hdr;
  int                 wimp_message_no;
  char                filename [208];
} SaveAsSaveCompletedEvent;
```

## Save As templates

The layout of a Save As template is shown below. Fields which have types MsgReference and StringReference are those which will require relocation when they are loaded from a resource file. If the template is being constructed in memory, then these fields should be real pointers (i.e. they do not require relocation).

For more details on relocation, see appendix *Resource File Formats* on page 499.

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| filename | 4 | MsgReference |
| filetype | 4 | word |
| title | 4 | MsgReference |
| max_title | 4 | word |
| window | 4 | StringReference |

### Underlying Window template

The Window object used to implement a Save As dialogue, has the following characteristics. These must be reproduced if the Window is replaced by a client-specified alternative Window template.

Title bar must be indirected.

**Gadgets**

Component ids are derived by adding to 0x82bc00.

| Component id | Details | |
|---|---|---|
| 0 | draggable (file icon) | must be sprite only |
| 1 | writable field (filename) | |
| 2 | action button (Cancel) | must be marked as a Cancel action button |
| 3 | action button (Save) | must be marked as the Default action button |
| 4 (if required) | option button (Selection) | |

## Save As Wimp event handling

| Wimp event | Action |
|---|---|
| Mouse Click | if this is a drag event on the file icon, then set up an appropriate Wimp drag box |
| ActionButton_Selected | on the **Save** button then start save operation |
| | on the **Cancel** button then hide the dialogue box, and raise a SaveAs_DialogueCompleted Toolbox event |
| Draggable_DragEnded (Toolbox event) | start save operation to the destination of the drag (i.e. send a Message_DataSave to the destination window/icon pair. |
| Key Pressed | if dialogue box has the input focus, and the key pressed is Return, then the **Save** Button is activated, and a save operation is started |
| | if key is Escape act as if Cancel had been pressed. |

| Wimp event | Action |
|---|---|
| User Message | Message_DataSaveAck |
| User Message Recorded | |

```
if (a SaveAs dialogue is in progress)
{
    if (the save can be done entirely
        by the SaveAs module)
    {
        do the save
        send Message_DataLoad to destination
    }
    else
    {
        raise a SaveAs_SaveToFile Toolbox event
    }
}
```

Message_DataLoadAck

```
if (a SaveAs dialogue is in progress)
{
    raise a SaveAs_SaveCompleted Toolbox event
    If (not an Adjust click on OK)
    (
        hide the dialogue box
        raise a SaveAs_DialogueCompleted
        Toolbox event
    )
}
```

Message_RAMFetch

```
if (a SaveAs dialogue is in progress)
{
    transfer current buffer contents
    send Message_RAMTransmit to destination
   if (save cannot be done entirely by the Toolbox
        module)
        raise SaveAs_FillBuffer Toolbox event
}
```

Message_MenusDeleted

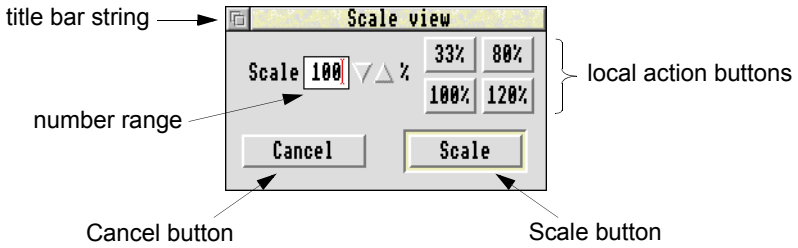
```
If (a SaveAs dialogue is in progress)
{
   raise a SaveAs_DialogueCompleted Toolbox event
}
```

# 15 Scale Dialogue box class

A Scale Dialogue object is used to present the user with a dialogue box from which he can set the scale factors for a view on a document. This scale is given as a percentage of the original size of the document.

## User interface

The Scale class provides a dialogue box from which a scale factor can be chosen:




The default Scale dialogue box has the following attributes:

- a title bar string
- a writable number range with up/down arrows and a percentage sign to the right of the up/down arrows
- four ‘standard’ size action buttons with the values: 33%, 80%, 100%, 120% as their text plus an optional **Scale to Fit** action button
- a Cancel action button
- a Scale action button.

The user can:

- type an integral value in the writable field between its lower and upper bounds or use the up/down arrows to adjust the value currently in the field
- use one of the standard size action buttons to set the scale factor. Clicking on these buttons only causes a value to be inserted in the writable field; it does not apply the scale factors
- click outside the dialogue box (if it is transient) or click on **Cancel**, to cancel the dialogue

- click on **Scale** or press Return to apply the scale factors
- if there is a **Scale to Fit** button, then clicking on it will have application-defined behaviour (e.g. Scale to Fit window).

## Application Program Interface

When a Scale object is created it has the following components:

- an optional **Scale To Fit** button.
- an alternative title to use instead of the default.
- alternative bounds and step size for the writable field.
- an optional list of different standard size action buttons where each gives a percentage value to insert into the Writable Field. These will be positioned appropriately by the Scale module in place of the default standard size buttons. When a Scale object is shown, the client will be delivered a Scale_DialogueAboutToBeShown Toolbox event (if enabled), just before the dialogue box becomes visible on the screen.

When the Scale dialogue is showing, the Scale module deals with all relevant Wimp events and reports user actions back to the client via Toolbox events. If there are any standard size action buttons in the dialogue box, then the Scale module deals with clicks on them, and inserts the correct percentage value into the writable field.

The client is guaranteed to receive a Scale_DialogueCompleted Toolbox event when the dialogue is over (i.e. the user has clicked on Cancel, or clicked outside the dialogue box (if it were transient), or clicked on Scale, or on Scale To Fit).

### Attributes

A Scale object has the following attributes which are specified in its object template and can be manipulated at run-time by the client application:

| **Attributes** | **Description** | |
|---|---|---|
| flags | **Bit** | **Meaning** |
| | 0 | when set, this bit indicates that a Scale_AboutToBeShown event should be raised when SWI Toolbox_ShowObject is called for this object. |
| | 1 | when set, this bit indicates that a Scale_DialogueCompleted event should be raised when the Scale object has been removed from the screen. |
| | 2 | when set, dialogue box has a **Scale To Fit** button |

| Attributes | Description |
|---|---|
| min val | alternative minimum value for the writable field |
| max val | alternative maximum value for the writable field |
| step size | alternative step size for up/down arrows |
| Scale title | alternative title for the dialogue rather than ‘Scale View’ (0 means use default) |
| max title length | this gives the maximum length in bytes of title text which will be used for this object |
| window | the name of an alternative window template to use instead of the default one (0 means use default) |
| std1 value | value of first std scale button |
| std2 value | value of second std scale button |
| std3 value | value of third std scale button |
| std4 value | value of fourth std scale button |

## Manipulating a Scale object

### Creating and deleting a Scale object

A Scale object is created using SWI Toolbox_CreateObject.

When this object is created it has no attached objects (see page 11).

A Scale object is deleted using SWI Toolbox_DeleteObject.

The setting of the non-recursive delete bit does not have a meaning for Scale objects.

### Showing a Scale object

When a Scale object is displayed on the screen using SWI Toolbox_ShowObject it has the following behaviour:

| Show type | Position |
|---|---|
| 0 (default) | the underlying window is shown at the last place shown on the screen, or the coordinates given in its template, if it has not already been shown |
| 1 (full spec) | R3 + 0 visible area minimum x coordinate<br>R3 + 4 visible area minimum y coordinate<br>R3 + 8 visible area maximum x coordinate<br>R3 + 12 visible area maximum y coordinate<br>R3 + 16 scroll x offset relative to work area<br>R3 + 20 scroll y offset relative to work area<br>R3 + 24 Wimp window handle of window to open behind<br>-1 means top of stack<br>-2 means bottom of stack<br>-3 means the window behind the Wimp's backwindow |
| 2 (topleft) | R3 + 0 visible area minimum x coordinate<br>R3 + 4 visible area minimum y coordinate |

### Before the Scale Dialogue box is shown

When SWI Toolbox_ShowObject is called on a Scale object, the Scale Class raises a Scale_AboutToBeShown Toolbox event (if enabled), just before it shows the underlying Window object which implements this dialogue. This will allow the client to set an initial suitable value in the Scale dialogue's Writable Field.

### Applying a Scale factor

When the user clicks on the **Scale** button, or on the **Scale To Fit** button if it is present, the Scale module delivers a Scale_ApplyFactor to the client, giving the percentage factor to apply. A special value of 0xffffffff is delivered if the **Scale To Fit** button is clicked.

### Cancelling a Scale dialogue

If the user clicks on the **Cancel** Button (or clicks outside the Scale dialogue box), then the Scale module delivers a Scale_DialogueCompleted Toolbox event to the client application. This allows the client to update any of its data structures and to clean up any state associated with this dialogue.

### Completion of a Scale dialogue

When the Scale module has hidden its dialogue box at the end of a dialogue, it delivers a Scale_DialogueCompleted Toolbox event to the client (if enabled), with an indication of whether a scale factor was reported to the client during the dialogue.

### Reading and setting the writable field

Normally a client will only need to respond to the Scale_ApplyFactor Toolbox event in order to allow the user to set scale factors. If, however, the client wishes to read the current value in the writable field, or to set it explicitly (to a suitable start value when the dialogue box is first shown), then it can use the Scale_SetValue/Scale_GetValue methods.

### Reading and setting the bounds of the writable field and step size

Normally a client will specify the bounds and step size of the writable field in the template description for the Scale object.

These can however be read and set dynamically using the Scale_SetBounds/ Scale_getBounds and Scale_GetStepSize/Scale_SetStepSize methods.

## Scale methods

The following methods are all invoked by calling SWI Toolbox_ObjectMiscOp with:

| | |
|---|---|
| R0 | holding a flags word |
| R1 | being a Scale Dialogue object id |
| R2 | being the method code which distinguishes this method |
| R3-R9 | potentially holding method-specific data |

### Scale_GetWindowID 0

**On entry**

R0 = flags
R1 = Scale object id
R2 = 0

**On exit**

R0 = Window object id for this Scale object

**Use**

This method returns the id of the underlying Window object used to implement this Scale object.

**C veneer**

```
extern _kernel_oserror *scale_get_window_id ( unsigned int flags,
                                              ObjectId scale,
                                              ObjectId *window
                                            );
```

## Scale_SetValue 1

**On entry**

R0 = flags
R1 = Scale object id
R2 = 1
R3 = value

**On exit**

R1-R9 preserved

**Use**

This method is used to set the value displayed in the writable field for this Scale object.

**C veneer**

```
extern _kernel_oserror *scale_set_value ( unsigned int flags,
                                          ObjectId scale,
                                          int value
                                        );
```

## Scale_GetValue 2

**On entry**

R0 = flags
R1 = Scale object id
R2 = 2

**On exit**

R0 = value

**Use**

This method returns the value in the writable field of this Scale object.

**C veneer**

```
extern _kernel_oserror *scale_get_value ( unsigned int flags,
                                          ObjectId scale,
                                          int *value
                                        );
```

## Scale_SetBounds 3

**On entry**

R0 = flags
    bit 0 set means set the lower bound to the given value
    bit 1 set means set the upper bound to the given value
    bit 2 set means set step size
R1 = Scale object id
R2 = 3
R3 = value of the lower bound
R4 = value of the upper bound
R5 = step size

**On exit**

R1-R9 preserved

**Use**

This method sets the lower and upper bounds and step size of the writable field in the Scale object.

**C veneer**

```
extern _kernel_oserror *scale_set_bounds ( unsigned int flags,
                                           ObjectId scale,
                                           int lower_bound,
                                           int upper_bound,
                                           int step_size
                                         );
```

## Scale_GetBounds 4

**On entry**

R0 = flags
bit 0 set means return the lower bound
bit 1 set means return the upper bound
bit 2 set means return step size
R1 = Scale object id
R2 = 4

**On exit**

R0 = value of the lower bound
R1 = value of the upper bound
R2 = value of the step size

**Use**

This method returns either the lower and upper bounds and step size of the writable field in the Scale object.

**C veneer**

```
extern _kernel_oserror *scale_get_bounds ( unsigned int flags,
                                           ObjectId scale,
                                           int *lower_bound,
                                           int *upper_bound,
                                           int *step_size
                                         );
```

## Scale_SetTitle 5

**On entry**

R0 = flags
R1 = Scale object id
R2 = 5
R3 = pointer to text string to use

**On exit**

R1-R9 preserved

**Use**

This method sets the text which is to be used in the title bar of the given Scale dialogue.

**C veneer**

```
extern _kernel_oserror *scale_set_title ( unsigned int flags,
                                          ObjectId scale,
                                          char *title
                                        );
```

## Scale_GetTitle 6

**On entry**

R0 = flags
R1 = Scale object id
R2 = 6
R3 = pointer to buffer to return the text in (or 0)
R4 = size of buffer

**On exit**

R4 = size of buffer required to hold the text (if R3 was 0)
else Buffer pointed to by R3 contains title text
R4 holds number of bytes written to buffer

**Use**

This method returns the text string used in a Scale dialogue's title bar.

**C veneer**

```
extern _kernel_oserror *scale_get_title ( unsigned int flags,
                                          ObjectId scale,
                                          char *buffer,
                                          int buff_size,
                                          int *nbytes
                                        );
```

## Scale events

The Scale module generates the following Toolbox events:

### Scale_AboutToBeShown (0x82c00)

**Block**

| | |
|---|---|
| + 8 | 0x82c00 |
| + 12 | flags (as passed in to Toolbox_ShowObject) |
| + 16 | value which will be passed in R2 to ToolBox_ShowObject |
| + 20... | block which will be passed in R3 to ToolBox_ShowObject for the underlying dialogue box |

**Use**

This Toolbox event is raised just before the Scale module is going to show its underlying Window object, to enable the client to set its initial value appropriately.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                show_type;
  union
  {
    TopLeft               pos;
    WindowShowObjectBlock full;
  } info;
} ScaleAboutToBeShownEvent;
```

## Scale_DialogueCompleted (0x82c01)

**Block**

+ 8 0x82c01
+ 12 flags

**Use**

This Toolbox event is raised after the Scale object has been hidden, either by a Cancel click, or by a click on **Scale** or **Scale To Fit**, or by the user clicking outside the dialogue box (if it is transient). It allows the client to tidy up its own state associated with this dialogue.

Note that if the dialogue was cancelled, a scale factor may still have been applied, for example if the user clicked Adjust on **Scale**, and then cancelled the dialogue.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
} ScaleDialogueCompletedEvent;
```

## Scale_ApplyFactor (0x82c02)

**Block**

+ 8 0x82c02
+ 16 unsigned integer scale factor to apply

**Use**

This Toolbox event is raised when the user clicks on the **Scale** button or the **Scale To Fit** button (if present), or presses Return.

The scale factor to apply is a percentage; 0xffffffff means Scale To Fit.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  unsigned int       factor;
} ScaleApplyFactorEvent;
```

## Scale templates

The layout of a Scale template is shown below. Fields which have types MsgReference and StringReference are those which will require relocation when they are loaded from a resource file. If the template is being constructed in memory, then these fields should be real pointers (i.e. they do not require relocation).

For more details on relocation, see appendix *Resource File Formats* on page 499.

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| min_val | 4 | word |
| max_val | 4 | word |
| step_size | 4 | word |
| title | 4 | MsgReference |
| max_title | 4 | word |
| window | 4 | StringReference |
| std1_value | 4 | word |
| std2_value | 4 | word |
| std3_value | 4 | word |
| std4_value | 4 | word |

### Underlying window template

The Window object used to implement a Scale dialogue, has the following characteristics. These must be reproduced if the Window is replaced by a client-specified alternative Window template:

Title bar must be indirected.

#### Gadgets

Component ids are derived by adding to 0x82c000.

| Component id | Details | |
|---|---|---|
| 0 | number range (Scale) | must have adjuster arrows, and be writable |
| 1-4 | action buttons (standard scale factors) | these should have the text 33%, 80%, 100% and 120% |
| 5 | action button (Cancel) | this must be marked as a Cancel action button |
| 6 | action button (Scale) | this must be marked as the default action button |

| Component id | Details |
|---|---|
| 7 | label (%) |
| 8 | label (Scale) |
| 9 | action button (Scale to fit) |

## Scale Wimp event handling

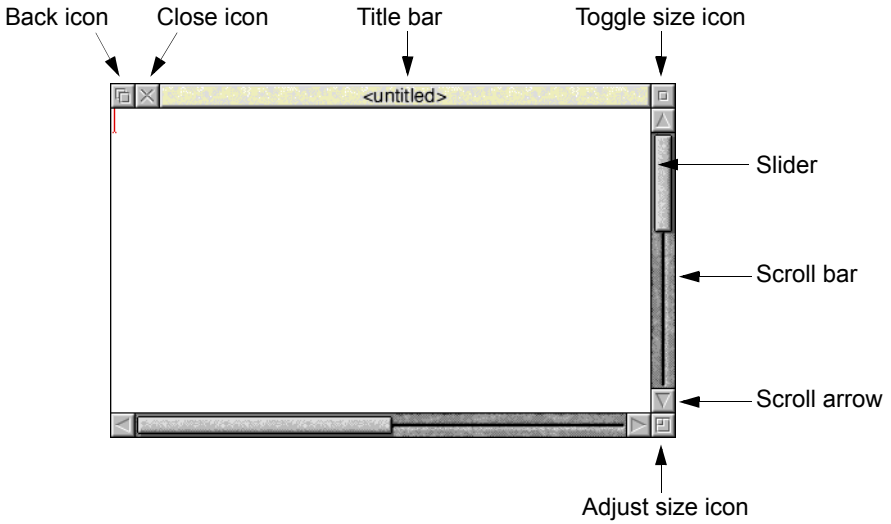
| Wimp event | Action |
|---|---|
| Mouse Click | on **Scale** or **Scale to Fit** buttons, then deliver a Scale_ApplyFactor Toolbox event |
| | on a standard size button then enter its value into the Writable Field |
| | on **Cancel** button then hide the dialogue box, and deliver a Scale_DialogueCompleted Toolbox event. |
| Key Pressed | if key is Return then act as if **Scale** button had been clicked |
| | if key is Escape then act as if **Cancel** button had been clicked. |
| User Message | Message_MenusDeleted |
| User Message Recorded | deliver a Scale_DialogueCompleted Toolbox event. |

# 16 Window class

Objects of the Window class are used by the client application to display its document windows, dialogue boxes etc.

## User interface

A Window is essentially an extension of a Wimp window (in fact part of the Window object definition is a Wimp window definition):




Many Wimp events which are delivered to this Window are dealt with automatically by the Toolbox, based on the attributes of the Window. In this chapter we give further details of exactly what a Window consists of, and the semantics attached to Wimp events for a Window.

The client application is always able to get the Wimp window handle of the underlying Wimp window used to implement this Window object, and can perform all the usual Wimp SWIs on that window (within reason, e.g. deleting an icon belonging to a gadget may have undesirable efects).

# Application Program Interface

## Attributes

A Window object has the following attributes which are specified in its object template and can be manipulated at run-time by the client application:

| Attribute | Description |
|---|---|
| flags word | **Bit** **Meaning**<br>0 when set, generate a Window_AboutToBeShown event before showing the underlying Wimp window<br>1 when set, automatically open this Window when a Wimp OpenWindowRequest is received (when set the client will not see the underlying Wimp requests)<br>2 when set, automatically close this Window when a Wimp CloseWindowRequest is received (when set the client will not see the underlying Wimp requests)<br>3 when set, generate a Window_HasBeenHidden Event after hiding the underlying Wimp window<br>4 when set, indicates that this template is of a toolbar (see *Toolbars* on page 322) |
| help message | when a HelpRequest is received for this Window, then this text is sent in a HelpReply message. Note that this Help message is only sent if the gadget (see later) for which the request was received has not got a Help message of its own, or if the pointer is not over any gadget. |
| max help | maximum length in bytes of help message |
| pointer shape | this gives the name of a sprite to use as the pointer shape, when a Pointer Entering Window event is received for this Window (0 means do not change the pointer shape). |
| max pointer shape | maximum length in bytes of sprite name |
| pointer x hot<br>pointer y hot | the x and y coordinates of the pointer's hot spot. These are relative pixels from the top left corner of the sprite. |

| Attribute | Description |
|---|---|
| menu | the name of the template to use to create a Menu object for this Window |
| num keyboard shortcuts | the number of keyboard short-cuts which are associated with this Window |
| keyboard shortcuts | the pointer to the list of keyboard short-cuts for this Window |
| num gadgets | the number of gadgets which are to appear in this Window |
| gadgets | the pointer to the list of gadgets for this Window. |
| default focus | the Component Id of the gadget which is given input focus when the window is shown.<br>If this field is -1 then no gadget will be given input focus<br>if -2 then window will be given input focus (but no caret) allowing keyboard short-cuts to work without having any writables |
| window | 88-byte structure is the standard block which is passed to Wimp_CreateWindow. The window is shown to contain no icons, since these are implemented by gadgets. |
| internal_bl | the window template to be used for this toolbar. Anchored to the bottom left corner inside the window. † |
| internal_tl | the window template to be used for this toolbar. Anchored to the top left corner inside the window. † |
| external_bl | the window template to be used for this toolbar. Anchored to the bottom left corner outside the window. † |
| external_tl | the window template to be used for this toolbar. Anchored to the top left corner outside the window. † |
| show_event | the event code to be raised when the window is shown. |
| hide_event | the event code to be raised after the window has been hidden. |

† these templates must have the Toolbar bit set.

**Keyboard short-cut**

The attributes of a Keyboard short-cut are as follows:

| Attributes | Description |
|---|---|
| flags word | **Bit** **Meaning**<br>0 when set, show attached object as ‘transient’ |
| wimp key code | the key code returned by the Wimp in a Key Pressed event block, for this keyboard short-cut |
| key event | this is the Toolbox event to be raised when the Wimp delivers a Key Pressed event with this Wimp key code.<br>0 means deliver no event |
| key show | the name of the template for an object to create and show when the Wimp delivers a Key Pressed event with this Wimp key code.<br>0 means show no object |

Note that because keyboard short-cuts work on Wimp key codes, certain key combinations (such as Shift-Ctrl-P) will require the client to provide extra code.

**Gadget**

All gadgets have a common header, followed immediately by a body which is gadget-specific. The header is described on page 326, and the gadget-specific bodies are described in their own sections.

## Manipulating a Window object

**Creating and deleting a Window object**

A Window object is created using SWI Toolbox_CreateObject.

When a Window object is created, the following attached objects (see page 11) will be created (if specified):

- menu
- key show (for each keyboard short-cut)
- Toolbars.

See the attributes table above for an explanation of what these objects are.

There are also attached objects which are associated with gadgets in a Window (see later):

- click show (for an action button)
- menu (for a Pop-up menu).

These attached objects are also created when such a gadget is added to the Window, and deleted when the gadget is removed.

A Window object is deleted using SWI Toolbox_DeleteObject. If it has any attached objects (see above), these are also deleted, unless the non-recursive bit is set for this SWI.

**Showing a Window**

When a Window object is displayed on the screen using SWI Toolbox_ShowObject it has the following behaviour:

| Show type | Position | |
|---|---|---|
| 0 (default) | the underlying window is shown at the last place shown on the screen, or the coordinates given in its template, if it has not already been shown | |
| 1 (full spec) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |
| | R3 + 8 | visible area maximum x coordinate |
| | R3 + 12 | visible area maximum y coordinate |
| | R3 + 16 | scroll x offset relative to work area |
| | R3 + 20 | scroll y offset relative to work area |
| | R3 + 24 | Wimp window handle of window to open behind |
| | | -1 means top of stack |
| | | -2 means bottom of stack |
| | | -3 means the window behind the Wimp's backwindow |
| 2 (topleft) | R3 + 0 | visible area minimum x coordinate |
| | R3 + 4 | visible area minimum y coordinate |

**The Window's menu**

Each Window object can optionally have attached to it a Menu object. The Window object holds the unique id of this Menu object.

When a Window is created, if the client has specified the name of a Menu template for that Window, then a Menu object is created from that template, and the id of that Menu is held in the Window object. This id will be used to show the Menu when the user presses the Menu button over the Window.

Whenever the user of the application presses the Menu mouse button over a Window, the Window class module opens its attached Menu object, by making a SWI Toolbox_ShowObject passing the attached Menu's id.

If the application wishes to perform some operations on the Menu before it is opened (ticking some entries for example), then by setting the appropriate bit in the Menu's flags word, the application can request that a special Toolbox event (Menu_AboutToBeShown) is delivered to it before the Menu is actually shown. The precise details of this Toolbox event are described in *Menu events* on page 197. On receipt of such a Toolbox event, the client application is expected to make any changes it wants to the Menu object, and then return to its SWI Wimp_Poll loop.

In most cases a Menu is attached to the Window at resource editing time by entering the name of the template to use for this Window's Menu. If the application wishes dynamically to attach and detach the menu for a given Window (maybe based on a mode of operation which is defined by the application, e.g. display mode or editing mode), then this can be done using the Window_SetMenu method described on page 298.

The id of the Menu attached to a Window can be read by using the Window_GetMenu method.

Window_SetMenu can only be used when a menu is not already being shown for this Window.

**Gadgets in a window**

A Window object can optionally contain a number of gadgets. Typically this is used to create dialogue boxes.

There are many kinds of gadget. The Toolbox provides facilities to allow the client application to manipulate a particular gadget in a manner which is appropriate to that gadget, rather than in 'low-level' terms like setting the state of a Wimp icon. The set of gadgets is defined to fit in with the RISC OS 3 Style Guide, and thus to encourage a standard look and feel across dialogue boxes.

Gadgets are normally specified as part of a Window object template, but they can be added to and removed from Window's dynamically at run-time using the Window_AddGadget and Window_RemoveGadget methods respectively.

Each gadget type defines its own set of methods, and many will have a number of Toolbox events associated with them. This allows the application to receive Toolbox events from user actions, rather than having to deal with mouse clicks and drags on Wimp icons. Much of the low-level Wimp operations are handled automatically by the Toolbox.

Gadgets are described in *Gadgets* on page 325.

### Keyboard short-cuts

Each Window object can optionally define a set of mappings from Wimp key codes to Toolbox events. This is particularly useful in allowing the client application to respond identically to a keystroke or an equivalent menu hit, by giving both the same Toolbox event. When a given keystroke is returned by the Wimp for the Window object, the corresponding Toolbox event is raised.

Note that Shift-Ctrl-letter combinations are not allowed.

It is also possible to provide the name of a template for an object which will be created and shown, when a particular keystroke happens. For example the client may wish to display a dialogue box when F4 is pressed. If bit 0 of the keyboard short-cut's flags word is set, then the object is shown with the 'Show with Wimp CreateMenu semantics' bit set in the R0 passed to Toolbox_ShowObject.

Sets of Keyboard short-cuts will normally be defined by the client application in its resource file, but they can also be added and removed dynamically using the Window_AddKeyboardShortcuts (page 303) and Window_RemoveKeyboardShortcuts (page 304) methods, passing as an argument an array of mappings.

### Pointer shapes

Each Window object can optionally have a pointer shape defined, giving the name of a sprite to use and its hot spot.

Whenever the Wimp pointer enters this Window, causing a PointerEnteringWindow event, the Toolbox changes the pointer shape appropriately.

In most cases a pointer shape is attached to the Window at resource editing time by entering the name of the sprite to be used, and the pointer's hot spot. If the application wishes dynamically to change the pointer for a given Window (maybe based on a mode of operation which is defined by the application, e.g. display mode or editing mode), then this can be done using the Window_SetPointer method described in *Window_SetPointer 5* on page 299.

The name of the sprite used for the Window's pointer shape and its hot spot can be read by using the Window_GetPointer method described in *Window_GetPointer 6* on page 300.

### Help messages

Each Window object can optionally have attached to it a Help message.

Whenever the Wimp delivers a HelpRequest message to the client application for this Window, the attached Help message is sent back automatically by the Toolbox.

In most cases a help message is attached to the Window at resource editing time. A Window's Help message can be set dynamically using the *Window_SetHelpMessage 7* described on page 301.

The text of the Help message can be read using the Window_GetHelpMessage method.

**Changing a window's title**

One of the attributes of a Window which is specified in the template for that Window is the text which appears in its title bar.

A Window's title can be changed dynamically at run-time using the Window_SetTitle method.

The current title string can be read using the Window_GetTitle method.

**Getting and setting a Window's client handle**

The client handle for a Window is set and read using SWI Toolbox_SetClientHandle and SWI Toolbox_GetClientHandle respectively.

A typical use of this client handle would be to hold a pointer to a data structure containing the state of a document which is being displayed in this Window in a multi-document editor.

## Window methods

The following methods are all invoked by calling SWI Toolbox_MiscOp with:

| | |
|---|---|
| R0 | holding a flags word |
| R1 | being a Window id |
| R2 | being the method code which distinguishes this method |
| R3-R9 | potentially holding method-specific data |

### Window_GetWimpHandle 0

**On entry**

R0 = flags
R1 = Window object id
R2 = 0

**On exit**

R0 = Wimp window handle for this window

**Use**

This method returns the handle of the underlying Wimp window used to implement this Window object.

**C veneer**

```
extern _kernel_oserror *window_get_wimp_handle ( unsigned int flags,
                                                 ObjectId window,
                                                 int *window_handle
                                               );
```

## Window_AddGadget 1

**On entry**

R0 = flags
R1 = Window object id
R2 = 1
R3 = pointer to description block for gadget

**On exit**

R0 = component id
R1-R9 preserved

**Use**

This method adds a gadget to the list of gadgets for this Window object. The format of the description block depends on the type of gadget being added.

If the Window is currently open on the screen, then the gadget will immediately be visible in the Window.

If the gadget's component id is specified as -1, then the Toolbox wil allocate an unused component id.

**C veneer**

```
extern _kernel_oserror *window_add_gadget ( unsigned int flags,
                                            ObjectId window,
                                            Gadget *gadget,
                                            ComponentId *gadget_component
                                          );
```

## Window_RemoveGadget 2

**On entry**

R0 = flags
R1 = Window object id
R2 = 2
R3 = component id

**On exit**

R1-R9 preserved

**Use**

This method removes a gadget from a Window object. If the Window is currently displayed on the screen, then this removal results in a redraw of the Window by the Toolbox.

**C veneer**

```
extern _kernel_oserror *window_remove_gadget ( unsigned int flags,
                                               ObjectId window,
                                               ComponentId gadget
                                             );
```

## Window_SetMenu 3

**On entry**

R0 = flags
R1 = Window object id
R2 = 3
R3 = menu object id

**On exit**

R1-R9 preserved

**Use**

This method is used to set the Menu which will be displayed when the Menu button is pressed over this Window object. The Toolbox handles opening the Menu for you.

If R3 is 0, then the Menu for this Window is detached.

**C veneer**

```
extern _kernel_oserror *window_set_menu ( unsigned int flags,
                                          ObjectId window,
                                          ObjectId menu_id
                                        );
```

## Window_GetMenu 4

**On entry**

R0 = flags
R1 = Window object id
R2 = 4

**On exit**

R0 = Menu id

**Use**

This method is used to get the id of the Menu which will be displayed when the Menu button is pressed over this Window object.

**C veneer**

```
extern _kernel_oserror *window_get_menu ( unsigned int flags,
                                          ObjectId window,
                                          ObjectId *menu_id
                                        );
```

## Window_SetPointer 5

**On entry**

R0 = flags
R1 = Window object id
R2 = 5
R3 = pointer to name of sprite to use for pointer
R4 = x hot spot
R5 = y hot spot

**On exit**

R1-R9 preserved

**Use**

This method is used to set the Pointer shape which will be used when the pointer enters this Window object. The Toolbox handles setting the Wimp Pointer shape for you.

If R3 is 0, then the Pointer for this Window is detached.

**C veneer**

```
extern _kernel_oserror *window_set_pointer ( unsigned int flags,
                                             ObjectId window,
                                             char *sprite_name,
                                             int x_hot_spot,
                                             int y_hot_spot
                                           );
```

## Window_GetPointer 6

**On entry**

R0 = flags
R1 = Window object id
R2 = 6
R3 = pointer to buffer
R4 = size of buffer to hold sprite name
R5 = x hot spot
R6 = y hot spot

**On exit**

R4 = size of buffer required for sprite name (if R3 was 0)
else buffer pointed at by R3 holds sprite name
R4 holds number of bytes written to buffer

**Use**

This method is used to get the name of the sprite which will be used when the pointer enters this Window object, and to get the pointer's hot spot.

**C veneer**

```
extern _kernel_oserror *window_get_pointer ( unsigned int flags,
                                             ObjectId window,
                                             char *buffer,
                                             int buff_size,
                                             int *nbytes,
                                             int *x_hot_spot,
                                             int *y_hot_spot
                                           );
```

## Window_SetHelpMessage 7

**On entry**

R0 = flags
R1 = Window object id
R2 = 7
R3 = pointer to message text

**On exit**

R1-R9 preserved

**Use**

This method is used to set the help message which will be returned when a Help Request message is received for this Window object. The Toolbox handles the reply message for you.

If R3 is 0, then the Help Message for this Window is removed.

**C veneer**

```
extern _kernel_oserror *window_set_help_message ( unsigned int flags,
                                                  ObjectId window,
                                                  char *message_text
                                                );
```

## Window_GetHelpMessage 8

**On entry**

R0 = flags
R1 = Window object id
R2 = 8
R3 = pointer to buffer
R4 = size of buffer to hold message text

**On exit**

R4 = size of buffer required for message text (if R3 was 0)
else Buffer pointed at by R3 holds message text
R4 holds number of bytes written to buffer

**Use**

This method is used to read the help message which will be returned when a Help Request message is received for this Window object.

**C veneer**

```
extern _kernel_oserror *window_get_help_message ( unsigned int flags,
                                                  ObjectId window,
                                                  char *buffer,
                                                  int buff_len,
                                                  int *nbytes
                                                );
```

## Window_AddKeyboardShortcuts 9

**On entry**

R0 = flags
R1 = Window object id
R2 = 9
R3 = number of short-cuts to add
R4 = pointer to memory block containing an array of description blocks for the keyboard short-cuts. Each block is laid out in memory as described in *Window templates* on page 317

**On exit**

R1-R9 preserved

**Use**

This method adds a number of keyboard short-cuts to the list of keyboard short-cuts for this Window object. When a Key Pressed event is received for this Window, the given Toolbox event is raised as the next Wimp event for the client application.

If any of the keyboard short-cuts are already defined for this Window, then they are replaced by the new short-cuts.

**C veneer**

```
extern _kernel_oserror *window_add_keyboard_shortcuts ( unsigned int flags,
                                                        ObjectId window,
                                                        int no_shortcuts,
                                                        KeyboardShortcut *shortcuts
                                                      );
```

## Window_RemoveKeyboardShortcuts 10

**On entry**

R0 = flags
R1 = Window object id
R2 = 10
R3 = -1 means remove all keyboard short-cuts
or R3 = number of short-cuts to remove
R4 = pointer to an array of key short-cuts to be removed
(number given in R3)

**On exit**

R1-R9 preserved

**Use**

This method removes a number of keyboard short-cuts which have been associated with this Window using the Window_AddKeyboardShortcuts method.

**C veneer**

```
extern _kernel_oserror *window_remove_keyboard_shortcuts ( unsigned int flags,
                                                           ObjectId window,
                                                           int no_remove,
                                                           KeyboardShortcut *shortcuts
                                                         );
```

## Window_SetTitle 11

**On entry**

R0 = flags
R1 = Window object id
R2 = 11
R3 = pointer to new text for title bar

**On exit**

R1-R9 preserved

**Use**

This method changes the text in a Window's title bar. If the string is too long for the title bar's buffer, an error is returned.

**C veneer**

```
extern _kernel_oserror *window_set_title ( unsigned int flags,
                                           ObjectId window,
                                           char *title
                                         );
```

## Window_GetTitle 12

**On entry**

R0 = flags
R1 = Window object id
R2 = 12
R3 = pointer to buffer to hold title text (or 0)
R4 = size of buffer

**On exit**

R4 = size of buffer required (if R3 was 0)
else Buffer pointed at by R3 holds title text
R4 holds number of bytes written to buffer

**Use**

This method returns the string currently used in a Window's title bar.

**C veneer**

```
extern _kernel_oserror *window_get_title ( unsigned int flags,
                                           ObjectId window,
                                           char *buffer,
                                           int buff_size,
                                           int *nbytes
                                         );
```

## Window_SetDefaultFocus 13

**On entry**

R0 = flags
R1 = Window object id
R2 = 13
R3 = component id

**On exit**

R1-R9 preserved

**Use**

This method sets the default focus component for a window. As with the template, a value of -1 means no default focus, and -2 means put the focus in the window.

Note that this sets the default, i.e. only takes effect when next shown.

**C veneer**

```
extern _kernel_oserror *window_set_default_focus ( unsigned int flags,
                                                   ObjectId window,
                                                   ComponentId focus
                                                 );
```

## Window_GetDefaultFocus 14

**On entry**

R0 = flags
R1 = Window object id
R2 = 14

**On exit**

R0 = component id
R1-R9 preserved

**Use**

This method returns the default focus component of a window.

**C veneer**

```
extern _kernel_oserror *window_get_default_focus ( unsigned int flags,
                                                   ObjectId window,
                                                   ComponentId *focus
                                                 );
```

## Window_SetExtent 15

**On entry**

R0 = flags
R1 = Window object id
R2 = 15
R3 = pointer to extent bounding box:
+0 minimum x coordinate
+4 minimum y coordinate
+8 maximum x coordinate
+12maximum y coordinate

**On exit**

R1-R9 preserved

**Use**

This method changes the extent of the underlying Wimp window.

**C veneer**

```
extern _kernel_oserror *window_set_extent ( unsigned int flags,
                                            ObjectId window,
                                            BBox *extent
                                          );
```

## Window_GetExtent 16

**On entry**

R0 = flags
R1 = Window object id
R2 = 16
R3 = pointer to four word block to hold extent

**On exit**

R1-R9 preserved and block pointed to by R3 updated:
+0 minimum x coordinate
+4 minimum y coordinate
+8 maximum x coordinate
+12 maximum y coordinate

**Use**

This method returns the extent of the underlying Wimp window.

**C veneer**

```
extern _kernel_oserror *window_get_extent ( unsigned int flags,
                                            ObjectId window,
                                            BBox *extent
                                          );
```

## Window_ForceRedraw 17

**On entry**

R0 = flags
R1 = Window object id
R2 = 17
R3 = pointer to area to redraw:
+0 minimum x coordinate
+4 minimum y coordinate
+8 maximum x coordinate
+12maximum y coordinate

**On exit**

R1-R9 preserved

**Use**

This method forces a redraw on the area of the window given by the work area coordinates pointed to by R3.

**C veneer**

```
extern _kernel_oserror *window_force_redraw ( unsigned int flags,
                                              ObjectId window,
                                              BBox *redraw_box
                                            );
```

## Window_SetToolBars 18

**On entry**

R0 = mask
    bit 0 set means set internal bl toolbar
    bit 1 set means set internal tl toolbar
    bit 2 set means set external bl toolbar
    bit 3 set means set external tl toolbar
R3 = object id of internal bl toolbar
R4 = object id of internal tl toolbar
R5 = object id of external bl toolbar
R6 = object id of external tl toolbar

**Use**

This method sets the object ids of the toolbars that are attached to a particular window object. If the object is showing then the new toolbars will be shown, and any toolbars of the same type will be hidden (it is not possible to have more than one toolbar of each type). The mask allows selective setting of toolbars.

Passing an Id of zero means that there is no toolbar of that type.

**C veneer**

```
extern _kernel_oserror *window_set_tool_bars ( unsigned int flags,
                                               ObjectId window,
                                               ObjectId ibl,
                                               ObjectId itl,
                                               ObjectId ebl,
                                               ObjectId etl
                                             );
```

## Window_GetToolBars 19

**On entry**

R0 = mask
bit 0 set means return internal bl toolbar
bit 1 set means return internal tl toolbar
bit 2 set means return external bl toolbar
bit 3 set means return external tl toolbar

**On exit**

R0 = object id of internal bl toolbar
R1 = object id of internal tl toolbar
R2 = object id of external bl toolbar
R3 = object id of external tl toolbar

**Use**

This method returns the object ids of the toolbars that are attached to a window object. By setting the mask it is possible to control which ids are returned.

**C veneer**

```
extern _kernel_oserror *window_get_tool_bars ( unsigned int flags,
                                               ObjectId window,
                                               ObjectId *ibl,
                                               ObjectId *itl,
                                               ObjectId *ebl,
                                               ObjectId *etl
                                             );
```

## Other SWIs

### SWI Window_GetPointerInfo (0x82883)

**On entry**

R0 = flags

**On exit**

R0 = x position
R1 = y position
R2 = buttons
bit set
0 adjust
1 menu
2 select
8 not over a toolbox window
R3 = Window id, or Wimp window handle if bit 8 set in R2
R4 = component id, or icon handle if bit 8 of R2 set

**Use**

This SWI is analogous to Wimp_GetPointerInfo, but returns Object ids and Component ids if the pointer is over a toolbox window.

**C veneer**

```
extern _kernel_oserror *window_get_pointer_info ( unsigned int flags,
                                                  int *x_pos,
                                                  int *y_pos,
                                                  int *buttons,
                                                  ObjectId *window,
                                                  ComponentId *component
                                                );
```

## SWI Window_WimpToToolbox (0x82884)

**On entry**

R0 = flags
R1 = Wimp window handle
R2 = icon handle

**On exit**

R0 = toolbox object handle for window
R1 = component id

**Use**

This SWI returns the object handle and component id that contains the specified icon.

If the Wimp handle is not known by the toolbox, then the returned object id is 0.

Note that this only applies to Window objects.

**C veneer**

```
extern _kernel_oserror *window_wimp_to_toolbox ( unsigned int flags,
                                                 int window_handle,
                                                 int icon_handle,
                                                 ObjectId *object,
                                                 ComponentId *component
                                               );
```

## SWI Window_ExtractGadgetInfo (0x828be)

**On entry**

R0 = flags
R1 = pointer window template
R2 = component id to match

**On exit**

R0 = pointer to Gadget
R1 = size of gadget

**Use**

This SWI returns a pointer to a block of memory suitable for passing to Window_AddGadget. It is typically used in conjunction with Toolbox_LookupTemplate and intended to be used for dynamic windows such as the Print dialogue box, or a task manager type application.

Note that the returned area should be copied as it cannot be guaranteed to persist for the duration of the task.

See *Implementing hotspots* on page 55 for an example of using this SWI.

**C veneer**

```
extern _kernel_oserror *window_extract_gadget_info ( unsigned int flags,
                                                     ObjectTemplateHeader *templ,
                                                     ComponentId gadget,
                                                     void **desc,
                                                     int *size
                                                     );
```

## Window events

The Window class generates the following Toolbox events:

### Window_AboutToBeShown (0x82880)

**Block**

| | |
|---|---|
| + 8 | 0x82880 |
| + 12 | flags (as passed in to Toolbox_ShowObject) |
| + 16 | value as passed in R2 to ToolBox_ShowObject |
| + 20... | block as passed in R3 to ToolBox_ShowObject |

**Use**

This Toolbox event is raised by the Toolbox when Toolbox_ShowObject is called on a Window which has the appropriate bit set in its template flags word. It enables a client application to set any appropriate attributes of the Window, before it appears on the screen.

**C data type**

```
typedef struct
{
  ToolboxEventHeader  hdr;
  int                 show_type;
  union
  {
    TopLeft                 top_left;
    WindowShowObjectBlock   full_spec;
  } info;
} WindowAboutToBeShownEvent;
```

### Window_HasBeenHidden (0x82890)

**Block**

+ 8 0x82890

**Use**

This Toolbox event is raised by the Toolbox when Toolbox_HideObject is called on a Window which has the appropriate bit set in its template flags word. It enables a client application to clear up after a window has been closed. It is also raised when clicking a non-local action button or clicking outside a window that was opened with 'CreateMenu' semantics.

**C data type**

```
typedef struct
{
  ToolboxEventHeader  hdr;
} WindowHasBeenHiddenEvent;
```

## Window templates

The layout of a Window template is shown below. Fields which have types MsgReference and StringReference are those which will require relocation when they are loaded from a resource file. If the template is being constructed in memory, then these fields should be real pointers (i.e. they do not require relocation). Note that the version in the object header should be 102.

For more details on relocation, see appendix *Resource File Formats* on page 499.

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| help_message | 4 | MsgReference |
| max_help | 4 | word |
| pointer_shape | 4 | StringReference |
| max_pointer_shape | 4 | word |
| pointer_x_hot | 4 | word |
| pointer_y_hot | 4 | word |
| menu | 4 | StringReference |
| num_keyboard_shortcuts | 4 | word |
| keyboard_shortcuts | 4 | ObjectOffset |
| num_gadgets | 4 | word |
| gadgets | 4 | ObjectOffset |

| Field | Size in bytes | Type |
|---|---|---|
| default_focus | 4 | word |
| show_event | 4 | word |
| internal_bl | 4 | StringReference |
| internal_tl | 4 | StringReference |
| external_bl | 4 | StringReference |
| external_tl | 4 | StringReference |
| hide_event | 4 | word |
| window | 88 | WimpWindow |
| data | variable | array of bytes |

A WimpWindow is an 88-byte structure with the following fields:

| Field | Size in bytes | Type |
|---|---|---|
| vis_xmin | 4 | word |
| vis_ymin | 4 | word |
| vis_xmax | 4 | word |
| vis_ymax | 4 | word |
| scroll_x | 4 | word |
| scroll_y | 4 | word |
| behind | 4 | word |
| window_flags | 4 | word |
| title_fore | 1 | byte |
| title_back | 1 | byte |
| work_fore | 1 | byte |
| work_back | 1 | byte |
| scroll_outer | 1 | byte |
| scroll_inner | 1 | byte |
| title_inputfocus | 1 | byte |
| filler | 1 | byte (must be 0) |
| work_xmin | 4 | word |
| work_ymin | 4 | word |
| work_xmax | 4 | word |
| work_ymax | 4 | word |
| title_flags | 4 | word |
| button_type | 4 | word |
| sprite_area | 4 | SpriteAreaReference |
| min_width | 2 | half-word |
| min_height | 2 | half-word |
| title_text | 4 | MsgReference |
| title_validation | 4 | StringReference |
| title_buflen | 4 | word |
| num_icons | 4 | word (must be zero) |

**Keyboard short-cut**

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| wimp_key_code | 4 | word |

| Field | Size in bytes | Type |
|---|---|---|
| key_event | 4 | word |
| key_show | 4 | StringReference |

**Gadget**

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| type/size | 4 | word |
| xmin | 4 | word |
| ymin | 4 | word |
| xmax | 4 | word |
| ymax | 4 | word |
| component_id | 4 | word |
| help_text | 4 | MsgReference |
| max_help | 4 | word |
| data | variable | array of bytes |

## Window Wimp event handling

Certain Wimp events for a Window are handled by the Window class, and either acted upon for you, or result in the raising of a Toolbox event. Such events are listed below:

| Wimp event | Action |
|---|---|
| Open Window Request | if the ‘auto-open’ bit is set for this Window object, then Toolbox_ShowObject is called for this Window |
| Close Window Request | if the ‘auto-close’ bit is set for this Window object, then Toolbox_HideObject is called for this Window |
| Pointer Leaving Window | if there is a pointer shape defined for this Window, then the pointer is set back to its default shape |
| Pointer Entering Window | if there is a pointer shape defined for this Window, then the pointer is set to that shape |
| Mouse Click | if the Menu button has been pressed, and there is a Menu object attached to this Window, then the Menu is shown using Toolbox_ShowObject |

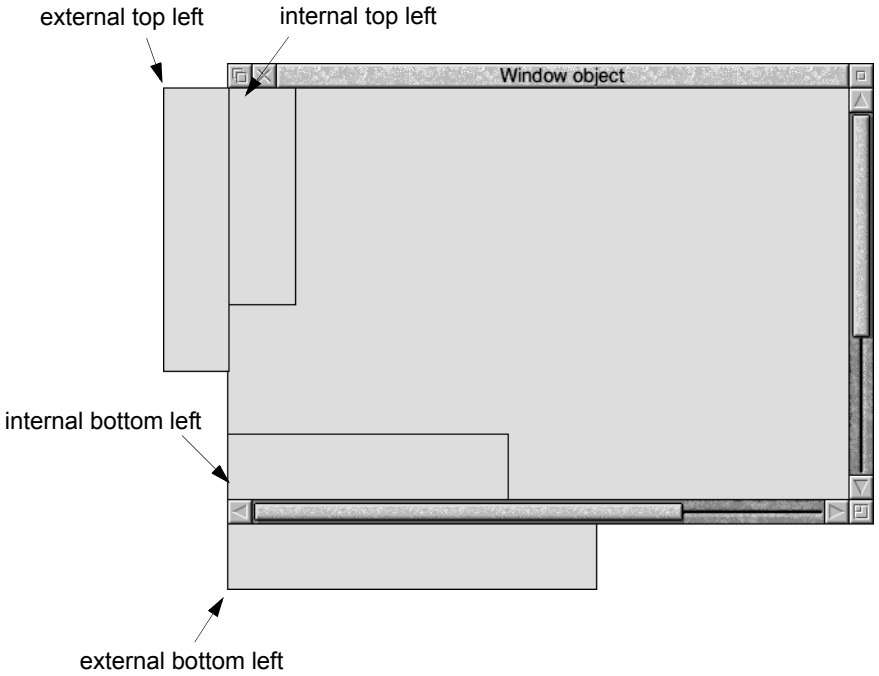
| Wimp event | Action |
|---|---|
| Key Pressed | if a keyboard short-cut for the given Wimp key code is attached to this Window, then its Toolbox event is raised as the next Wimp event for the client application |
| User Msg | Message_HelpRequest<br>if a help message is attached to this Window, then a reply is sent on the application's behalf |

## Toolbars

Toolbars are attachments to windows, and are used mainly as tool boxes and status lines. They cannot exist purely by themselves. By using the toolbars supplied by the Window module, applications will have a consistent mechanism for displaying/accessing such functionality. It is not intended that they be used for anything beyond this.

## User interface

A toolbar is a restricted window object – it cannot have any window furniture (such as a title bar), nor does it have an absolute position when shown on the screen. It is anchored either to the bottom left or to the top left of the parent's visible area; i.e. it does not move or scroll when the parent scrolls its work area.




A toolbar can be considered to be either internal (in which case its size will be clipped when the parent resizes) or external (i.e. lying entirely outside the parent's visible area). On moving a window with an external toolbar close to the extremities of the screen, the bar will 'bounce' over the window until the window itself moves off screen.

Toolbars are displayed in a definite order:

- external toolbars will always be displayed above internal ones

- top left toolbars will always be displayed above bottom left ones.

Usually, this will only be noticed when reducing the size of a window.

For example, when moving a window to the left of the screen, the external toolbar will be displayed above any toolbar inside the window.

### Use of toolbars

**Application tool box**

It is anticipated that the top left variety of toolbars will be used as application tool boxes, i.e. they will consist of gadgets that are used to control the behaviour of the application. The decision as to whether an internal or external one is used would typically depend on the number of ‘tools’ that are required.

**Status lines**

Internal bottom left toolbars are usually for status lines. For example:

```
The data is loading, 50% complete
```

and external bottom left toolbars for toolboxes that require width (e.g. because they contain a writable) but are unlikely to be as wide as the work area (in which case they would leave an irregular work space).

Note that if a toolbar contains a non-local action button then clicking on it will hide that toolbar.

## Application program interface

### Attributes

Toolbar object attributes are described in the window attributes section on page 288.

Note that a toolbar should not have toolbars itself.

### Manipulating a toolbar

#### Creating and deleting a toolbar object

Toolbar objects are created and deleted using the standard Toolbox_CreateObject and Toolbox_DeleteObject methods.

#### Showing and Hiding

A toolbar can only be shown whilst its parent is showing. The only defined show type is ShowAsDefault. This will make the window module show the toolbar in the place appropriate for its type. It is possible to hide a toolbar without hiding its parent. If a toolbar is hidden, then this is 'remembered' such that hiding then showing the parent will result in the toolbar still being hidden.

When a toolbar object is displayed on the screen using SWI Toolbox_ShowObject it behaves in the same way as shown in *User interface* on page 322.

## Toolbar methods

Toolbars use the same methods as windows (see *Window methods* on page 295). However, the behaviour of the following methods are undefined:

Window_SetTitle
Window_GetTitle
Window_SetToolBars
Window_GetToolBars
Window_AddKeyboardShortcuts
Window_RemoveKeyboardShortcuts

Getting and setting the toolbars associated with a window object are described in *Window_GetToolBars 19* on page 312 and *Window_SetToolBars 18* on page 311.

Normally this would be done using ResEd.

# Gadgets

## Application Program Interface

Gadgets are not objects in their own right, but exist only as a component of a Window object. Within that object they have unique component ids.

A gadget is essentially a part of a Window which provides functionality (for example, a button or a slider), and is usually implemented using Wimp icons. The use of icons is transparent to the client, who manipulates the gadgets using higher-level, abstract methods which are appropriate to the particular gadget type.

Wherever a gadget is implemented as a set of Wimp icons, the client can access these using low-level Wimp SWIs, but in the vast majority of cases this should not prove necessary.

Some gadgets are 'Composite' in that they consist of gadgets themselves. These are identifiable by the client as they have a NULL icon list. The client will receive toolbox events on both the composite gadget and the gadgets that make them up, but will generally only be interested in the former. Certain gadgets have methods for accessing the component ids of the gadgets that make them up, e.g. NumberRange_GetComponents.

Some gadgets support anti-aliased fonts in place of the system font (which may itself be an outline font on RISC OS 3 (version 3.5). When this is the case, the Window module handles mode changes and losing fonts on the client's behalf.

The window module reserves all component ids greater than 0xffffff. Standard dialogues use the range 0x800000 to 0xffffff, leaving 0 to 0x7fffff free for the client.

There are many kinds of gadget. The Toolbox provides facilities to allow the client application to manipulate a particular gadget in a manner which is appropriate to that gadget, rather than in ‘low-level’ terms like setting the state of a Wimp icon. The set of gadgets is defined to fit in with the RISC OS 3 Style Guide, and thus to encourage a standard look and feel across dialogue boxes.

The available set of gadgets is currently:

| Gadget | See page |
|---|---|
| *Labelled boxes* | 367 |
| *Number ranges* | 368 |
| *Option buttons* | 376 |
| *Pop-up menus* | 383 |
| *Radio buttons* | 387 |
| *Sliders* | 395 |
| *String sets* | 403 |
| *Writable fields* | 411 |

## Attributes

All gadgets have the following attributes which are specified in a window template, and most can be manipulated at run-time by the client application:

| Attribute | Description |
|---|---|
| flags word | **Bit** **Meaning**<br>30 when set, gadget is at the back, i.e. created first<br>31 when set, gadget is ‘faded’ |
| type/size | this holds the size of the gadget’s template (including its header) in its top two bytes, and the type of the gadget in its lower two bytes. The list of currently known gadget types is given below. |
| xmin | the minimum x coordinate of the gadget’s bounding box (in window work area coordinates). |
| ymin | the minimum y coordinate of the gadget’s bounding box (in window work area coordinates). |
| xmax | the maximum x coordinate of the gadget’s bounding box (in window work area coordinates). |
| ymax | the maximum y coordinate of the gadget’s bounding box (in window work area coordinates). |
| component id | this identifies the gadget uniquely within this Window |
| help message | when a HelpRequest message is received for this gadget, then this string is sent back in a HelpReply message. If 0, then the help message for the Window will be sent. |
| max help | maximum length in bytes of the gadget’s help message. |

Note that for the gadgets listed below, the size is 'built in' to the Window module, and so the size can be set to zero though gadgets.h defines *gadget*_Type which includes the size.

The type of a gadget is one of:

| Gadget type | Type field |
|---|---|
| Action Button | 128 |
| Option Button | 192 |
| Labelled Box | 256 |
| Label | 320 |
| Radio Button | 384 |
| Display Field | 448 |
| Writable Field | 512 |
| Slider | 576 |
| Draggable | 640 |
| PopUp Menu | 704 |
| Adjuster Arrow | 768 |
| Number Range | 832 |
| String Set | 896 |
| Button | 960 |

## Manipulating a Gadget

Each gadget type defines its own set of methods, and many will have a number of Toolbox events associated with them. This allows the application to receive Toolbox events from user actions, rather than having to deal with mouse clicks and drags on Wimp icons. Most of the low-level Wimp operations are handled automatically by the Toolbox.

Normally all of the gadgets in a particular Window object will be specified in the template for that Window in the resource file, but the Toolbox provides two methods for adding and removing gadgets from a Window object dynamically, namely Window_AddGadget and Window_RemoveGadget.

All gadgets have standard attributes, which give the gadget's component id in this Window, the gadget's bounding box, and the help message to be associated with this gadget. These attributes are normally specified in the application's resource file; the Help messages can be changed and read using the methods Gadget_SetHelpMessage/Gadget_GetHelpMessage. Sending back a help message is automatically handled by the Toolbox.

Each gadget has a flags word which defines the behaviour of that gadget; the exact list of bit settings in this flags word depends on the type of gadget. The client can read and set this word using the Gadget_GetFlags and Gadget_SetFlags methods. The top 8 bits of this flags word are generic flags of relevance to all gadgets. The other 24 bits are used to hold Gadget-specific flags. Currently the defined generic flags are:

| Bit | Meaning when set |
|---|---|
| 30 | Gadget is at the back, i.e. created first |
| 31 | Gadget is 'faded' |

There is a gadget method which returns a list of Wimp icon numbers for the icons used to implement the gadget. The details of this list and the way in which icon numbers map to the individual components of the gadget are specific to each gadget, and this mapping is documented below for each gadget type. The method is called Gadget_GetIconList.

This is implementation specific and subject to change in future releases of the window module:

| Gadget type | Number of icon numbers returned | Icon list |
|---|---|---|
| action button | 1 | the icon for the action button |
| option button | 2 | the icon for the sprite<br>the icon for the text |
| labelled box | 2 | the icon for the label<br>the icon for the box |
| label | 1 | the icon for the label |
| radio button | 2 | the icon for the sprite<br>the icon for the text |
| display field | 1 | the icon for the display field |
| writable field | 1 | the icon for the writable field |
| slider | 3 | the icon for the 'well'<br>the icon for the 'background'<br>the icon for the 'bar' |
| draggable | 1 | the icon for the draggable |
| pop-up menu | 1 | the icon for the PopUp's button |
| adjuster arrow | 1 | the icon for the arrow |
| number range | 0 | composite |
| string set | 0 | composite |
| button | 1 | |

Composite gadgets have specific methods to get the component ids of their constituent gadgets. In this way run time methods (e.g. the colour of a slider in a number range) may be applied to the underlying gadgets. It is unlikely however that this will be particularly useful and could in fact affect the behaviour of the toolbox.

## Generic gadget methods

In all of the methods on gadgets

| | |
|---|---|
| R0 | is used as a flags word |
| R1 | holds the object id of this gadget's parent Window object |
| R2 | holds the method code |
| R3 | holds the component id for this gadget |
| R4-R9 | potentially holding method-specific data |

The following methods can be applied to all gadgets.

### Gadget_GetFlags 64

**On entry**

R0 = 0
R1 = Window object id
R2 = 64
R3 = Gadget component id

**On exit**

R0 = flags settings for this gadget

**Use**

This method returns the flags word for the given gadget.

**C veneer**

```
extern _kernel_oserror *gadget_get_flags ( unsigned int flags,
                                           ObjectId window,
                                           ComponentId gadget,
                                           unsigned int *flags_settings
                                         );
```

## Gadget_SetFlags 65

**On entry**

R1 = Window object id
R2 = 65
R3 = Gadget component id
R4 = new flags settings

**On exit**

R1-R9 preserved

**Use**

This method sets the flags word for the given gadget. The only flags that can usefully be changed are the faded bits. Modifying other bits is undefined.

**C veneer**

```
extern _kernel_oserror *gadget_set_flags ( unsigned int flags,
                                           ObjectId window,
                                           ComponentId gadget,
                                           unsigned int new_flags_settings
                                         );
```

## Gadget_SetHelpMessage 66

**On entry**

R0 = flags
R1 = Window object id
R2 = 66
R3 = Gadget component id
R4 = pointer to help message text

**On exit**

R1-R9 preserved

**Use**

This method sets the help message which will be returned, when a help request is received for this gadget.

**C veneer**

```
extern _kernel_oserror *gadget_set_help_message ( unsigned int flags,
                                                  ObjectId window,
                                                  ComponentId gadget,
                                                  char *message_text
                                                );
```

## Gadget_GetHelpMessage 67

**On entry**

R0 = flags
R1 = Window object id
R2 = 67
R3 = Gadget component id
R4 = pointer to buffer
R5 = size of buffer

**On exit**

R5 = size of buffer required to hold help text (if R4 was 0)
else buffer pointed at by R4 holds help text
R5 gives number of bytes written to buffer

**Use**

This method returns the help message which will be returned, when a help request is received for this gadget.

**C veneer**

```
extern _kernel_oserror *gadget_get_help_message ( unsigned int flags,
                                                  ObjectId window,
                                                  ComponentId gadget,
                                                  char *buffer,
                                                  int buff_size,
                                                  int *nbytes
                                                );
```

## Gadget_GetIconList 68

**On entry**

R0 = flags
R1 = Window object id
R2 = 68
R3 = Gadget component id
R4 = pointer to buffer
R5 = size of buffer

**On exit**

R5 = size of buffer required to hold icon list (if R4 was 0)
else buffer pointed at by R4 holds list of Wimp icon numbers for this gadget
R5 holds number of bytes written to buffer

**Use**

This method returns a list of Wimp icon numbers (integers) for the icons used to implement this gadget. For a composite gadget the size returned will be zero.

**C veneer**

```
extern _kernel_oserror *gadget_get_icon_list ( unsigned int flags,
                                               ObjectId window,
                                               ComponentId gadget,
                                               int *buffer,
                                               int buff_size,
                                               int *nbytes
                                             );
```

The client should not cache the results of this call, since these values may change at a later date.

## Gadget_SetFocus 69

**On entry**

R0 = flags

**On exit**

R1-R9 preserved

**Use**

This method sets the input focus to the given component of a window. Note that such a component must be a writable field, or a composite gadget which includes a writable field such as a number range.

**C veneer**

```
extern _kernel_oserror *gadget_set_focus ( unsigned int flags,
                                           ObjectId window,
                                           ComponentId component
                                         );
```

## Gadget_GetType 70

**On entry**

R0 = 0
R1 = Window object id
R2 = 70
R3 = Gadget component id

**On exit**

R0 = type of this Gadget

**Use**

Usage:

This method returns the type of the given gadget.

**C veneer**

```
extern _kernel_oserror *gadget_get_type ( unsigned int flags,
                                          ObjectId window,
                                          ComponentId gadget,
                                          int *type
                                        );
```

## Gadget_MoveGadget 71

**On entry**

R0 = flags
R1 = Window object id
R2 = 71
R3 = Gadget component id
R4 = pointer to new bounding box

**On exit**

R1-R9 preserved

**Use**

This method moves an already created gadget within a window. Note that as a new bounding box is given, it allows the gadget to be resized as well, though the exact behaviour of this feature will depend on the gadget type.

**C veneer**

```
extern _kernel_oserror *gadget_move_gadget ( unsigned int flags,
                                             ObjectId window,
                                             ComponentId gadget,
                                             BBox *new_bbox
                                           );
```

## Gadget_GetBBox 72

**On entry**

R0 = flags
R1 = Window object id
R2 = 72
R3 = Gadget component id
R4 = pointer to 4 word buffer

**On exit**

R1-R9 preserved

**Use**

This method copies the bounding box of a gadget into the supplied buffer.

**C veneer**

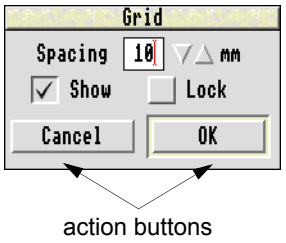
```
extern _kernel_oserror *gadget_get_bbox ( unsigned int flags,
                                          ObjectId window,
                                          ComponentId gadget,
                                          BBox *box
                                        );
```

## Gadget Wimp event handling

| Wimp event | Action |
|---|---|
| Mouse Click | if Select or Adjust on an action button, option button or radio button member, then if a Toolbox event is associated with this event, it is raised. Otherwise the appropriate default Toolbox event is raised.<br>if on a pop-up menu button, then the associated Menu is shown.<br>if on a draggable then a Draggable_Click/Draggable_DoubleClick is reported. |
| Key Pressed | This depends on the type of gadget.<br>For a writable field, if the keystroke is a down or up arrow, then the caret is placed in the next or previous writable field (using the field's 'before' and 'after' values).<br>If return is pressed, then the Default action button is activated (if present). |
| User Message | Message_HelpRequest<br>if a help message is attached to the gadget, then a reply is sent on the application's behalf. |

## Action buttons

An action button is normally used to invoke an operation which is available from a dialogue box (e.g. a Cancel button or an OK button):




Such a gadget contains a text string, which is specified when the gadget is created.

The above attributes can be set and read using the methods

ActionButton_SetText / ActionButton_GetText

Whenever the user clicks the Select or Adjust buttons on an action button an ActionButton_Selected event is raised with the flags word indicating which mouse button was used. The client can supply an alternative Toolbox event code in the template description for the action button, and can set and read this event code at run-time using the ActionButton_SetEvent and ActionButton_GetEvent methods.

The client can also specify an object which is to be shown when the action button is clicked on using the Select or Adjust buttons. The name of this object can be given in the action button template or manipulated at run-time using the ActionButton_SetClickShow and ActionButton_GetClickShow methods.

In a dialogue box, one action button can be chosen as the Default action button. This button is displayed with a distinctive border, and is activated when Return is pressed. An action button is marked as Default by setting a bit in the flags word for the gadget.

One action button can also be marked as the Cancel action button, by setting a bit in its flags word. This action button is also activated when its parent dialogue box has the input focus, and the user presses Escape.

By default, when an action button is clicked using Select, its parent dialogue box is closed. This behaviour can be over-ridden by setting a bit in the action button's flags word, to indicate that it is a 'local' button, whose effect is only to raise its associated Toolbox event. This facility is generally used for buttons which only have a local effect on the state of the dialogue box itself (e.g. a Try button in a font selector).

Clicking Adjust on an action button, raises its Toolbox event and keeps its parent dialogue box open (if it is marked as a Cancel action button, then the contents of any Gadgets are returned to how they were when the parent window was last shown). The Toolbox does not do this for you.

Bits in the flags word for an action button have the following meaning:

| Bit | Meaning |
|---|---|
| 0 | this is the Default action button |
| 1 | this is the Cancel action button |
| 2 | this is a local action button |
| 3 | if set, then the 'click show' object will be shown transiently (i.e. with Wimp_CreateMenu semantics – default is to show persistently) |

## Action button methods

### ActionButton_SetText 128

**On entry**

R0 = flags
R1 = Window object id
R2 = 128
R3 = Gadget component id
R4 = pointer to text to appear in button

**On exit**

R1-R9 preserved

**Use**

This method sets the text which will be displayed in this action button.

**C veneer**

```
extern _kernel_oserror *actionbutton_set_text ( unsigned int flags,
                                                ObjectId window,
                                                ComponentId action_button,
                                                char *text
                                              );
```

## ActionButton_GetText 129

**On entry**

R0 = flags
R1 = Window object id
R2 = 129
R3 = Gadget component id
R4 = pointer to buffer
R5 = size of buffer

**On exit**

R5 = size of buffer required to hold text (if R4 was 0)
else buffer pointed at by R4 holds text
R5 holds number of bytes written to buffer

**Use**

This method returns the text which is currently displayed in this action button.

**C veneer**

```
extern _kernel_oserror *actionbutton_get_text ( unsigned int flags,
                                                ObjectId window,
                                                ComponentId action_button,
                                                char *buffer,
                                                int buff_size,
                                                int *nbytes
                                              );
```

## ActionButton_SetEvent 130

**On entry**

R0 = flags
R1 = Window object id
R2 = 130
R3 = Gadget component id
R4 = Toolbox event code

**On exit**

R1-R9 preserved

**Use**

This method sets the Toolbox event code which will be raised when this action button is clicked. The rest of the Toolbox event block remains the same as in ActionButton_Selected.

**C veneer**

```
extern _kernel_oserror *actionbutton_set_event ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId action_button,
                                                 int event
                                               );
```

## ActionButton_GetEvent 131

**On entry**

R0 = flags
R1 = Window object id
R2 = 131
R3 = Gadget component id

**On exit**

R0 holds Toolbox event code

**Use**

This method returns the Toolbox event code which will be raised when this action button is clicked.

**C veneer**

```
extern _kernel_oserror *actionbutton_get_event ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId action_button,
                                                 int *event
                                               );
```

## ActionButton_SetClickShow 132

**On entry**

R0 = flags
R1 = Window object id
R2 = 132
R3 = Gadget component id
R4 = object id of the object to show (or 0)
R5 = show flags: bit 0
    if clear show persistently
    if set show transiently

**On exit**

R1-R9 preserved

**Use**

This method allows the client to specify the object to show when the user clicks Select or Adjust on the action button. By setting bit 0 of R5 it is possible to control whether the show is persistent or not.

If R4 is 0, then no object should be shown.

**C veneer**

```
extern _kernel_oserror *actionbutton_set_click_show ( unsigned int flags,
                                                      ObjectId window,
                                                      ComponentId action_button,
                                                      ObjectId object,
                                                      int show_flags
                                                    );
```

## ActionButton_GetClickShow 133

**On entry**

R0 = flags
R1 = Window object id
R2 = 133
R3 = Gadget component id

**On exit**

R0 = id of object to be shown
R1 = show flags

**Use**

This method returns the object id of the object which will be shown when the user clicks Select or Adjust on the action button. If bit 0 of R1 is set on exit, it means that the object will be shown transiently.

**C veneer**

```
extern _kernel_oserror *actionbutton_get_click_show ( unsigned int flags,
                                                      ObjectId window,
                                                      ComponentId action_button,
                                                      ObjectId *object,
                                                      int * show_flags
                                                    );
```

## Action button Toolbox events

### ActionButton_Selected (0x82881)

**Block**

+ 8 0x82881
+ 12 flags
bits 0, 1 and 2 show how the activation was done:
bit 0 set means Adjust was held down
bit 1 reserved
bit 2 set means Select was held down
If bits 0-2 are all 0, then Return was pressed on a default action button, or Escape was pressed activating the cancel action button.
bits 3, 4 and 5 indicate what type of button it is:
bit 3 set means that this is a Default action button
bit 4 set means that this is a Cancel action button
bit 5 set means that this is a local action button (i.e its parent window has not been closed)

**Use**

This Toolbox event is raised when the user clicks on an action button (or in the case of a default action button presses Return), and the client has not specified their own event to be associated with this button (by setting the event in the template to non-zero).

The returned flags word indicates whether the action button is a default and/or a cancel button, and also which mouse button was used to select the button.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
} ActionButtonSelectedEvent;
```

## Action button templates

| Field | Size in bytes | Type |
|---|---|---|
| text | 4 | MsgReference |
| max_text_len | 4 | word |
| click_show | 4 | StringReference |
| event | 4 | word |

## Adjuster arrows

An adjuster arrow gadget will be displayed as an up, down, left or right arrow icon, and clicking on the arrow will raise an Adjuster_Clicked Toolbox event, with an indication of whether the change is up or down:


The adjuster arrow's flags word indicates whether the adjuster is an incrementor or decrementor. There is also a bit to indicate whether this is part of an 'up/down' or 'left/right' pair.

Bits in the flags word for an adjuster arrow have the following meaning:

| Bit | Meaning |
|---|---|
| 0 | set ⇒ 'increment'<br>clear ⇒ 'decrement' |
| 1 | set ⇒ one of an 'up/down' pair<br>clear one of a 'left/right' pair |

## Adjuster arrows Toolbox events

### Adjuster_Clicked (0x8288c)

**Block**

+ 8 0x8288c
+ 16 (0 ⇒ down, 1 ⇒ up)

**Use**

This Toolbox event is raised when the user clicks the mouse on an adjuster arrow (Adjust clicks on a down arrow are reported as 'up', on an up arrow as 'down').

**C data type**

```
typedef struct
{
  ToolboxEventHeader  hdr;
  int                 direction;
} AdjusterClickedEvent;
```

## Adjuster arrow templates

There are no extra fields than those in the gadget header.

## Button gadget

The Button gadget is similar to a Wimp icon. The main differences are that a Button will always have indirected data and that not all icon flags are settable:

- A Button created as sprite only cannot be made into any sort of text Button.
- A Button created as text only cannot be made into a sprite only Button.
- A sprite only Button can only refer to sprites by name and these must be in the Wimp sprite pool or the task's sprite area.

Bits in the flags word for a Button gadget have the following meanings:

| Bit | Meaning |
|---|---|
| 0 | Use the task's sprite area (requires the window to have client sprite area) for sprite only buttons else use the Wimp sprite pool |
| 1 | return menu clicks |

## Button methods

### Button_GetFlags 960

**On entry**

R0 = flags
R1 = Window object id
R2 = 960
R3 = Gadget component id

**On exit**

R0 = icon flags
R1-R9 preserved

**Use**

This method returns the flags of the given button gadget. The bits have the same meaning as those of a Wimp Icon.

**C veneer**

```
extern _kernel_oserror *button_get_flags ( unsigned int flags,
                                           ObjectId window,
                                           ComponentId button,
                                           int *icon_flags
                                         );
```

## Button_SetFlags 961

**On entry**

R0 = flags
R1 = Window object id
R2 = 961
R3 = Gadget component id
R4 = clear word
R5 = EOR word

**On exit**

R1-R9 preserved

**Use**

This method sets the flags of a button. The effect of the clear word and the EOR word are analogous to those of Wimp_SetIconState, except that, as described above, not all combinations are settable.

**C veneer**

```
extern _kernel_oserror *button_set_flags ( unsigned int flags,
                                           ObjectId window,
                                           ComponentId button,
                                           int clear_word,
                                           int EOR_word
                                         );
```

## Button_SetValue 962

**On entry**

R0 = flags
R1 = Window object id
R2 = 962
R3 = Gadget component id
R4 = new value

**On exit**

R1-R9 preserved

**Use**

This method sets the value (i.e. text or sprite name) of a Button.

**C veneer**

```
extern _kernel_oserror *button_set_value ( unsigned int flags,
                                           ObjectId window,
                                           ComponentId button,
                                           char *value
                                         );
```

## Button_GetValue 963

**On entry**

R0 = flags
R1 = Window object id
R2 = 963
R3 = Gadget component id
R4 = pointer to buffer to hold string
R5 = size of buffer

**On exit**

R5 = size of buffer required (if R4 was 0)
else buffer pointed at by R4 holds string
R5 holds number of bytes written to buffer

**Use**

This method returns the value of a Button.

**C veneer**

```
extern _kernel_oserror *button_get_value ( unsigned int flags,
                                           ObjectId window,
                                           ComponentId button,
                                           char *buffer,
                                           int buff_size,
                                           int *nbytes
                                         );
```

## Button_SetValidation 964

**On entry**

R0 = flags
R1 = Window object id
R2 = 964
R3 = Gadget component id
R4 = new value

**On exit**

R1-R9 preserved

**Use**

This method sets the validation string (e.g. sprite name) of a Button.

**C veneer**

```
extern _kernel_oserror *button_set_validation ( unsigned int flags,
                                                ObjectId window,
                                                ComponentId button,
                                                char *value
                                              );
```

## Button_GetValidation 965

**On entry**

R0 = flags
R1 = Window object id
R2 = 965
R3 = Gadget component id
R4 = pointer to buffer to hold string
R5 = size of buffer

**On exit**

R5 = size of buffer required (if R4 was 0)
else buffer pointed at by R4 holds string
R5 holds number of bytes written to buffer

**Use**

This method returns the validation string of a Button.

**C veneer**

```
extern _kernel_oserror *button_get_validation ( unsigned int flags,
                                                ObjectId window,
                                                ComponentId button,
                                                char *buffer,
                                                int buff_size,
                                                int *nbytes
                                              );
```

### Button_SetFont 966

**On entry**

R0 = flags
R1 = Window object id
R2 = 966
R3 = Gadget component id
R4 = pointer to font name to use
R5 = width in 16ths of a point
R6 = height in 16ths of a point

**On exit**

R1-R9 preserved

**Use**

This method makes the Button use an anti-aliased font. If the font name is NULL, then the field will use system font.

**C veneer**

```
extern _kernel_oserror *button_set_font ( unsigned int flags,
                                          ObjectId window,
                                          ComponentId button,
                                          char *font_name,
                                          int width,
                                          int height
                                        );
```

## Button toolbox events

The button gadget does not have any toolbox events. All click or key presses are returned as Wimp events but with the component and window id of the tasks id block updated.

## Button templates

| Field | Size in bytes | Type |
|---|---|---|
| button_flags | 4 | Word |
| value | 4 | MsgReference |
| max_value | 4 | word |
| validation | 4 | StringReference |
| max_validation | 4 | word |

## Display fields

A display field gadget is used to display information in a ‘read-only’ manner:


The display field has a ‘slabbed in’ boxed display area in which a text string is displayed. The contents of the display area can be set and read using the DisplayField_SetValue and DisplayField_GetValue methods.

Bits in the flags word for a Label have the following meaning:

| Bit | Meaning |
|---|---|
| 1-2 | justification:<br>0 ⇒ left-justified<br>1 ⇒ right-justified<br>2 ⇒ centred |

## Display field methods

### DisplayField_SetValue 448

**On entry**

R0 = flags
R1 = Window object id
R2 = 448
R3 = Gadget component id
R4 = pointer to text string to use

**On exit**

R1-R9 preserved

**Use**

This method sets the text string shown in a display field. The change is immediately visible if the parent dialogue box is currently on the screen.

**C veneer**

```
extern _kernel_oserror *displayfield_set_value ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId display_field,
                                                 char *text
                                               );
```

## DisplayField_GetValue 449

**On entry**

R0 = flags
R1 = Window object id
R2 = 449
R3 = Gadget component id
R4 = pointer to buffer
R5 = size of buffer

**On exit**

R5 = size of buffer required else (if R4 was 0)
buffer pointed at by R4 contains text
R5 holds number of bytes written to buffer

**Use**

This method returns the text string shown in a display field.

**C veneer**

```
extern _kernel_oserror *displayfield_get_value ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId display_field,
                                                 char *buffer,
                                                 int buff_size,
                                                 int *nbytes
                                               );
```

### DisplayField_SetFont 450

**On entry**

R0 = flags
R1 = Window object id
R2 = 450
R3 = Gadget component id
R4 = pointer to font name to use
R5 = width in 16ths of a point
R6 = height in 16ths of a point

**On exit**

R1-R9 preserved

**Use**

This method makes the display field use an anti-aliased font. If the font name is NULL, then the field will use system font.

**C veneer**

```
extern _kernel_oserror *displayfield_set_font ( unsigned int flags,
                                                ObjectId window,
                                                ComponentId display_field,
                                                char *font_name,
                                                int width,
                                                int height
                                              );
```

## Display field templates

| Field | Size in bytes | Type |
|---|---|---|
| text | 4 | MsgReference |
| max_text_len | 4 | word |

## Draggable gadgets

A draggable gadget consists of a sprite, text or text&sprite which appears in a dialogue box, and can be dragged using the mouse. When the drag occurs, if this is a sprite or text&sprite draggable, then the Toolbox will use the standard CMOS bit to decide whether to do a ‘solid’ drag or a ‘dotted line’ drag.

Solid dragging makes use of the DragAnObject module allowing both text and sprite to be dragged (unlike DragASprite).

If it is a sprite draggable gadget, then the sprite used can be set and read dynamically using the Draggable_SetSprite/Draggable_GetSprite methods.

If it is a text draggable gadget, then the text used can be set and read dynamically using the Draggable_SetText/Draggable_GetText methods.

With a draggable of type click or doubleclick, a clicks or double click on the gadget will be returned as a Wimp mouse click event, but the toolbox id block will be updated to reflect the component and window (i.e. no special toolbox event is returned).

When the user begins to drag a draggable, the client can choose to receive a Draggable_DragStarted Toolbox event. When the drag ends, the client will always receive a Draggable_DragEnded Toolbox event.

Bits in the flags word for a draggable have the following meaning:

| Bit | Meaning |
|---|---|
| 0 | warn of drag start using Draggable_DragStarted |
| 1 | draggable contains a sprite |
| 2 | draggable contains text |
| 3-5 | Draggable type:<br>0 ⇒ drag only<br>1 ⇒ click, drag, doubleclick<br>2 ⇒ click selects, doubleclick, drag |
| 6 | deliver drag ended events as Toolbox id's rather than Wimp windows (if possible) |
| 7 | dragged object has a drop shadow (if solid) |
| 8 | dragged object is not dithered (if solid) |

## Draggable methods

### Draggable_SetSprite 640

**On entry**

R0 = flags
R1 = Window object id
R2 = 640
R3 = Gadget component id
R4 = pointer to sprite name to use

**On exit**

R1-R9 preserved

**Use**

This method sets the name of the sprite which will be used for this draggable.

**C veneer**

```
extern _kernel_oserror *draggable_set_sprite ( unsigned int flags,
                                               ObjectId window,
                                               ComponentId draggable,
                                               char *sprite_name
                                             );
```

## Draggable_GetSprite 641

**On entry**

R0 = flags
R1 = Window object id
R2 = 641
R3 = Gadget component id
R4 = pointer to buffer (or 0)
R5 = size of buffer to hold sprite name

**On exit**

R5 = size of buffer required for message text (if R4 was 0)
else buffer pointed at by R4 holds sprite name
R5 holds number of bytes written to buffer

**Use**

This method returns the name of the sprite which is currently being used for this draggable.

**C veneer**

```
extern _kernel_oserror *draggable_get_sprite ( unsigned int flags,
                                               ObjectId window,
                                               ComponentId draggable,
                                               char *buffer,
                                               int buff_size,
                                               int *nbytes
                                             );
```

## Draggable_SetText 642

**On entry**

R0 = flags
R1 = Window object id
R2 = 642
R3 = Gadget component id
R4 = pointer to text to use

**On exit**

R1-R9 preserved

**Use**

This method sets the text which will be displayed in this draggable.

**C veneer**

```
extern _kernel_oserror *draggable_set_text ( unsigned int flags,
                                             ObjectId window,
                                             ComponentId draggable,
                                             char *text
                                           );
```

## Draggable_GetText 643

**On entry**

R0 = flags
R1 = Window object id
R2 = 643
R3 = Gadget component id
R4 = pointer to buffer
R5 = size of buffer

**On exit**

R5 = size of buffer required (if R4 was 0)
else buffer pointed at by R4 holds text
R5 holds number of bytes written to buffer

**Use**

This method returns the text which is currently being used for this draggable.

**C veneer**

```
extern _kernel_oserror *draggable_get_text ( unsigned int flags,
                                             ObjectId window,
                                             ComponentId draggable,
                                             char *buffer,
                                             int buff_size,
                                             int *nbytes
                                           );
```

## Draggable_SetState 644

**On entry**

R0 = flags
R1 = Window object id
R2 = 644
R3 = Gadget component id
R4 = state (0 ⇒ deselected, 1 ⇒ selected).

**On exit**

R1-R9 preserved

**Use**

This method sets the Draggable's state to either selected or deselected.

**C veneer**

```
extern _kernel_oserror *draggable_set_state ( unsigned int flags,
                                              ObjectId window,
                                              ComponentId draggable,
                                              int state
                                            );
```

## Draggable_GetState 645

**On entry**

R0 = flags
R1 = Window object id
R2 = 645
R3 = Gadget component id

**On exit**

R0 = state

**Use**

This method returns the Draggables' state (0 ⇒ deselected, 1 ⇒ selected).

**C veneer**

```
extern _kernel_oserror *draggable_get_state ( unsigned int flags,
                                              ObjectId window,
                                              ComponentId draggable,
                                              int *state
                                            );
```

## Draggable Toolbox events

### Draggable_DragStarted (0x82887)

**Block**

| | |
|---|---|
| + 8 | 0x82887 |
| + 12 | flags<br>bit 0 means Adjust is held down<br>bit 1 will be 0<br>bit 2 means Select is held down<br>bit 3 means Shift is held down<br>bit 4 means Ctrl is held down |

**Use**

This Toolbox event is raised when the user starts a drag of a draggable gadget.

**C data type**

```
typedef struct
{
  ToolboxEventHeader    hdr;
} DraggableDragStartedEvent;
```

### Draggable_DragEnded (0x82888)

**Block**

+ 8 0x82888
+ 12 flags:
bit 0 clear then:
+16 Wimp window handle of end of drag
+ 20 Wimp icon handle of end of drag
or bit 0 set:
+16 Window id of end of drag
+20 component id of end of drag
+24 destination x coordinate of mouse pointer
+28 destination y coordinate of mouse pointer

**Use**

This Toolbox event is raised when the user ends a drag of a draggable gadget. By setting bit 6 when the draggable is created it is possible to receive events in terms of window object ids and gadget component ids. If the drag ended over a non-toolbox window (or bit 6 was zero) then Wimp handles are returned.

**C data type**

```
typedef struct
{
  ToolboxEventHeader  hdr;
  int                 window_handle;
  int                 icon_handle;
  int                 x;
  int                 y;
} DraggableDragEndedEvent;
```

## Draggable templates

| Field | Size in bytes | Type |
|---|---|---|
| text | 4 | MsgReference |
| max_text_len | 4 | word |
| sprite | 4 | StringReference |
| max_sprite_len | 4 | word |

## Labels

A label consists of some explanatory text which appears in a dialogue box. The client application can choose whether the bounding box of the label is shown by a visible box or not.

- a label contains text, which is unchangeable at run-time
- a label can be right-justified, left-justified, or centred, as indicated by its flags word.

Bits in the flags word for a label have the following meaning:

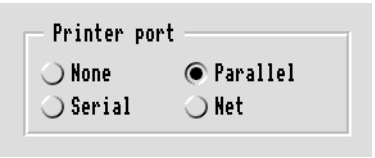
| Bit | Meaning |
|---|---|
| 0 | omit bounding box |
| 1-2 | justification:<br>0 ⇒ left-justified<br>1 ⇒ right-justified<br>2 ⇒ centred |

## Label templates

| Field | Size in bytes | Type |
|---|---|---|
| label | 4 | MsgReference |

## Labelled boxes

A labelled box gadget is used for collecting together a set of related items:




The box has a label which can be either text or a sprite, and this label will appear at the top left hand corner of the box (a bit in the flags word for the gadget indicates whether text or a sprite is to be used). ResEd creates labelled boxes with bit 30 set so that they are created behind other gadgets.

There are no Toolbox events or methods associated with a labelled box.

Bits in the flags word for a labelled box have the following meaning:

| Bit | Meaning |
|---|---|
| 0 | labelled box has a sprite label (default is text) |
| 1 | in the case of a sprite label, the icon is filled if this bit is set, otherwise it is unfilled. This is because certain sprites will sufficiently obscure the border, and may be masked so should allow the tile sprite to show through. |

## Labelled box templates

| Field | Size in bytes | Type |
|---|---|---|
| label | 4 | MsgReference or StringReference |

## Number ranges

A number range is a gadget used to display one of a range of possible integer or fixed point values. The value is shown in a display area, which can either be writable (in which case a writable field is used) or not writable (in which case a display field is used). It is also possible to create a Number Range where there is no display area.

The value which the client gives to a Number Range Gadget (and which it receives back) is a signed integer, to which a ‘precision’ will be applied. The precision is essentially the power of 10 by which the value should be divided, and the number of places which will be shown after the decimal point. For example to get the value 3.42 displayed in a Number Range the client would pass the value 342 with a precision of 2. Normally the precision of a Number Range is specified when the Gadget is created, but it can be set and read at run-time using the NumberRange_SetBounds and NumberRange_GetBounds methods. A Number Range can be made to display merely integer values by specifying a precision of 0. The maximum precision is 10, i.e. there can be up to ten digits after the decimal point.

The value displayed in a number range gadget is set using the NumberRange_SetValue method. The value passed is an integer which will be divided by 10^precision and will have precision digits after the decimal point. The value of a number range is read using the NumberRange_GetValue method; this value is an integer which should be divided by 10^precision to get its real equivalent. A number range has a lower and upper bound which constrains the values to which it can be set; these bounds are in ‘integer’ terms (i.e. before the precision has been applied). For example if a number range gadget has a precision of 3, and the client wishes to have a lower bound of 1.000 and an upper bound of 4.999, then the lower and upper bounds of the gadget should be set to 1000 and 4999 respectively.

A number range can also be given a step size. The step size is expressed in integer terms (i.e. before the precision is applied). For example if a number range gadget has a precision of 2, then setting a step size of 5 will result in a ‘real’ step size of 0.05. The bounds and step size can be set and read using the NumberRange_SetBounds and NumberRange_GetBounds methods.

A number range can also have a pair of adjuster arrows placed 8 OS Units to the right of its display area (either the writable or display field). When the user clicks on these arrows, the value of the number range is either decremented or incremented by its step size, subject to its lower and upper bounds (and displayed using its precision).

A number range can also have an associated slider. The slider is like a slider gadget, except that it can only be positioned relative to the Number Range's display area. The possible positionings are:

- a horizontal slider 8 OS Units to the right of the display area
- a horizontal slider 8 OS Units to the left of the display area.

When both a slider and adjusters are requested, then the adjusters appear at either end of the slider, rather than the positioning outlined above.

If the Number Range is writable, then the underlying Writable Field is given a validation string which will only permit input of numeric digits (0-9), the decimal point character for the current territory (unless the precision field is 0) and where applicable the minus sign. It also has ‘before’ and ‘after’ values which are used to move the caret in the same way as described for Writable Fields. Another Writable may reference the component id of a Number Range in its before and after fields.

Whenever the value changes in a number range gadget, the client is informed of the change via an NumberRange_ValueChanged Toolbox event, if it has set the appropriate bit in the gadget’s flags word.

Included in the definition of the number range is the length of the display field in OS Units (display_length as shown in *Number range templates* on page 375). This is ignored if there is no slider.

Bits in the flags word for a number range gadget have the following meanings:

| **Bit** | **Meaning when set** |
|---|---|
| 0 | inform client of value changes using NumberRange_ValueChanged |
| 2 | writable (default is read-only display) |
| 3 | no display area |
| 4 | has adjuster arrows |
| 5-7 | slider type:<br>**value** **meaning**<br>0 ⇒ no slider<br>1 ⇒ slider to the right of the display area<br>2 ⇒ slider to the left of the display area |
| 8-9 | justification:<br>0 ⇒ left-justified<br>1 ⇒ right-justified<br>2 ⇒ centred |
| 12-15 | desktop colour of slider bar |
| 16-19 | desktop colour of slider background |

Note: slider colours are in the same flag position as a Slider Gadget.

## Number range methods

### NumberRange_SetValue 832

**On entry**

R0 = flags
R1 = Window object id
R2 = 832
R3 = Gadget component id
R4 = new value

**On exit**

R1-R9 preserved

**Use**

This method sets the value displayed in the number range's display area, subject to its bound constraints. The value will be displayed taking into account its precision.

**C veneer**

```
extern _kernel_oserror *numberrange_set_value ( unsigned int flags,
                                                ObjectId window,
                                                ComponentId number_range,
                                                int value
                                              );
```

## NumberRange_GetValue 833

**On entry**

R0 = flags
R1 = Window object id
R2 = 833
R3 = Gadget component id

**On exit**

R0 holds current value

**Use**

This method returns the value of the number range. Note that this is the integer form of what is actually displayed in the display area (i.e. not taking ‘precision’ into account).

**C veneer**

```
extern _kernel_oserror *numberrange_get_value ( unsigned int flags,
                                                ObjectId window,
                                                ComponentId number_range,
                                                int *value
                                              );
```

## NumberRange_SetBounds 834

**On entry**

R0 = flags
    bit 0 set means change the lower bound
    bit 1 set means change the upper bound
    bit 2 set means change the step size
    bit 3 set means change the precision
R1 = Window object id
R2 = 834
R3 = Gadget component id
R4 = new lower bound
R5 = new upper bound
R6 = new step size
R7 = precision

**On exit**

R1-R9 preserved

**Use**

This method is used to set the lower and upper bounds, the step size and the precision of the number range. Note that the bounds and step size are expressed in terms of an integer before they are transformed using the precision value.

**C veneer**

```
extern _kernel_oserror *numberrange_set_bounds ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId number_range,
                                                 int lower_bound,
                                                 int upper_bound,
                                                 int step_size,
                                                 int precision
                                               );
```

## NumberRange_GetBounds 835

**On entry**

R0 = flags
    bit 0 set means return the lower bound
    bit 1 set means return the upper bound
    bit 2 set means return the step size
    bit 3 set means return the precision
R1 = Window object id
R2 = 835
R3 = Gadget component id

**On exit**

R0 = lower bound
R1 = upper bound
R2 = step size
R3 = precision

**Use**

This method returns the lower and upper bounds, the step size and the precision of the number range, depending on the setting of the appropriate flags bits. Note that the bounds and step size are expressed in terms of an integer before they are transformed using the precision value.

**C veneer**

```
extern _kernel_oserror *numberrange_get_bounds ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId number_range,
                                                 int *lower_bound,
                                                 int *upper_bound,
                                                 int *step_size,
                                                 int *precision
                                               );
```

## NumberRange_GetComponents 836

**On entry**

R0 = flags
bit 0 set means return the numerical field
bit 1 set means return the left adjuster
bit 2 set means return the right adjuster
bit 3 set means return the slider
R1 = Window object id
R2 = 836
R3 = Gadget component id

**On exit**

R0 = numeric id
R1 = left adjuster id
R2 = right adjuster id
R3 = slider id

**Use**

This method returns the component ids of the gadgets that make up the number range depending on which flag bits are set. Note that the numeric id will be the component id of the Display Field or Writable, dependent on how the Gadget was created.

**C veneer**

```
extern _kernel_oserror *numberrange_get_components ( unsigned int flags,
                                                     ObjectId window,
                                                     ComponentId number_range,
                                                     ComponentId *numeric_field,
                                                     ComponentId *left_adjuster,
                                                     ComponentId *right_adjuster,
                                                     ComponentId *slider
                                                   );
```

## Number range Toolbox events

### NumberRange_ValueChanged (0x8288d)

**Block**

+ 8 0x8288d
+ 16 new value shown in display area

**Use**

This Toolbox event is raised when the value of the Number Range has changed.

**C data type**

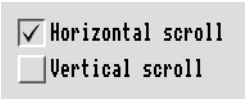
```
typedef struct
{
  ToolboxEventHeader  hdr;
  int                 new_value;
} NumberRangeValueChangedEvent;
```

## Number range templates

| Field | Size in bytes | Type |
|---|---|---|
| lower_bound | 4 | word |
| upper_bound | 4 | word |
| step_size | 4 | word |
| initial_value | 4 | word |
| precision | 4 | word |
| before | 4 | word |
| after | 4 | word |
| display_length | 4 | word |

## Option buttons

An option button is used to indicate whether a particular option has been chosen or not (e.g. case-sensitive in a Find dialogue box). It has two states – on and off:




Such a gadget is displayed with a standard option icon, together with a textual label; the textual label can be read and set at run-time using the OptionButton_SetLabel and OptionButton_GetLabel methods.

The on/off state of the option button can be set and read using the OptionButton_SetState/OptionButton_GetState methods.

If bit zero of the flags is set, then whenever the state of the Option Button changes, an OptionButton_StateChanged event is raised, with the flags word indicating which mouse button was used. The client can supply an alternative Toolbox Event code in the template description for the Option Button, and can set and read this event code at run-time using the OptionButton_SetEvent and OptionButton_GetEvent methods.

Bits in the flags word for Option Button have the following meaning:

| Bit | Meaning |
|---|---|
| 0 | generate a OptionButton_StateChanged when user clicks. |
| 2 | when set, this means that the Option Button is ‘On’ when first created. |

## Option button methods

### OptionButton_SetLabel 192

**On entry**

R0 = flags
R1 = Window object id
R2 = 192
R3 = Gadget component id
R4 = pointer to string giving label to use

**On exit**

R1-R9 preserved

**Use**

This method sets the label which will be used for this option button.

**C veneer**

```
extern _kernel_oserror *optionbutton_set_label ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId option_button,
                                                 char *label
                                               );
```

## OptionButton_GetLabel 193

**On entry**

R0 = flags
R1 = Window object id
R2 = 193
R3 = Gadget component id
R4 = pointer to buffer
R5 = size of buffer

**On exit**

R5 = size of buffer required to hold label (if R4 was 0)
else buffer pointed at by R4 holds label
R5 holds number of bytes written to buffer

**Use**

This method returns the label which is currently displayed for this option button.

**C veneer**

```
extern _kernel_oserror *optionbutton_get_label ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId option_button,
                                                 char *buffer,
                                                 int buff_size,
                                                 int *nbytes
                                               );
```

## OptionButton_SetEvent 194

**On entry**

R0 = flags
R1 = Window object id
R2 = 194
R3 = Gadget component id
R4 = Toolbox event code

**On exit**

R1-R9 preserved

**Use**

This method sets the Toolbox event which will be raised when the state of this option button changes. The rest of the Toolbox event block remains the same as in OptionButton_StateChanged.

**C veneer**

```
extern _kernel_oserror *optionbutton_set_event ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId option_button,
                                                 int event
                                               );
```

## OptionButton_GetEvent 195

**On entry**

R0 = flags
R1 = Window object id
R2 = 195
R3 = Gadget component id

**On exit**

R0 holds Toolbox event code.

**Use**

This method returns the Toolbox event which will be raised when this option button's state changes.

**C veneer**

```
extern _kernel_oserror *optionbutton_get_event ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId option_button,
                                                 int *event
                                               );
```

## OptionButton_SetState 196

**On entry**

R0 = flags
R1 = Window object id
R2 = 196
R3 = Gadget component id
R4 = state (0 ⇒ off, 1 ⇒ on)

**On exit**

R1-R9 preserved

**Use**

This method sets the option button's state to on or off.

**C veneer**

```
extern _kernel_oserror *optionbutton_set_state ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId option_button,
                                                 int state
                                               );
```

## OptionButton_GetState 197

**On entry**

R0 = flags
R1 = Window object id
R2 = 197
R3 = Gadget component id

**On exit**

R0 = state

**Use**

This method returns the option button's state (0 ⇒ off, 1 ⇒ on).

**C veneer**

```
extern _kernel_oserror *optionbutton_get_state ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId option_button,
                                                 int *state
                                               );
```

## Option button Toolbox events

### OptionButton_StateChanged (0x82882)

**Block**

+8 0x82882
+ 12 flags
bits 0, 1 and 2 show how the activation was done:
bit 0 set means Adjust was held down
bit 1 reserved
bit 2 set means Select was held down
+ 16 new state (0 ⇒ off, 1 ⇒ on)

**Use**

This Toolbox event is raised when the state of an option button changes, and the client has not specified an event to be associated with this change.

The returned flags word indicates which mouse button was used to select the button.

**C data type**

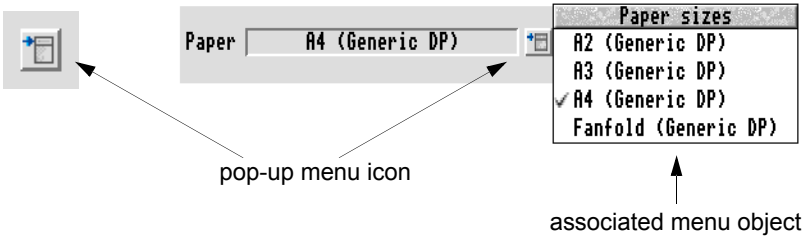
```
typedef struct
{
  ToolboxEventHeader hdr;
  int                new_state;
} OptionButtonStateChangedEvent;
```

## Option button templates

| Field | Size in bytes | Type |
|---|---|---|
| flags | 4 | word |
| label | 4 | MsgReference |
| max_label_len | 4 | word |
| event | 4 | word |

## Pop-up menus

A pop-up menu gadget will be displayed as a ‘menu-arrow’ icon, and its associated Menu object will be displayed when a mouse button is clicked over this icon:




The Menu to be displayed can be set and read dynamically at run-time using the PopUp_SetMenu and PopUp_GetMenu methods. It can also be done with ResEd.

If the appropriate bit is set in the flags word, then a PopUp_AboutToBeShown Toolbox event is delivered before the associated pop-up Menu is shown. This allows the client to build a new Menu object and associate it with the pop-up using PopUp_SetMenu.

Note that Menu ‘hits’ will be reported for the Menu object, and not for the pop-up gadget. The Menu will have as its parent, the dialogue box in which the pop-up exists, and the pop-up itself as the parent component. Note also that the associated pop-up Menu may also have its flags word bit set which requests a warning before it is shown; this event will be delivered after the PopUp_AboutToBeShown event.

Bits in the flags word for a pop-up Menu have the following meaning:

| Bit | Meaning |
|---|---|
| 0 | warn using PopUp_AboutToBeShown before the associated menu is shown. |

## Pop-up menu methods

### PopUp_SetMenu 704

**On entry**

R0 = flags
R1 = Window object id
R2 = 704
R3 = Gadget component id
R4 = object id of Menu to use

**On exit**

R1-R9 preserved

**Use**

This method sets the Menu object which will be shown when the pop-up button is clicked on.

**C veneer**

```
extern _kernel_oserror *popup_set_menu ( unsigned int flags,
                                         ObjectId window,
                                         ComponentId popup,
                                         ObjectId menu
                                       );
```

## PopUp_GetMenu 705

**On entry**

R0 = flags
R1 = Window object id
R2 = 705
R3 = Gadget component id

**On exit**

R0 = Menu object id

**Use**

This method returns the object id of the Menu which will be shown when the pop-up button is clicked on.

**C veneer**

```
extern _kernel_oserror *popup_get_menu ( unsigned int flags,
                                         ObjectId window,
                                         ComponentId popup,
                                         ObjectId *menu
                                       );
```

## Pop-up menu Toolbox events

### PopUp_AboutToBeShown (0x8288b)

**Block**

+ 8 0x8288b
+ 16 object id of Menu object which will be shown
(note that the 'self' id and component fields will refer to the
parent Window's object id and the PopUp's component id respectively)

**Use**

This Toolbox event is raised when the user has clicked on a pop-up button. The Menu is actually shown on the next call to Wimp_Poll.

**C data type**

```
typedef struct
{
  ToolboxEventHeader  hdr;
  ObjectId            menu_id;
}PopUpAboutToBeShownEvent;
```

## Pop-up menu templates

| Field | Size in bytes | Type |
|---|---|---|
| menu | 4 | StringReference |

## Radio buttons

A radio button is used for making a single choice from a set of options, and a number of radio buttons are normally used in a ‘group’. The group to which a radio button belongs is determined by the radio button's ‘group number’.

A radio button is displayed as a standard radio icon, together with a text label. The label for a radio button can be set and read using the RadioButton_SetLabel and RadioButton_GetLabel methods.

A radio button has two states: ‘On’ and ‘Off”. Only one radio button in a group is in the on state at any one time. When the user clicks on a radio button its state is set to on.


Whenever the state of a radio button changes, a RadioButton_StateChanged event is raised, with the flags word indicating which mouse button was used, if the appropriate bit was set in the flags word for the radio button, requesting that a RadioButton_StateChanged event is generated. The client can supply an alternative Toolbox event code in the template description for the radio button, and can set and read this event code at run-time using the RadioButton_SetEvent and RadioButton_GetEvent methods.

Bits in the flags word for a radio button have the following meaning:

| Bit | Meaning |
|---|---|
| 0 | generate a RadioButton_StateChanged when user clicks |
| 2 | when set, means that the radio button is On when first created |

## Radio button methods

### RadioButton_SetLabel 384

**On entry**

R0 = flags
R1 = Window object id
R2 = 384
R3 = Gadget component id
R4 = pointer to string giving label to use

**On exit**

R1-R9 preserved

### Use

This method sets the label which will be used for this radio button.

### C veneer

```
extern _kernel_oserror *radiobutton_set_label ( unsigned int flags,
                                                ObjectId window,
                                                ComponentId radio_button,
                                                char *label
                                              );
```

## RadioButton_GetLabel 385

**On entry**

R0 = flags
R1 = Window object id
R2 = 385
R3 = Gadget component id
R4 = pointer to buffer
R5 = size of buffer

**On exit**

R5 = size of buffer required to hold label (if R4 was 0)
else buffer pointed at by R4 holds label
R5 holds number of bytes written to buffer

**Use**

This method returns the label which is currently displayed for this radio button.

**C veneer**

```
extern _kernel_oserror *radiobutton_get_label ( unsigned int flags,
                                                ObjectId window,
                                                ComponentId radio_button,
                                                char *buffer,
                                                int buff_size,
                                                int *nbytes
                                              );
```

## RadioButton_SetEvent 386

**On entry**

R0 = flags
R1 = Window object id
R2 = 386
R3 = Gadget component id
R4 = Toolbox event code

**On exit**

R1-R9 preserved

**Use**

This method sets the Toolbox event which will be raised when the state of the radio button changes. The rest of the Toolbox event block will be the same as for the RadioButton_StateChanged Toolbox event.

**C veneer**

```
extern _kernel_oserror *radiobutton_set_event ( unsigned int flags,
                                                ObjectId window,
                                                ComponentId radio_button,
                                                int event
                                              );
```

## RadioButton_GetEvent 387

**On entry**

R0 = flags
R1 = Window object id
R2 = 387
R3 = Gadget component id

**On exit**

R0 holds Toolbox event code

**Use**

This method returns the Toolbox event which will be raised when this radio button's state changes.

**C veneer**

```
extern _kernel_oserror *radiobutton_get_event ( unsigned int flags,
                                                ObjectId window,
                                                ComponentId radio_button,
                                                int *event
                                              );
```

## RadioButton_SetState 388

**On entry**

R0 = flags
R1 = Window object id
R2 = 388
R3 = Gadget component id
R4 = state (0 ⇒ Off, 1 ⇒ On)

**On exit**

R1-R9 preserved

**Use**

This method sets the state of the radio button to On or Off. When a button which is Off is set to On, the button which was previously On is set to Off. If by setting the radio button to Off, this would result in no button being On in the group, then an error is returned.

**C veneer**

```
extern _kernel_oserror *radiobutton_set_state ( unsigned int flags,
                                                ObjectId window,
                                                ComponentId radio_button,
                                                int state
                                              );
```

## RadioButton_GetState 389

**On entry**

R0 = flags
R1 = Window object id
R2 = 389
R3 = Gadget component id

**On exit**

R0 = state (0 ⇒ Off, 1 ⇒ On)
R1 = component id of radio button which is On in the group

**Use**

This method returns the state of the given radio button.

The client can determine which radio button is On in a group by calling this method for any one button in the group, since the component id of the On button is also returned (in R1).

**C veneer**

```
extern _kernel_oserror *radiobutton_get_state ( unsigned int flags,
                                                ObjectId window,
                                                ComponentId radio_button,
                                                int *state,
                                                Component Id *selected
                                              );
```

## Radio button Toolbox events

### RadioButton_StateChanged (0x82883)

**Block**

+ 8 0x82883
+ 12 flags
bits 0, 1 and 2 show how the activation was done:
bit 0 set means Adjust was held down
bit 1 is reserved
bit 2 set means Select was held down
+16 state (0 ⇒ Off, 1 ⇒ On)
+20 component id of the radio button within the group which was On before this state change

**Use**

This Toolbox event is raised when the state of a radio button changes, and the client has not specified an event to be associated with this change.

The returned flags word indicates which mouse button was used to select the radio button.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  int                state;
  ComponentId        old_on_button;
} RadioButtonStateChangedEvent;
```

## Radio button templates

| Field | Size in bytes | Type |
|---|---|---|
| group_number | 4 | word |
| label | 4 | MsgReference |
| max_label_len | 4 | word |
| event | 4 | word |

## Sliders

A slider gadget is used to display a ‘bar’, which may be draggable by the user, displayed in a ‘well’. Whether the slider is draggable or not is indicated by its flags word:


By setting a bit in the slider’s flags word the client can request that all changes in the slider’s value are returned as the bar is dragged. Alternatively it may request to receive value changes only when the bar dragging stops (i.e. when the user releases the mouse button). Such changes are reported via the Slider_ValueChanged Toolbox event.

A slider is specified as either being ‘vertical’ or ‘horizontal’.

A slider has associated with it an initial value, a minimum value, a maximum value, and a step size. If the slider is draggable (indicated by a flags bit), then when the user drags the bar with the mouse, the bar moves a number of pixels commensurate with the step size, and the bounding box of the slider.

The maximum and minimum values and the step size can be set and read dynamically using the Slider_SetBound/Slider_GetBound methods.

A Slider also has associated with it, the colour used for its ‘bar’ – this is a Desktop colour. This is normally specified in the resource file, but can be set and read dynamically using the Slider_SetColour/Slider_GetColour methods.

The current value of the slider can be set and read using the Slider_SetValue/Slider_GetValue methods.

Bits in the flags word for a slider have the following meaning:

| Bit | Meaning |
|---|---|
| 0 | if set then deliver value changes when user clicks/drags |
| 1 | if set then deliver value changes constantly whilst dragging else just at start/end |
| 3 | if set means slider is vertical (default is horizontal) |
| 4 | if set then bar is draggable/clickable |
| 12-15 | the desktop colour of the bar |
| 16-19 | the desktop colour of the background |

## Slider methods

### Slider_SetValue 576

**On entry**

R0 = flags
R1 = Window object id
R2 = 576
R3 = Gadget component id
R4 = integer value

**On exit**

R1-R9 preserved

**Use**

This method sets the value of a slider. The slider's bar is changed accordingly.

**C veneer**

```
extern _kernel_oserror *slider_set_value ( unsigned int flags,
                                           ObjectId window,
                                           ComponentId slider,
                                           int value
                                         );
```

## Slider_GetValue 577

### On entry

R0 = flags
R1 = Window object id
R2 = 577
R3 = Gadget component id

### On exit

R0 = slider's value

### Use

This method returns the value of a slider.

### C veneer

```
extern _kernel_oserror *slider_get_value ( unsigned int flags,
                                           ObjectId window,
                                           ComponentId slider,
                                           int *value
                                         );
```

## Slider_SetBound 58

**On entry**

R0 = flags
    bit 0 set means set upper bound
    bit 1 set means set lower bound
    bit 2 set means set step size
R1 = Window object id
R2 = 578
R3 = Gadget component id
R4 = upper bound
R5 = lower bound
R6 = step size

**On exit**

R1-R9 preserved

**Use**

This method sets the lower bound, upper bound and step size of a slider gadget.

**C veneer**

```
extern _kernel_oserror *slider_set_bound ( unsigned int flags,
                                           ObjectId window,
                                           ComponentId slider,
                                           int upper_bound,
                                           int lower_bound,
                                           int step_size
                                         );
```

## Slider_GetBound 579

**On entry**

R0 = flags
    bit 0 set means return upper bound
    bit 1 set means return lower bound
    bit 2 set means return step size
R1 = Window object id
R2 = 579
R3 = Gadget component id

**On exit**

R0 = upper bound
R1 = lower bound
R2 = step size

**Use**

This method returns the lower bound, upper bound and step size of a slider gadget.

**C veneer**

```
extern _kernel_oserror *slider_get_bound ( unsigned int flags,
                                           ObjectId window,
                                           ComponentId slider,
                                           int *upper_bound,
                                           int *lower_bound,
                                           int *step_size
                                         );
```

## Slider_SetColour 580

**On entry**

R0 = flags
R1 = Window object id
R2 = 580
R3 = Gadget component id
R4 = Desktop colour value for bar
R5 = Desktop colour value for background

**On exit**

R1-R9 preserved

**Use**

This method sets the Desktop colour used in a slider.

**C veneer**

```
extern _kernel_oserror *slider_set_colour ( unsigned int flags,
                                            ObjectId window,
                                            ComponentId slider,
                                            int bar_colour,
                                            int back_colour
                                          );
```

## Slider_GetColour 581

**On entry**

R0 = flags
R1 = Window object id
R2 = 581
R3 = Gadget component id

**On exit**

R0 = Desktop colour value for bar
R1 = Desktop colour value for background

**Use**

This method returns the Desktop colour used in a slider.

**C veneer**

```
extern _kernel_oserror *slider_get_colour ( unsigned int flags,
                                            ObjectId window,
                                            ComponentId slider,
                                            int *bar_colour,
                                            int *back_colour
                                          );
```

## Slider Toolbox events

### Slider_ValueChanged (0x82886)

**Block**

+ 8 0x82886
+ 12 flags:
bits 0 -2:
0 means ‘start of drag or just click’
1 means ‘drag still in progress’
2 means ‘drag has ended’
+ 16 new value of slider.

**Use**

This Toolbox event is raised when the value of the slider has changed. This may be due to an update caused by a user action (e.g. dragging the bar).

**C data type**

```
typedef struct
{
  ToolboxEventHeader  hdr;
  int                 new_value;
} SliderValueChangedEvent;
```

## Slider templates

| Field | Size in bytes | Type |
|---|---|---|
| lower_bound | 4 | word |
| upper_bound | 4 | word |
| step_size | 4 | word |
| initial_value | 4 | word |

## String sets

A string set is a gadget used to display one of an ordered set of text strings.

The string which is shown in the display area is known as the ‘selected string’. The display area can be either writable (in which case a writable field is used) or not writable (in which case a display field is used).

A string set has a pop-up Menu placed 8 OS Units to the right of the display area. The client supplies a set of available strings, and the Toolbox will display the selected string in the string set's display area. The Toolbox will build a Menu on the client's behalf, and display it when the pop-up menu button is clicked. The selected string will be shown as ticked in the Menu, and hits on the Menu will result in the string corresponding to the Menu entry text becoming the selected string.

If the string set is writable, then if the user enters a string which is not in the string set, no entry would be shown as ticked in an associated pop-up Menu.

The set of available strings can be set at run-time using the StringSet_SetAvailable method. The selected string is set and read using the StringSet_SetSelected and StringSet_GetSelected methods.

Whenever the selected string changes in a string set gadget, the client is informed of the change via a StringSet_ValueChanged Toolbox event, if it has set the appropriate bit in the gadget’s flags word.

If a string set is writable, it can also have a set of allowable characters which the user can type into the display area. This is identical to the ‘a’ directive used in a Wimp icon’s validation string.

The set of allowable characters can be set at run-time using the StringSet_SetAllowable method.

In the template description for a writable string set, the client specifies the component ids of any writable fields which come before and after it. These are used to move the caret between writable fields when the user presses the arrow and tab keys. A special value of -1 indicates that there is no writable field before or after this one.

Bits in the flags word for a string set gadget have the following meanings:

| Bit | Meaning |
|---|---|
| 0 | inform client of changes to the selected string using StringSet_ValueChanged |
| 1 | writable (default is read-only display) |
| 3 | inform client just before showing the menu |

| Bit | Meaning |
|---|---|
| 4 | does not have any display field or writable |
| 5-6 | justification:<br>0 ⇒ left-justified<br>1 ⇒ right-justified<br>2 ⇒ centred |

## String set methods

### StringSet_SetAvailable 896

**On entry**

R0 = flags
R1 = Window object id
R2 = 896
R3 = Gadget component id
R4 = pointer to block of contiguous strings which are to be used as the available set of strings

**On exit**

R1-R9 preserved

**Use**

This method is used to set the available set of strings in a string set, and a pop-up menu will be built from them. Strings are separated using a comma (‘,’); a comma must be escaped using the \ character, if the client wishes it to appear in the display area. To get the '\' character itself, '\\' should be used.

Note that there is no StringSet_GetAvailable.

**C veneer**

```
extern _kernel_oserror *stringset_set_available ( unsigned int flags,
                                                  ObjectId window,
                                                  ComponentId string_set,
                                                  char *strings
                                                );
```

## StringSet_SetSelected 898

**On entry**

R0 = flags
bit 0set means index of string is supplied in R4
clear means the string itself is supplied
R1 = Window object id
R2 = 898
R3 = Gadget component id
R4 = pointer to string to be selected or R4 = index of string to be selected

**On exit**

R1-R9 preserved

**Use**

This method sets which string in the string set is selected. The string can either be specified as a text string or as an index into the array of available strings (depending on the setting of bit 0 in the flags word). The selected string is shown in the string set's display area, and will be ticked in the associated pop-up Menu.

**C veneer**

```
extern _kernel_oserror *stringset_set_selected ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId string_set,
                                                 char *string_to_select
                                               );
```

## StringSet_GetSelected 899

**On entry**

R0 = flags
    bit 0 set means return index of selected string
        clear means the string itself is returned
R1 = Window object id
R2 = 899
R3 = Gadget component id
R4 = index of selected string or R4 = pointer to buffer to hold selected string
R5 = size of buffer

**On exit**

R0 = index of selected string (if bit 0 of flags word was set)
    else
        if R4 was 0 then R5 holds size of buffer required
    else
        buffer pointed at by R4 holds selected string
        R5 holds number of bytes written to buffer

**Use**

This method returns the currently selected string for this string set (i.e. the one shown in the display area). This may be either an index into the set of available strings or a buffer containing the string itself. If the selected string is not in the available set (e.g. it has been typed into a writable string set), then the value -1 is returned if an index is requested (by setting bit 0 of the flags word for this call).

**C veneer**

```
extern _kernel_oserror *stringset_get_selected ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId string_set,
                                                 ...
                                               );
```

## StringSet_SetAllowable 900

**On entry**

R0 = flags
R1 = Window object id
R2 = 900
R3 = Gadget component id
R4 = pointer to string giving new set of allowable characters

**On exit**

R1-R9 preserved

**Use**

This method defines the set of allowable characters which can be typed into a writable string set. The set is specified in the same way as a Wimp ‘a’ validation string directive (without including the letter ‘a’).

**C veneer**

```
extern _kernel_oserror *stringset_set_allowable ( unsigned int flags,
                                                  ObjectId window,
                                                  ComponentId string_set,
                                                  char *allowable
                                                );
```

## StringSet_GetComponents 902

**On entry**

R0 = flags
bit 0 set means return the alphanumerical field
bit 1 set means return the popup menu
R1 = Window object id
R2 = 902
R3 = Gadget component id

**On exit**

R0 = alphanumeric id
R1 = popup id

**Use**

This method returns the component ids of the gadgets that make up the string set depending on which flag bits are set. Note that the alphanumeric id will be the component id of the Display Field or Writable, dependent on how the Gadget was created.

**C veneer**

```
extern _kernel_oserror *stringset_get_components ( unsigned int flags,
                                                   ObjectId window,
                                                   ComponentId string_set,
                                                   ComponentId *alphanumeric_field,
                                                   ComponentId *popup_menu
                                                      );
```

## String set Toolbox events

### StringSet_ValueChanged (0x8288e)

**Block**

| | |
|---|---|
| + 8 | 0x8288e |
| + 12 | flags<br>if bit 0 is set, then the text string was too long to fit into the event block |
| + 16... | text string shown in string set's display area (or null string if too long to fit) |

**Use**

This Toolbox event is raised when the value of the string set has changed. If the text string was too long to fit into the event block, then bit 0 of the flags word is set.

**C data type**

```
typedef struct
{
  ToolboxEventHeader  hdr;
  char
string[sizeof(ToolboxEvent)-sizeof(ToolboxEventHeader)];
} StringSetValueChangedEvent;
```

### StringSet_AboutToBeShown (0x8288f)

**Block**

| | |
|---|---|
| + 8 | 0x8288f |

**Use**

This Toolbox event is raised just before the string set's menu is to be shown. This allows the client to make changes to the string set just when it is used, rather than continually.

**C data type**

```
typedef struct
{
  ToolboxEventHeader  hdr;
} StringSetAboutToBeShownEvent;
```

## String set templates

| Field | Size in bytes | Type |
|---|---|---|
| string_set | 4 | MsgReference |
| initial_selected_string | 4 | MsgReference |

| Field | Size in bytes | Type |
|---|---|---|
| max_selected_string_len | 4 | word |
| allowable | 4 | MsgReference |
| max_allowable | 4 | word |
| before | 4 | word |
| after | 4 | word |

## Writable fields

The writable field has a boxed display area in which a text string is displayed and can be edited by the user. The contents of the display area can be set and read using the WritableField_SetValue and WritableField_GetValue methods. The user can click the mouse in a writable field and enter its value from the keyboard:


Whenever the value in a writable field is changed, the client receives a WritableField_ValueChanged Toolbox event, if it has set the appropriate bit in the flags word. This will happen when the user presses a key whilst the caret is in it.

Note that it is possible to get different values from Writable_GetValue on subsequent calls, without receiving a ValueChanged Event in between. This is because the value represents what is actually visible in the gadget.

A writable field can also have a set of allowable characters which the user can type into the display area. This is identical to the ‘a’ directive used in a Wimp icon’s validation string.

The set of allowable characters can be set at run-time using the WritableField_SetAllowable method. To allow all characters, this attribute should be NULL.

In the template description for a writable field, the client specifies the component ids of writable fields which come ‘before’ and ‘after’ it. These are used to move the caret between writable fields when the user presses the arrow and tab keys. A special value of -1 indicates that there is no writable field before or ‘after this one. The exact semantics for the keys are as follows:

| | | |
|---|---|---|
| up-arrow or shift-TAB | ⇒ | move the caret to the writable field before the one which currently has the caret |
| down-arrow or TAB | ⇒ | move the caret to the writable field after the one which currently has the caret |

Bits in the flags word for a writable field have the following meaning:

| Bit | Meaning |
|---|---|
| 0 | inform of value changes using WritableField_ValueChanged |
| 2-3 | justification:<br>0 ⇒ left-justified<br>1 ⇒ right-justified<br>2 ⇒ centred |
| 4 | do not display text, use ‘-’ for each character (password support) |

## Writable field methods

### WritableField_SetValue 512

**On entry**

R0 = flags
R1 = Window object id
R2 = 512
R3 = Gadget component id
R4 = pointer to text string to use

**On exit**

R1-R9 preserved

**Use**

This method sets the text string shown in a writable field. The change is immediately visible if the parent dialogue box is currently on the screen.

**C veneer**

```
extern _kernel_oserror *writablefield_set_value ( unsigned int flags,
                                                  ObjectId window,
                                                  ComponentId writable,
                                                  char *text
                                                );
```

## WritableField_GetValue 513

**On entry**

R0 = flags
R1 = Window object id
R2 = 513
R3 = Gadget component id
R4 = pointer to buffer
R5 = size of buffer

**On exit**

R5 = size of buffer required (if R4 was 0)
else buffer pointed at by R4 contains text
R5 holds number of bytes written to buffer

**Use**

This method returns the text string shown in a writable field.

**C veneer**

```
extern _kernel_oserror *writablefield_get_value ( unsigned int flags,
                                                  ObjectId window,
                                                  ComponentId writable,
                                                  char *buffer,
                                                  int buff_size,
                                                  int *nbytes
                                                );
```

### WritableField_SetAllowable 514

**On entry**

R0 = flags
R1 = Window object id
R2 = 514
R3 = Gadget component id
R4 = pointer to string giving new set of allowable characters

**On exit**

R1-R9 preserved

**Use**

This method defines the set of allowable characters which can be typed into a writable field. The set is specified in the same way as a Wimp ‘a’ validation string directive (without including the letter ‘a’). If the string is NULL, then all characters are allowable.

**C veneer**

```
extern _kernel_oserror *writablefield_set_allowable ( unsigned int flags,
                                                      ObjectId window,
                                                      ComponentId writable,
                                                      char *allowed
                                                    );
```

## WritableField_SetFont 516

**On entry**

R0 = flags
R1 = Window object id
R2 = 516
R3 = Gadget component id
R4 = pointer to font name to use
R5 = width in 16ths of a point
R6 = height in 16ths of a point

**On exit**

R1-R9 preserved

**Use**

This method makes the writable field use an anti-aliased font. If the font name is NULL, then the field will use system font.

**C veneer**

```
extern _kernel_oserror *writablefield_set_font ( unsigned int flags,
                                                 ObjectId window,
                                                 ComponentId writable_field,
                                                 char *font_name,
                                                 int width,
                                                 int height
                                               );
```

## Writable field Toolbox events

### WritableField_ValueChanged (0x82885)

**Block**

| | |
|---|---|
| + 8 | 0x82885 |
| + 12 | flags<br>if bit 0 is set, then the text string was too long to fit into the event block |
| + 16... | text string shown in writable field |

**Use**

This Toolbox event is raised when the value of the writable field has changed. The text string is copied into the event block, and is nul-terminated. If the text string was too long to fit into the event block, then bit 0 of the flags word is set and a null string is supplied.

**C data type**

```
typedef struct
{
  ToolboxEventHeader hdr;
  char
string[sizeof(ToolboxEvent)-sizeof(ToolboxEventHeader)];
} WritableFieldValueChangedEvent;
```

## Writable field templates

| Field | Size in bytes | Type |
|---|---|---|
| text | 4 | MsgReference |
| max_text_len | 4 | word |
| allowable | 4 | MsgReference |
| max_allowable_len | 4 | word |
| before | 4 | word |
| after | 4 | word |

# 17 ResEd

ResEd is the tool used to construct and edit Toolbox resource files. It provides the following:

- A display of the object templates present in the resource file (called the resource file display), each object template being represented by a named icon. You can drag these icons to move and copy object templates between resource file displays (and other co-operating applications).
- A selection of pre-defined object templates for you to drag into a resource file display (this is the standard way to populate a resource file display with object templates).
- A specialised editor to allow you to edit all the various classes of object templates.

To use this chapter you should have a basic understanding of the Toolbox and objects.

## Overview

The process for creating, editing, and saving a resource file can be summarised as follows:

**1** Start ResEd.

**2** Open a new resource file display.

**3** Open an object prototypes display containing pre-defined object templates.

**4** Drag the object templates you require from the object prototypes window into the resource file display.

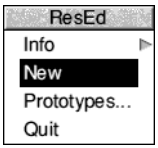
**5** Double-click on an object template to open an editing window for it.

**6** Edit the object templates.

**7** Save the edited object templates into a resource file.

The following section, *Creating and editing a Toolbox resource file*, gives a detailed description of the above process.

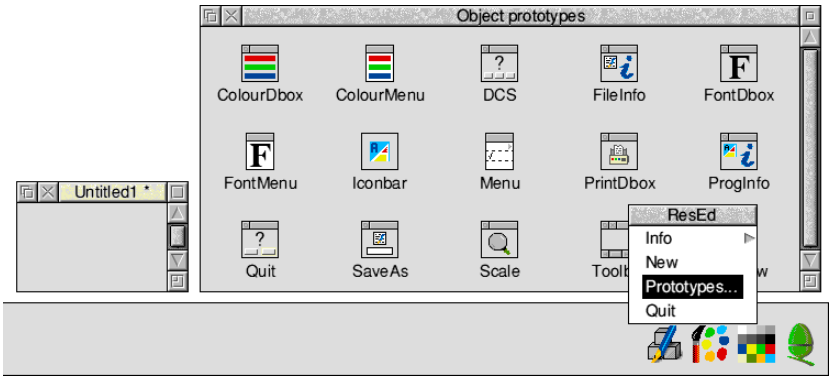
## Creating and editing a Toolbox resource file



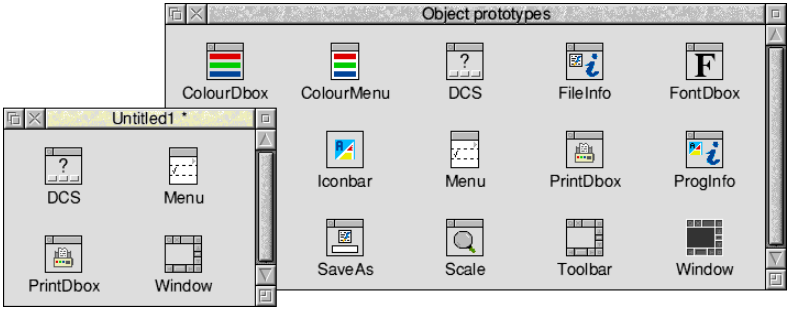

1 Start ResEd in a similar way to other RISC OS applications, by double-clicking on its application icon. It loads and installs an icon on the iconbar.

2 Open a new resource file display by clicking Select on the ResEd iconbar icon or choosing **New** from the ResEd menu. A new, untitled resource file display will appear on the screen.

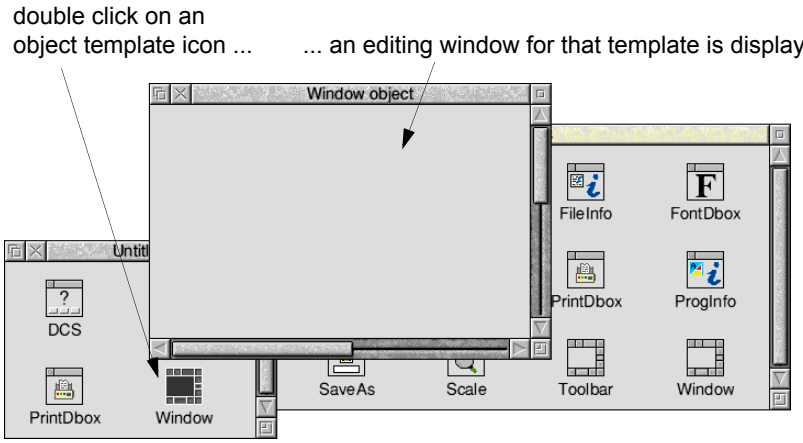
3 The object prototypes window allows you to drag any prototype object template into the resource file display. To open the object prototypes window click Adjust on the iconbar icon or choose **Prototypes...** from the ResEd menu.



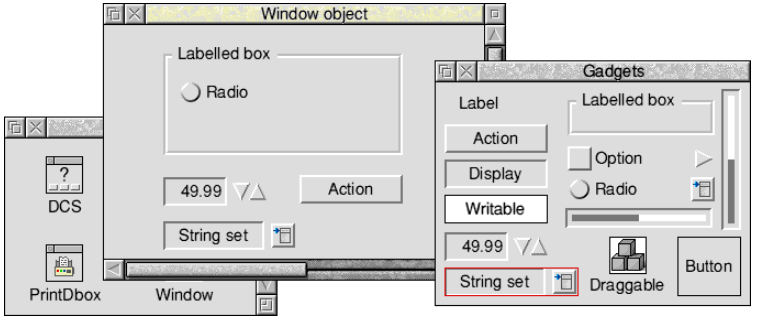

4 Drag one or more object templates from the object prototypes window into the resource file display.



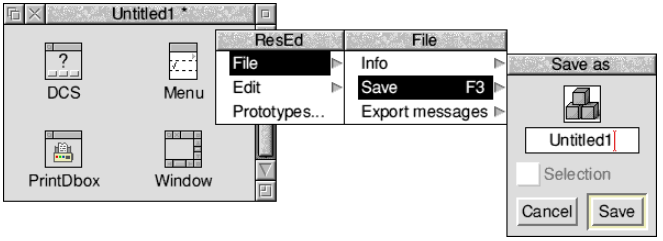
5 To edit a Window object template double-click on its icon in the resource file display. An editing window will appear showing the object template in full:




6 When you have finished editing a window object template, close the editing window using the close icon (some object templates are displayed for editing in dialogue boxes, and you close these by clicking on the **OK** button):




7 When you have finished editing all the object templates you can save them using the **Save** option from the resource file display menu. This leads to a Save as dialogue box, which allows you to save some or all of your object templates.



## Starting ResEd

Start ResEd in a similar way to other RISC OS applications, by double-clicking on its application icon. It loads and installs an icon on the iconbar. It may also be loaded by double-clicking on a file of type `Resource`, in which case the file is loaded and displayed.

Each resource file is displayed in its own resource file display. If you load a file which is already loaded, that file's window is raised to the top of the window stack.

Whenever a resource file is loaded, a corresponding Sprites file is sought in the same directory. If one is found its sprites are loaded with *iconsprites and used when displaying the resources in the resource file display. Sprite files may also be loaded by dragging to the iconbar icon.

### The iconbar icon

The iconbar icon responds to the mouse in the following ways:

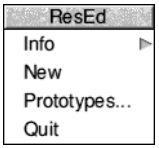
- clicking Select on the icon opens an empty resource file display
- clicking Menu on the icon opens the ResEd Menu
- clicking Adjust on the icon opens the object prototypes window.

Empty resource files are opened with incrementally-unique names (`Untitled1`, `Untitled2` etc). Each one is opened in a slightly different position to the last.

The object prototypes window contains prototype object templates of each class. You can drag these into the resource file display in order to populate it with object templates. The object prototypes window is fully described in *The object prototypes window* on page 421.

#### The iconbar menu

Clicking Menu on the iconbar icon displays the following menu:




**Info** displays an Info dialogue box.

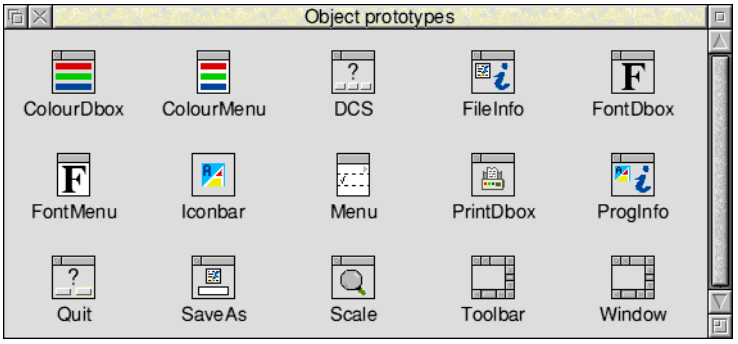
**New** opens an empty, untitled resource file display.

**Prototypes...** opens the object prototypes window (described on page 421).

**Quit** exits the program.

## The object prototypes window

Resource file displays may be populated with object templates by dragging them in from the object prototypes window. The templates are named after the classes they represent. You can copy them into your resource file display by drag and drop, rename them as desired, and then view and edit them by double-clicking on their icons.




The following object templates are available:

Colour DboxColour menuDCS DboxFile Info Dbox

Font DboxFont menuIconbar iconMenu

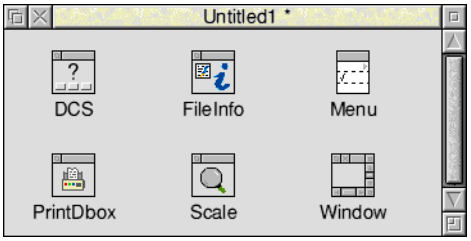
Print DboxProg Info DboxQuit DboxSave As Dbox

Scale DboxToolbarWindow

To open or raise the object prototypes window, choose **Prototypes...** from the iconbar menu or click Adjust on the iconbar icon. The object prototypes window is very similar to an ordinary resource file display, but attempts to move, rename, modify or delete object templates within it are ignored. It is not possible to edit an object template within the object prototypes window; instead you must first drag the object template into a resource file display. The object prototypes window does not have a menu and only Ctrl-Z and Ctrl-A keyboard short-cuts are available.

## The resource file display

The resource file display is Filer-like, in that it contains a grid of icons, one per object template held in the resource file. The sprite associated with each icon is a pictorial clue as to the type of object template that icon represents; each class of object template has its own sprite. The text associated with each icon is the name assigned to that object template.




Icons may be selected, deselected and dragged from one resource file display to another (as in the Filer).

### Editing an object template

To edit an object template, double-click on its icon. A window will then open for that object template. Some common features of editing object templates are described in *Editing object templates in general* on page 425.

For details of editing the individual types of object templates see

- *Editing the Menu class* on page 429
- *Editing a Window object template and gadgets* on page 438
- *Editing other classes* on page 474.

### Copying object templates

You can copy object templates between resource file displays by dragging their icons. You can also make a copy of an object template within one resource file display by using Shift-Drag Select.

### Moving object templates

You can move an object template from one resource file display to another using Shift-Drag Select. This will remove the object template from the source window.

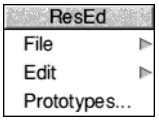
**Note**: Copy or move operations that would result in duplicate names are resolved by the new object templates' names being automatically disambiguated by the addition of a unique numeric suffix (you will be warned if this happens).

If you drag a selection into a different application, the result is the exporting of a resource file containing just the selected object templates. This file is named `Selection`.

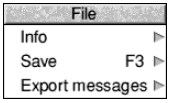
If the resource file display is the target of a drag and drop or DataSave interaction from another application, it checks the file type and rejects the file if not of type Resource or Text (for more information on text files see *Exporting and importing messages* on page 486). Resource files are imported into the resource file display and object template names are disambiguated if necessary, as described above. Importing a file does not alter the filename of the destination resource file display – the name of the incoming file is simply ignored.

## The resource file display menu

Clicking Menu on the resource file display shows the ResEd menu:




### The File menu



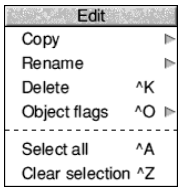

**Info** leads to a File Info dialogue box.

**Save** leads to a Save as dialogue box, which includes a Selection button for saving only the selected object templates.

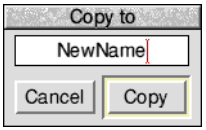
**Export messages** leads to a Save as dialogue box allowing you to produce a text file containing all the user-visible messages for the file (or selection, if **Save selection** is set). The messages may then be edited (typically, translated into a different language) and then re-imported by dropping the file back into the resource file display.

For more information about exporting and importing messages see *Exporting and importing messages* on page 486.

### The Edit menu

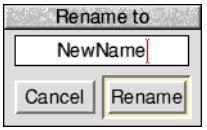



**Copy** (which is shaded unless only one object template is selected) leads to the following dialogue box:



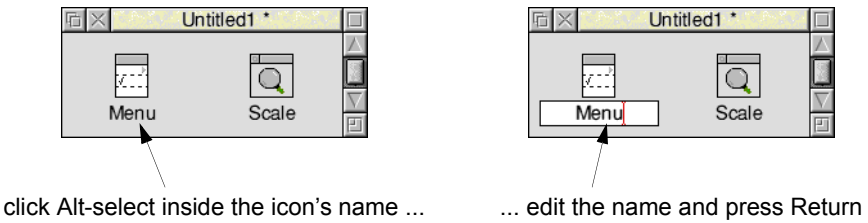

The name field is filled in with the name of the selected object template. To make a copy of the object template in the same file, alter the name and click **Copy**.

**Rename** leads to a dialogue box with a writable icon for entering a new name for the selected object template and a **Rename** button to accept the change:




The writable icon is initially filled in with the current name. When **Rename** is pressed, the object template is renamed unless a name clash would occur, in which case an error message is issued instead.

You can also change an object template's name by clicking Alt-Select inside the icon's name, editing the string and pressing Return:




Pressing Escape or clicking outside the writable icon cancels the rename.

**Delete** deletes all the selected object templates.

**Object flags** allows you to edit the settings of the object flags for the selected object templates. See *The Object flags dialogue box* on page 425 for more details.

**Select all** selects all the object templates in a resource file display.

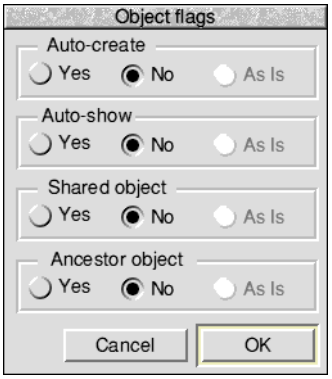
**Clear selection** deselects all the selected object templates.

### Prototypes...

This option displays the object prototypes window.

### The Object flags dialogue box

You can edit most object template data by double-clicking on its icon. There is, however, a 32-bit flags field in the object header. These flags are applicable to all classes of object, and you may view the flags of an individual object template by selecting it and entering the **Object flags** dialogue box. It has the following appearance:




To summarise, the flags are:

| Bit | Meaning when set |
|---|---|
| 0 | create object when resource file is loaded |
| 1 | show object as soon as it is created |
| 2 | object is shared |
| 3 | mark this as an Ancestor object |

If there is one object template selected, or multiple object templates which have identical flag values, the buttons will be set to **Yes** or **No** as appropriate. If there are multiple selected object templates with different flag settings, then the flags which differ will be set to **As Is**, indicating to the user that the flag value differs across the object templates.

You may adjust the settings as required, and on pressing **OK** the new flag values will be applied to the selected object templates. Any flags which are set to **As Is** will not be applied to the selected object templates; each object template will retain its existing value for those flags. So, for example, you could change a number of object templates to be ‘Shared’ without altering their other flags.

## Editing object templates in general

Once you have dragged an object template from the Objects prototype window into the resource file display you can edit it by double-clicking on its icon. You can then edit a properties box for that object template specifying how you want it to appear and behave. All the object properties boxes share the following features.

## Length fields

### Help messages

The Window and Menu object templates, and all gadget templates, include the facility to specify a help message:


If you switch on the **Help text** option you are then able to enter a help message into the associated message field:


By default an asterisk is displayed in the **Length** field. This asterisk ensures that, whatever string you enter into the message field, the exact length of that string (including its terminator) will be passed to the Toolbox.

Alternatively you can manually change the size of the **Length** field to be greater than the length of the help message itself. This is useful if you wish to alter the help message at run-time. If you type a number into the **Length** field directly, then, when you click on **OK**, the size of the **Length** field will be set to the length of the string you entered +1 (unless the number you entered is greater than the length of the string, in which case the number will remain as you entered it).

The following are some points to bear in mind when entering help text:

- If you switch off the **Help text** option then any help message you entered in the associated message field will be removed.
- If you switch on the **Help text** option, but leave the associated message field empty, then the Interactive help window will go blank when the user moves the pointer over the relevant object.

### Other length fields

Some other options in object properties boxes behave in a similar manner to the above; for example, editing the Titles of objects.

## The selection model

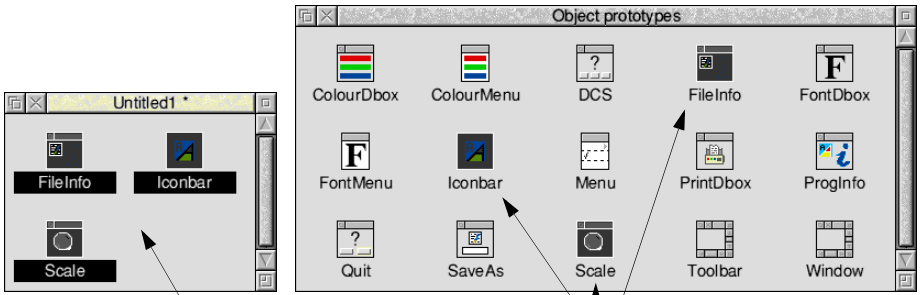
ResEd supports some new selection techniques to improve the way you can manipulate objects and object templates.

### Selection highlighting

ResEd provides two levels of selection with two corresponding types of highlight:

- a full highlight for a selection within a window that has the input focus
- a partial highlight for the previous selection in a deselected window.

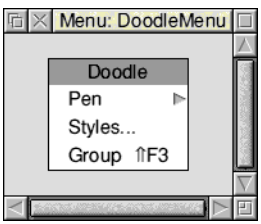
For example, when you select one or more object templates in the object prototypes window and drag them to a resource file display, the original object templates remain partially highlighted. This allows you to return to the object prototypes window and, by clicking on any of the object templates within the original selection, automatically select all of the original selection. For example:




object templates fully highlighted in resource file display

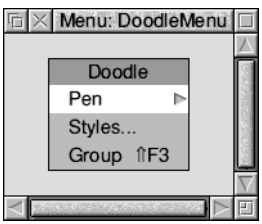
object templates remain partially highlighted in previous window

You can use this additional selection technique throughout ResEd; for example, you can select menu entries when editing a Menu object template, and still retain them as a selection if you temporarily need to edit a different window:




Window has input focus

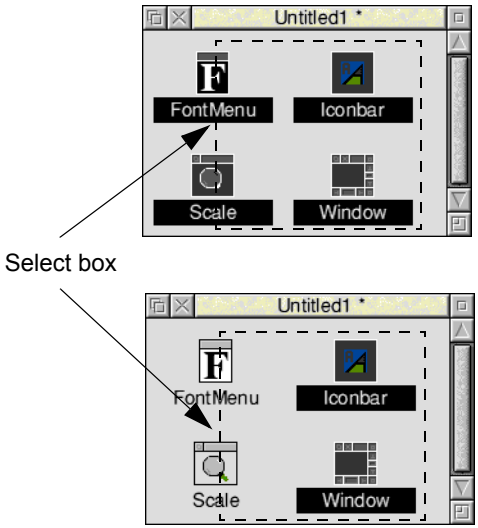

two menu entries selected within the window




menu entries still selected when the window no longer has the input focus

### Box selection

If you use the mouse to drag a Select box around a group of object templates, you can control whether all the objects (even those partly) within the box are selected, or just the ones wholly within the box:




dragging a box around a group of object templates will select any object template partly or wholly within the Select box

dragging a box around a group of object templates while holding down Shift will select only objects wholly within the Select box

Groups of gadgets (in the Window editor) or groups of menu entries (in the Menu editor) can be selected in a similar way.

## Cancel and OK

### Cancel

Clicking **Cancel** (or pressing Escape) will close the dialogue box without making any changes.

Clicking Adjust **Cancel** (or pressing Shift-Escape) will leave the dialogue box displayed but will remove any changes made since opening the box.

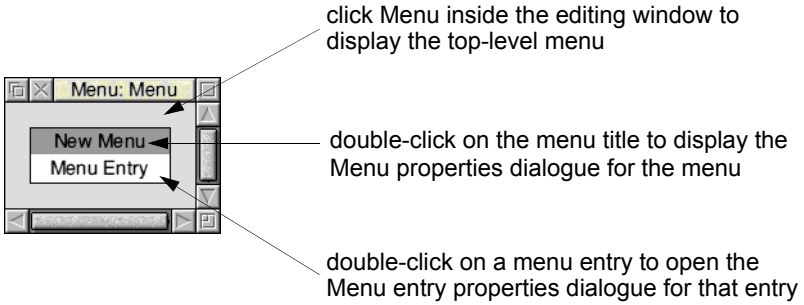
### OK

Clicking **OK** (or pressing Return) will close the dialogue box and include any changes in the object template.

Clicking Adjust **OK** (or pressing Shift-Return) will leave the dialogue box displayed and update all changes made since opening the box (e.g. if you increased the contents of a help message field, the **Length** field would then be increased automatically).

## Editing the Menu class

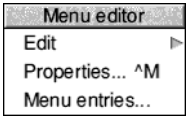
Double-clicking on a menu object template in the resource file display will display a Menu editing window with the following appearance:



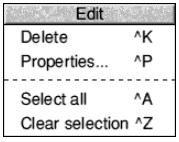

The editing window displays the menu as it will appear when displayed by the Toolbox.

### The Menu editor

Clicking Menu inside the editing window displays the following menu:




**Edit** leads to the Edit submenu.




**Delete** deletes the selected menu entries.

**Properties...** opens the Menu entry properties dialogue box for the selected menu entry (see *Editing a Menu entry* on page 430).

**Select all** selects all the menu entries in the menu.

**Clear selection** deselects all the menu entries in the menu.

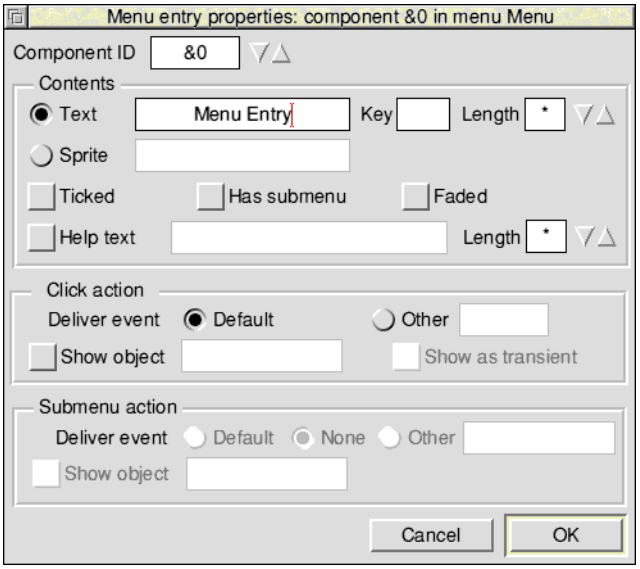
**Properties...** displays the Menu properties dialogue box, described in *Editing the Menu* on page 432.

**Menu entries...** displays the Menu entries window, described in *Inserting a new Menu entry* on page 433.

## Editing a Menu entry

### The Menu entry properties dialogue box

This is a dialogue box for viewing and editing the characteristics of a menu entry. You can open it by selecting a menu entry in the editing window and then selecting **Properties...** from the Edit menu (or by double-clicking on a menu entry):




**Component ID** is a text field containing the hexadecimal component identifier of this menu entry. Normally there is no need for you to edit this field as the component identifiers are automatically assigned. If you wish to assign identifiers yourself, you must ensure that they are unique within each menu.

Note: Clicking **OK** while any component ids are the same will elicit an error message and the dialogue box will stay open until this is sorted out.

**Text** and **Sprite** determine the contents of the menu entry:

If you select **Text**, you can then enter the text and keyboard short-cut to be displayed, and the maximum permissible length for the entry's text to be set to at run-time.

If you want to enter a keyboard short-cut into the **Key** field manually, you may have to use !Chars to display short-cuts such as Shift F3. It is more advisable to create a keyboard short-cut first (in the Keyboards shortcut dialogue box), and then drag this short-cut to the menu entry properties dialogue box.

This process is fully described in *Using a keyboard short-cut entry to 'fill in' a menu entry* on page 447.

If you select **Sprite**, you may then enter the name of a sprite to be displayed.

**Ticked** displays a tick next to this entry.

**Has Submenu** controls whether the entry has a submenu arrow.

**Faded** displays this entry in grey; when the menu is shown by an application the entry will be unselectable.

The writable field next to **Help text** allows you to supply a suitable interactive help string for the Toolbox to send to !Help when the mouse pointer is over this menu. If **Help text** is switched off, the Toolbox will instead supply any help text associated with the menu as a whole – see *Editing the Menu* on page 432).

The **Click action** section specifies what happens when the user selects this menu entry. The first thing that will happen is that the application will receive an event:

Selecting **Default** specifies that you will receive the default event (Menu_Selection).

Selecting **Other** allows you to receive whichever event you specify in the associated writable field (the event can be entered as a hex number, e.g. '&345', or as a decimal number).

After the event has been delivered, you can specify whether an object will be shown automatically. You can do this by turning on the **Show object** option and entering the name of the object to be shown in the associated writable field.

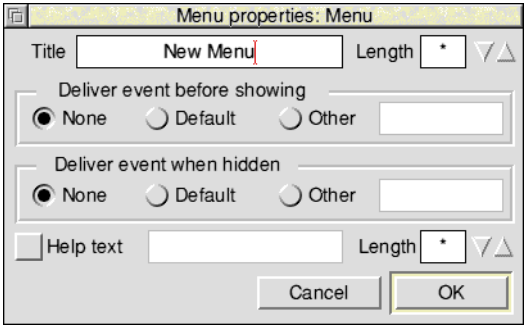
The **Submenu action** section is very similar, and specifies what should happen when the user traverses the submenu arrow of this entry. (The section is faded unless the **Has Submenu** option has been selected). The text fields have the same meanings as for menu selection. The default event in this case is Menu_Submenu.

The two **Show object** name fields may be filled in by dragging an object template's icon from the resource file display into the appropriate text entry field (or onto the corresponding option icon if the text entry field is shaded).

## Editing the Menu

### The Menu properties dialogue box

This is a dialogue box for editing the top-level characteristics of a menu. It is opened from the Edit menu or by double-clicking on the menu's title:




The **Title** field contains the text shown at the head of the menu.

Note: If a Menu with no title is shown, the Wimp will not display a title bar. This is not Style Guide compliant, but the Menu editor allows this so that you can set a title at run-time.

**Deliver event before showing** controls the following:

- **None** specifies that no event should be returned.
- **Default** specifies that the default event (Menu_AboutTobeShown) should be returned immediately before showing the window.
- **Other** allows you to specify a different event to be delivered to the application. The associated field displays the event code in hex; you may enter event codes in either decimal or hex (by prefixing with '&').

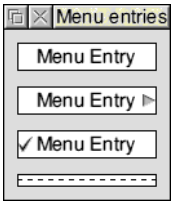
**Deliver event when hidden** controls the following:

- **None** specifies that no event should be returned.
- **Default** specifies that the default event (Menu_HasBeenHidden) should be returned immediately after the window is hidden.
- **Other** allows you to specify a different event to be delivered to the application. The associated field displays the event code in hex; you may enter event codes in either decimal or hex (by prefixing with '&').

The writable field next to **Help text** allows you to supply a suitable interactive help string for the Toolbox to send to !Help when the mouse pointer is over this menu (if **Help text** is switched off, the Toolbox will not reply to such HelpRequest messages).

## Inserting a new Menu entry

You can insert new menu entries into the menu using the Menu entries window. The Menu entries window is opened by selecting **Menu entries...** from the top-level menu.




The Menu entries window contains a dotted line separator and three prototype menu entries:

- a basic menu entry
- a menu entry with a submenu arrow
- a ticked menu entry.

The menu entries in the Menu entries window may be dragged with the mouse and dropped over the menu area to insert new menu entries and separators. The new entry is placed between two existing entries according to the vertical position of the drop point. If the mouse pointer is within the menu's title, it is inserted after the title; if it is dropped after the final entry it is appended at the bottom.

## Manipulating menu entries

### Copying menu entries

You can copy a menu entry from one part of a menu to another using Shift-Drag Select. The insertion point is determined as for inserting a new item. New menu entries are automatically assigned unique component ids within the menu.

You can also use Drag Select to copy menu entries between editing windows.

### Moving menu entries between different editing windows

You can move menu entries between different Menu editing windows using Shift-Drag Select. The selected entries are deleted from the source window.

### Re-ordering menu entries

You can re-order menu entries using Drag Select. The insertion point is determined as for inserting a new item.

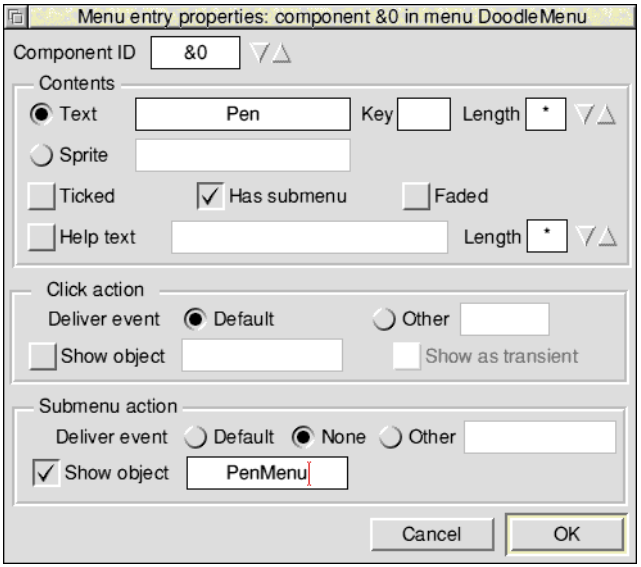
Note: If a copy or move operation results in a menu containing two entries with the same component id, the editor forces the newly inserted one to have a unique id.

## Example menu

This example shows you how you might create the three menu entries in the following typical menu:


### Creating a submenu

The first menu entry in the above example (**Pen**) has an associated submenu, so the Menu entry properties box could be filled in as follows:



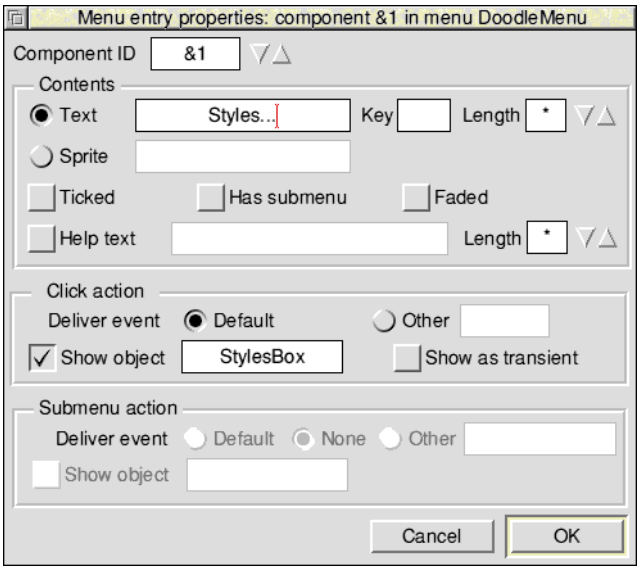

The minimum sections to edit in the Menu entry properties box are

- **Text** – give the menu entry a unique name (e.g. ‘Pen’).
- **Has submenu** – switch it on.
- **Show object** (in the Submenu action area) – switch it on and specify the name of the object to show if the user traverses the submenu arrow (e.g. ‘PenMenu’).

You would then create another menu object template and give it the name ‘PenMenu’. This object would be displayed when the user traverses the submenu arrow.

## Displaying a dialogue box

The second menu entry in the above example (**Styles...**) has an associated dialogue box, so the Menu entry properties box could be filled in as follows:




The minimum sections to edit in the Menu entry properties box are as follows:

- **Text** – give the menu entry a unique name (e.g. ‘Styles’). In this particular example the ellipsis (...) signifies to the user that the dialogue box that will be displayed is a persistent dialogue box (so the **Show as transient** option should not be selected).
- **Show object** (in the Click action area) – switch it on and specify the name of the object to show if the user clicks on this entry (e.g. ‘StylesBox’).

You would then create a window object template for the dialogue box and give it the name ‘StylesBox’. This object would be displayed when the user clicks on **Styles...**

**Note**: Any object (e.g. submenus and dialogue boxes) can also be built dynamically at run-time by the client application (see *Attaching a submenu dynamically* on page 171).

## Creating a keyboard short-cut

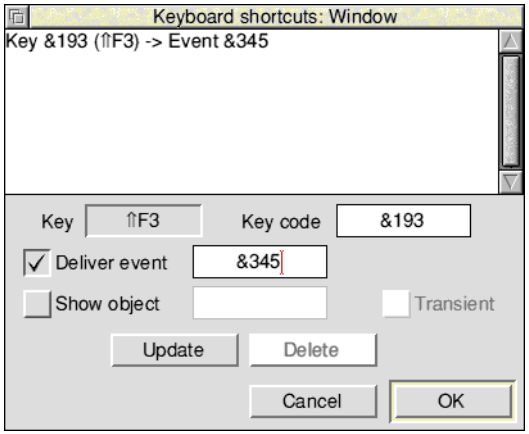
The third menu entry in the above example (**Group ⇑F3**) returns an event if the user clicks on the entry or uses a keyboard short-cut (Shift F3); this would allow the client application to perform an appropriate action on receipt of the event.

Creating this keyboard short-cut requires two stages:

- defining the keyboard short-cut within the window object template itself.
- dragging this keyboard short-cut to the Menu entry properties box.

### Defining the keyboard short-cut

The first stage is to define the keyboard short-cut within the window object template itself. For example:



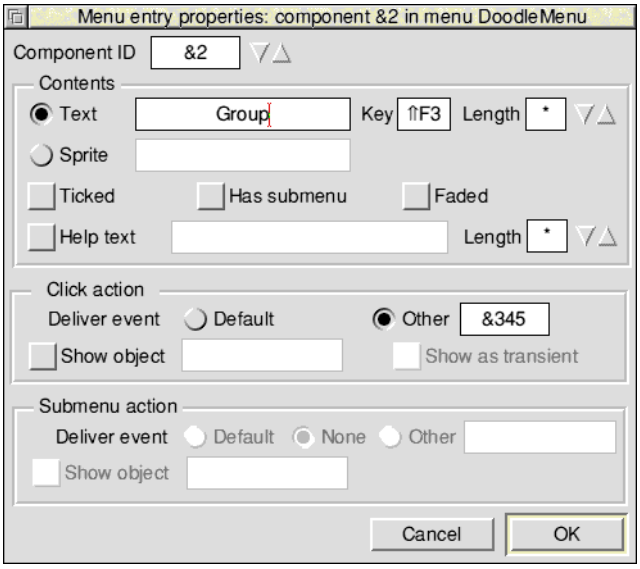

1. Click Select on the **Key** field and press Shift F3; the corresponding code (&193) is automatically entered into the **Key code** field.
2. Specify the event code in the **Deliver event** box (e.g. '&345').
3. Click on **Update** to add the new keyboard short-cut to the scrolling list.
4. Click on **OK** to add the new keyboard short-cut to the Window object template.

For more information on keyboard short-cuts see *Keyboard short-cuts* on page 445.

### Filling in the Menu entry properties box

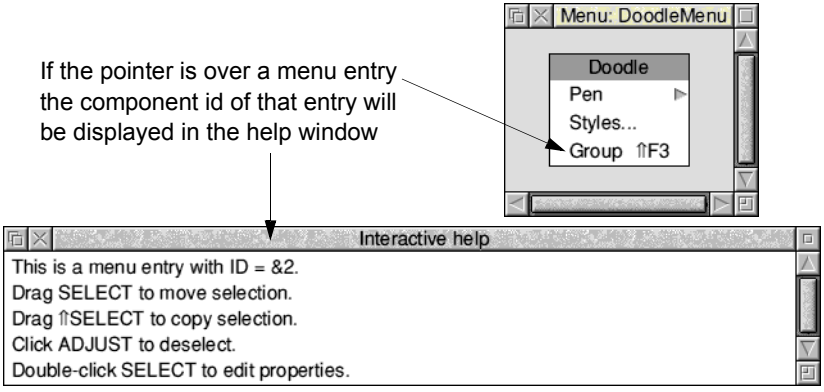
The next stage is to open the third menu entry and give it a unique name (i.e. ‘Group’), and then drag the keyboard short-cut to it. This will automatically fill in:

- the Key short-cut (e.g. Shift F3) in the **Key** field
- the event code to return if the user clicks on this entry (e.g. ‘&345’):



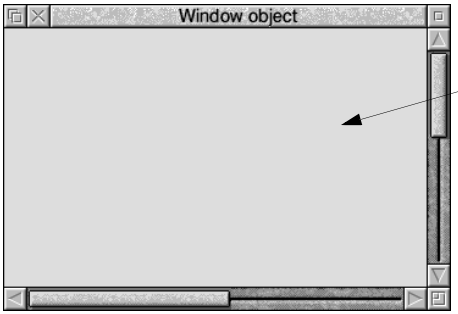

## Interactive help for menu entries

The Help window gives you information about the Menu window and also displays the component id of an individual menu entry:

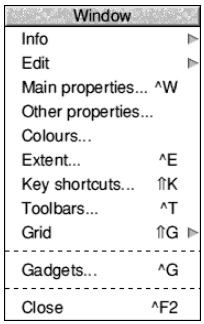


## Editing a Window object template and gadgets

Double-clicking on a window object template in the resource file display will display an editing window. This window displays the window object template as it will appear (complete with gadgets) when displayed by the Toolbox. It has the following appearance:




### The Window menu




**Info** leads to an Info box showing the object template's name.

**Edit** leads to the Edit submenu for the selected gadget(s).
See *The Edit submenu* on page 452.

**Main properties...** opens the Main window properties dialogue box. This box allows you to specify those properties.
See *The Main properties dialogue box* on page 439 for more details.

**Other properties...** opens the Other window properties dialogue box. This box allows you to edit those properties of a window object template that you would normally only specify once.
See *The Other properties dialogue box* on page 441 for more details.

**Colours...** opens the Window Colours dialogue box.
See *Window Colours* on page 444 for more details.

**Extent...** opens the Window Extent dialogue box.
See *Window Extent* on page 444 for more details.

**Key shortcuts...** opens the Keyboard short-cuts dialogue box. This allows you to define keyboard short-cuts for use inside the window.
See *Keyboard short-cuts* on page 445 for more details.

**Toolbars...** allows you to attach toolbar object templates to this window. See *Toolbar object template* on page 456 for more details.

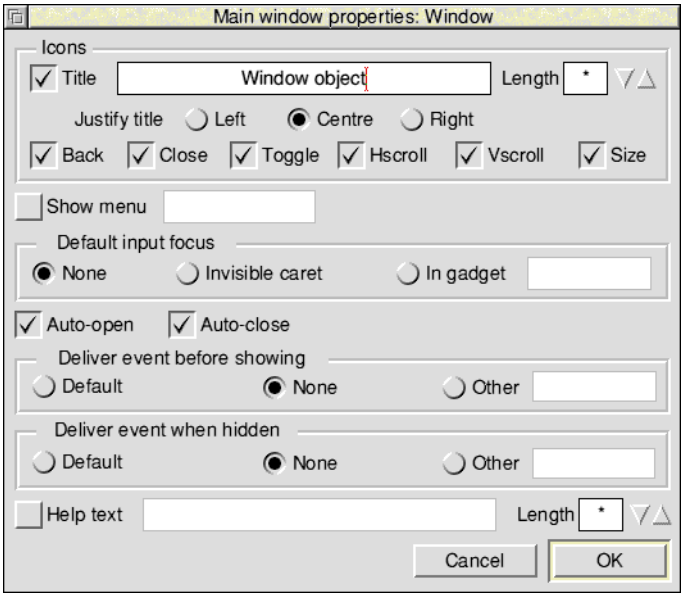
**Grid** leads to the Grid dialogue box. This allows you to display an optional grid of alignment points to assist in the uniform placement of gadgets.
See *The Grid* on page 447 for more details.

**Gadgets...** opens, or brings to the front, the gadgets window. This is a selection of gadgets which may be dragged into a Window object template to populate it with gadgets. See *The gadgets window* on page 449 for more details.

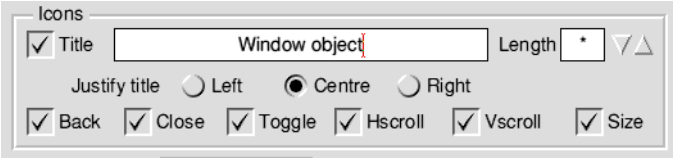
**Close** closes the window and incorporates any changes.

## The Main properties dialogue box


This dialogue box allows you to edit the main properties of a window object template. The name of the window object template that the dialogue box refers to is displayed in the titlebar. Choose **Main properties...** from the Window menu or double-click Select on the window background to display this box:



**Icons** controls the following features:




**Title** allows you to enter the title of the window within the title bar. If you switch this option off the window will not have a title bar.

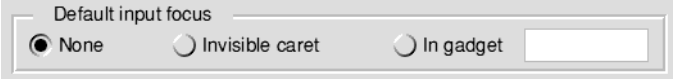
Note: The window title is always a vertically-centred, indirected text icon in system font; there is no facility to set a validation string.

**Justify title** allows you to specify the justification of the title within the title bar.

The **Back**, **Close**, **Toggle**, **Hscroll**, **Vscroll** and **Size** option buttons control whether the Back icon, Close icon, Toggle Size icon, Horizontal scroll bar, Vertical scroll bar and Adjust size icons are displayed.

**Show Menu** is an option button that controls whether the window has a menu attached to it. If this is switched on, the associated writable field is unshaded for the menu object template's name to be entered. Alternatively the field can be filled in by dropping a menu object template onto it (or onto the corresponding option icon if the field itself is shaded).

**Default input focus** allows you to set the characteristics of the default input focus for the window.




**None** specifies that the window has no input focus and no caret.

**Invisible caret** specifies that the window has input focus, but no caret is displayed until the user clicks in an appropriate area.

**In gadget** specifies that the window has input focus and the caret is displayed inside a gadget. You can enter the component id of the gadget in the adjoining field or drag a gadget to the field (or to the corresponding radio button if the field itself is shaded).

**Auto-open** controls whether the Window module automatically (re-)opens the window when a Wimp_OpenWindowRequest event is received.

**Auto-close** controls whether the Window module automatically closes the window when a Wimp_CloseWindowRequest event is received.

**Deliver event before showing** controls the following:

- **Default** specifies that the default event (Window_AboutTobeShown) should be returned immediately before showing the window.
- **None** specifies that no event should be returned.
- **Other** allows you to specify a different event to be delivered to the application. The associated field displays the event code in hex; you may enter event codes in either decimal or hex (by prefixing with '&').

**Deliver event when hidden** controls the following:

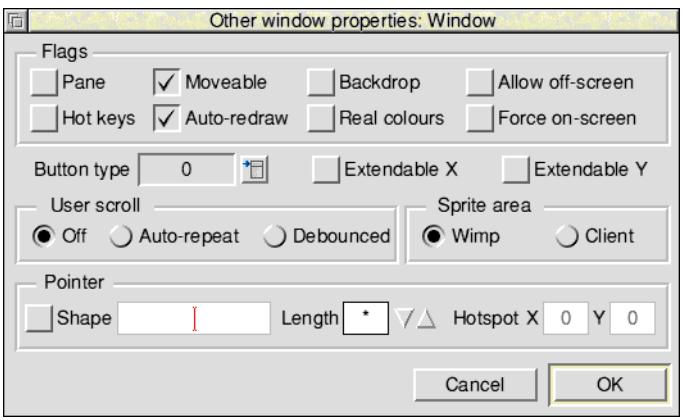
- **Default** specifies that the default event (Window_HasBeenHidden) should be returned immediately after the window is hidden.
- **None** specifies that no event should be returned.
- **Other** allows you to specify a different event to be delivered to the application. The associated field displays the event code in hex; you may enter event codes in either decimal or hex (by prefixing with '&').

The writable field next to **Help text** allows you to supply a suitable interactive help string for the Toolbox to send to !Help when the mouse pointer is over this window (if **Help text** is switched off, the Toolbox will not reply to such HelpRequest messages).

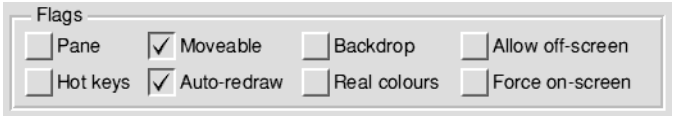
The above controls are described in the *Window Manager* chapter in Volume 3 of the *RISC OS 3 Programmer's Reference Manual*, and in the chapter *Window class* on page 287.

### The Other properties dialogue box


This dialogue box allows you to edit those properties of a window object template that you would normally only specify once. You can only display this box by choosing **Other properties...** from the Window menu:



**Flags** controls the following features:




**Pane** specifies that the window is a pane.

**Moveable** determines if the window is moveable, i.e. it can be dragged by the user.

**Backdrop**, if selected, does not allow any other windows to be opened below this one.

**Allow offscreen** allows the window to be opened or dragged outside the screen area (regardless of the Configure option settings).

**Hot keys** allows events to be generated for hot keys.

**Auto-redraw** specifies that the window can be redrawn entirely by the Wimp, i.e. there are no user graphics in the work area.

**Real colours** specifies that the window colours should be treated as `GCOL` numbers instead of standard Wimp colours.

**Force on-screen** forces the window to stay on screen.

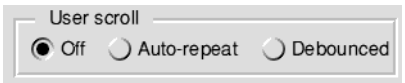
Note: Old-style window flags are not supported (i.e. bit 31 of the window flags word is always set).

**Button type** determines how the Wimp will deal with mouse movements and clicks over the window's background. There are 16 possible types which can be selected from the Pop-up menu (see the *RISC OS 3 Programmer's Reference Manual* entry for Wimp_CreateIcon on page 1-93 for more details).

**Extendable X** ignores the right-hand extent if the Adjust size icon of the window is dragged.

**Extendable Y** ignores the lower extent if the Adjust size icon of the window is dragged.

**User scroll** controls the Scroll_Request event:




**Off** does not return a Scroll_Request event.

**Autorepeat** returns a Scroll_Request event 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).

**Debounced** returns a Scroll_Request event when a mouse button is clicked on one of the arrow icons (but with no auto-repeat) or in the outer scroll bar region (no auto-repeat).

**Sprite area** controls whether sprites are located in the client area or the Wimp sprite area.

**Shape** is an option button that controls whether the mouse pointer should change shape when it is over the window. If this is switched on, the associated writable fields are unshaded for the pointer sprite's name, its length, and the coordinates of its hotspot to be entered.

## Manipulating the window

You can use the icons around the window object template to manipulate the window's size, position and scroll offsets. This information is saved with the template. The stacking position is not saved; all templates are saved with a stacking position of -1 (top of stack) unless the window's Backdrop flag is set, in which case the position is -2 (bottom of the stack).

### Re-sizing the window

You can resize windows which have no scrollbar using Ctrl-Shift-Drag Adjust. The window can only be resized subject to the constraints of its current work area extent.

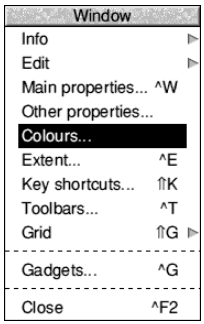
### Moving the window

You can move windows which have no title bar using Ctrl-Shift-Drag Select.

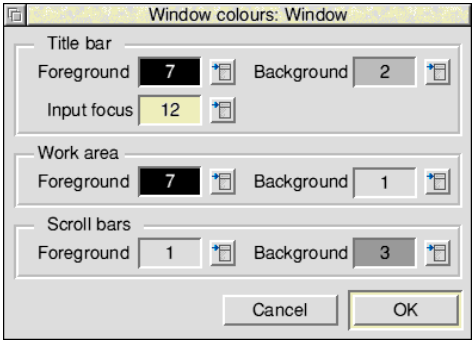
### Closing the window

The window's Close icon, if present, may be used to close the window. The window may also be closed by using the **Close** menu option, or by the keyboard short-cut Ctrl-F2.

## Window Colours



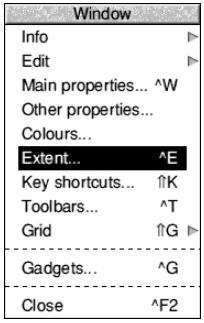

This dialogue box allows you to edit the colours of a window:



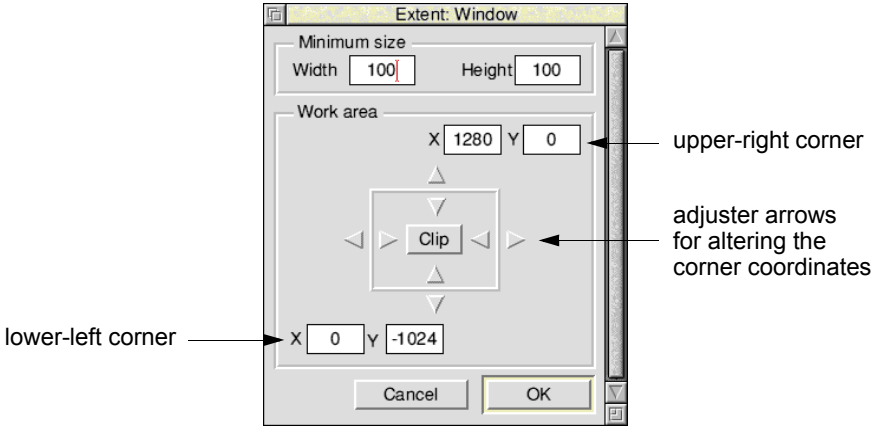

The display fields contain the Wimp colour number of the chosen colour, and have their backgrounds set to that colour. The menu buttons invoke a pop-up menu offering a choice of the 16 Wimp colours. The menus for **Titlebar: Foreground** and **Work area: Background** also offer the choice **Transparent**.

An alternative form of this dialogue box is displayed if the window object's **Real colours** flag has been set (see *The Other properties dialogue box* on page 441). In this case the pop-up menus are not available and the colour display fields are replaced by writable icons; values in the range 0 to 255 may be entered.

## Window Extent




This dialogue box allows you to edit the extent (work area size) of a window:



The **Work area** is represented by two pairs of x,y coordinates for the lower-left and upper-right corners. You may adjust these coordinates by typing into the adjoining writable fields, or using the adjuster arrows on the ‘adjustable square’.

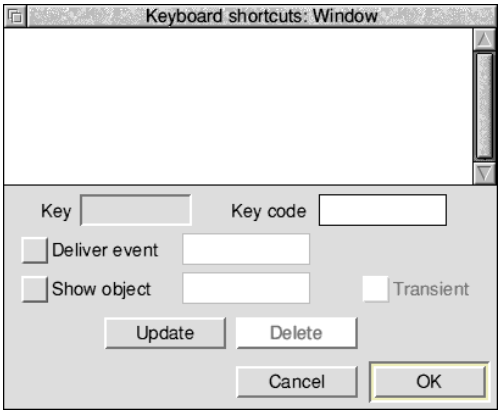
Clicking on the **Clip** button causes the size of the work area to be made equal to the window's current visible area on your screen.

**Width** and **Height** allow you to enter the size below which the window may not go.

## Keyboard short-cuts


Each window may have a list of keyboard short-cuts associated with it. These are programmable mappings from Wimp key codes to Toolbox events. When a keystroke event is delivered, the Window module checks to see if it is in the list of short-cuts for the window containing the caret. If so, it delivers the associated event to the application. Alternatively (or additionally), a keyboard short-cut may be associated with an object template which specifies an object to be shown when the keystroke happens.

The keyboard short-cuts assigned to a window may be created and modified using the Keyboard shortcuts dialogue box. The name of the window that the dialogue box refers to is displayed in the titlebar:




Existing keyboard short-cuts are displayed in the scrolling area. Double-click on one of them to load its details into the icons below for editing; alternatively simply type in the details of the new one.

**Key** is a special icon which allows you to define a key code by pressing the corresponding key(s) on the keyboard. First click Select on the icon to activate it and then press the key combination. The corresponding code appears in the **Key code** field, and a description of the key appears in the **Key** field. Note that Shift-Ctrl-letter combinations are not allowed.

**Key code** is the Wimp keycode for the event as described in the *RISC OS 3 Programmer's Reference Manual* entry for Wimp_Poll (see page 1-112). This code is displayed automatically when you enter a key press into the **Key** field, or you may specify it yourself as a decimal number or a hex number (by preceding it with &).

**Deliver event** selects whether the keystroke will generate an event. The associated writable field allows you to enter the event code as a decimal or hex number.

**Show object** selects whether the keystroke should show an object. The associated writable field allows you to specify the name of the object template to be shown.

**Transient** causes the object to be shown with transient behaviour.

**Update** adds the new keyboard short-cut to the scrolling list, replacing any short-cut for the same key already present.

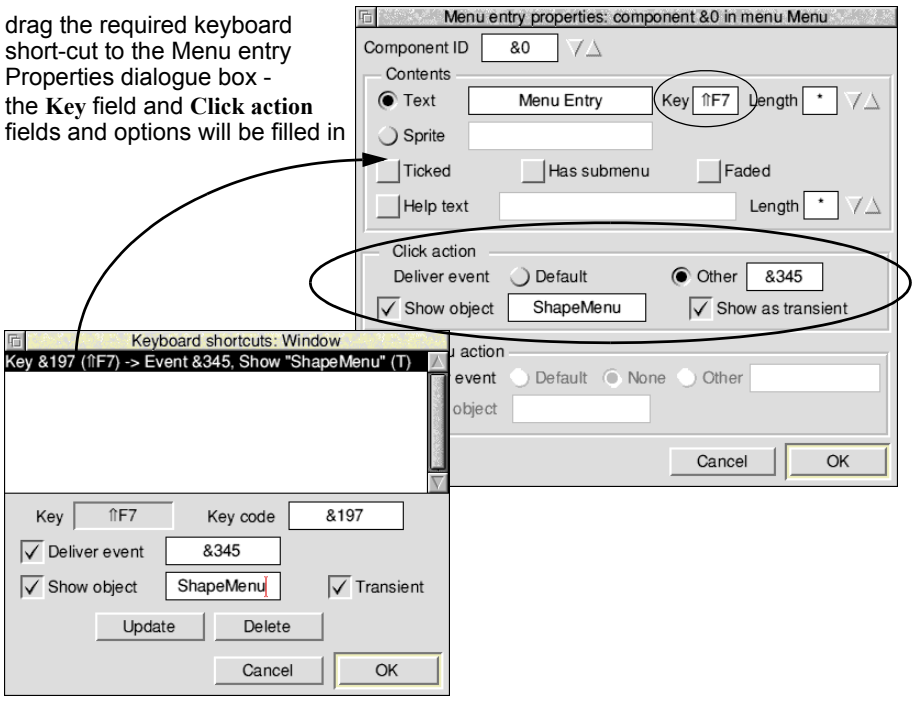
**Delete** deletes the selected short-cuts from the list. The short-cuts listed in the scrolling list can be selected for deleting by clicking on them (Adjust toggles whether the short-cut is selected or not).

**OK** accepts the updated list of short-cuts and closes the window.

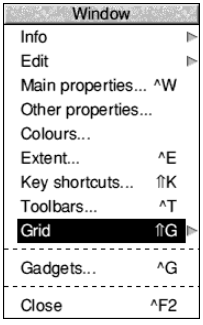
**Cancel** closes the window, discarding any changes.

### Using a keyboard short-cut entry to 'fill in' a menu entry

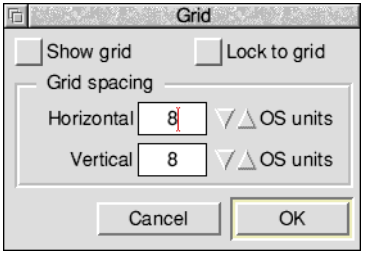
You can fill in the **Key** field and **Click action** fields (**Deliver event**, **Show object** and **Show as transient**) in a menu entry by dragging a keyboard short-cut entry from the **Keyboard shortcuts** scrolling area and dropping it into a Menu entry properties dialogue box in the Menu editor:




## The Grid




The Grid dialogue box can display an optional grid of alignment points to assist in the uniform placement of gadgets:



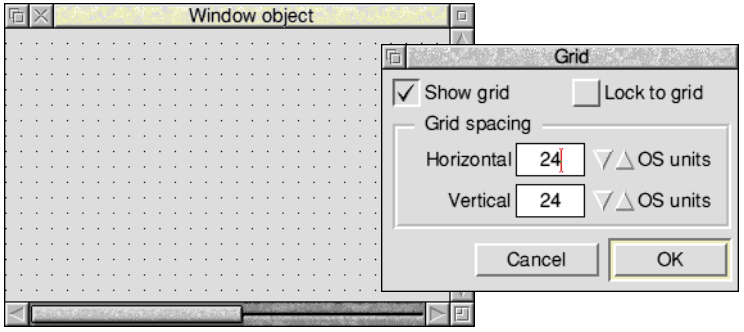
The grid is represented by a matrix of dots which overlay the contents of the window. The grid spacing is specified as a number of OS Units between grid points, this being configurable independently for different windows.

**Show grid** controls whether the grid is currently displayed for this window.

If **Lock to grid** is selected, gadgets may only be moved or resized in units of grid spacing. This means that if you have a group of gadgets then you can move (or resize) them, either horizontally or vertically, in multiples of the selected grid spacing, and they will keep their relative positions.

Note: If you drag gadgets into a window, the gadgets will not be locked to the grid in the window until you use the **Snap to grid** option (see page 453).

**Grid spacing** controls the spacing of the grid. For maximum compatibility across different RISC OS modes you are advised to set grid spacings to exact multiples of 8, and to this end the adjuster arrows alter the grid spacing in steps of 8. Values that are not a multiple of 8 may be entered from the keyboard but will be forced to be exact multiples of 4. For example:

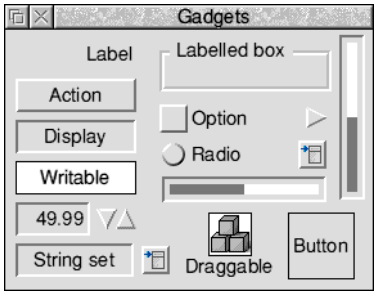



There is also an option that allows you to snap gadgets to grid points. This is described in *Snap to grid* on page 453.

# Gadgets

## The gadgets window

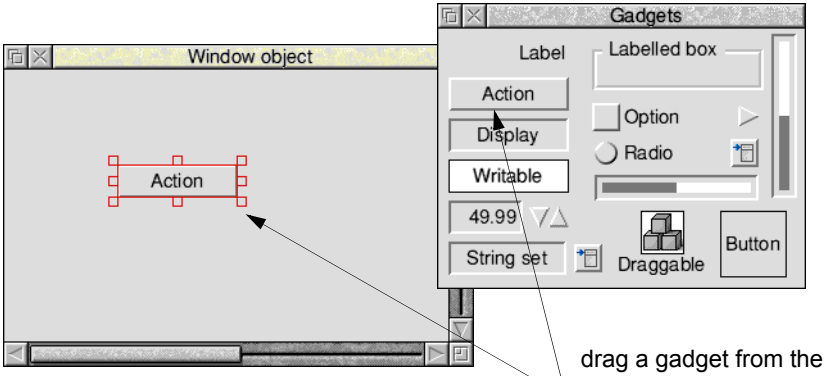

You can populate a window with gadgets by dragging them in from the gadgets window. This is a read-only window containing a typical example of each supported gadget type. You can display the gadgets window by choosing the **Gadgets...** option from a Window menu (or by pressing Ctrl-G):




The gadgets in the gadgets window may not be moved or deleted. The gadgets window does not have a menu, and only the keyboard short-cuts ^A and ^Z are available.

## Positioning and moving gadgets

You can drag any of the gadgets from the gadgets window into your window object template and drop them wherever is appropriate.



### Repositioning and copying

You can reposition one or more gadgets in your window by first selecting them and then using Drag-Select with the pointer over one of the selected gadgets. If **Lock to grid** is on, the gadgets are moved by the nearest multiple of the grid spacing. If you hold down Shift, a copy of the gadgets is made.

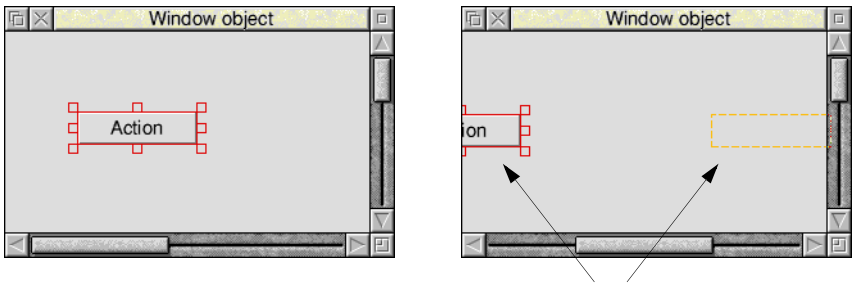
### Accurate positioning

There are three ways to position a gadget accurately:

- specify its coordinates in the window's work-area coordinate system (see *The Coordinates dialogue* on page 454)
- align it with one or more other gadgets using the Align menu (see page 455).
- move the gadget (or selection of gadgets) using the cursor keys. This can be done by selecting a gadget, holding down the Select button (as if dragging), and then pressing any of the four cursor keys.

### Auto-scrolling

If you want to move a gadget beyond the visible area of the window on the screen you must drag the gadget **very slowly** towards one of the sides of the window.
Auto-scrolling of the window will occur when the mouse pointer comes close to a side of the window; scrolling is faster the closer the pointer is to the edge.



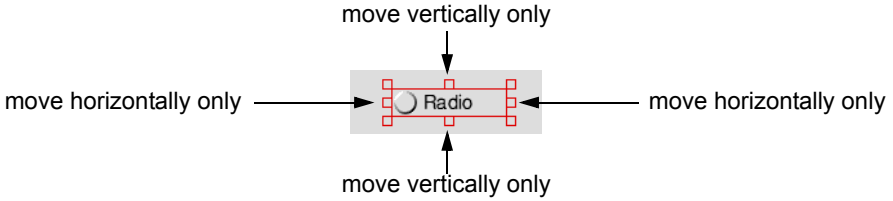

### Moving gadgets between windows

You can copy gadgets between windows by dragging them from one window object template to another (to avoid auto-scrolling you should not drag a gadget too slowly when dragging between windows).

If you hold down Shift the gadgets are deleted from the source window.

**Moving a gadget in one direction only**

You can move a gadget in one direction only using Drag-Adjust on the top, bottom, left or right resize handles (if **Lock to grid** is switched on, the gadgets are moved by the nearest multiple of the grid spacing):




## Changing the size of a gadget

You can change the size of a gadget using Drag-Select on a resize handle (if **Lock to Grid** is on the change in size of the gadget (or selection of gadgets) is always a multiple of the grid spacing).

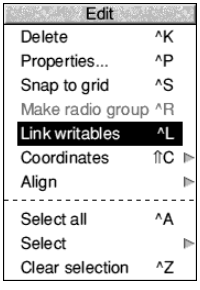
You can also change the size of one gadget, or of a selection of gadgets, using the **Width** and **Height** options in the Coordinates dialogue box (see page 454).

## Stacking

Gadgets are not intended to be stacked; so there are no facilities for placing one gadget ‘above’ another. Gadgets whose bounding boxes overlap will stack in an arbitrary order; there is no way you can guarantee that this order will remain unchanged. The exception to this rule is the labelled box gadget, which is always placed beneath all other gadgets.

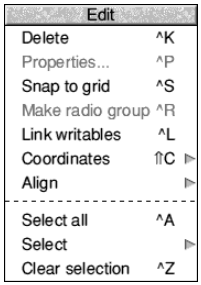
## Moving the caret between writable gadgets

You can define the order in which the caret is moved between writable gadgets (in response to the Tab, Shift-Tab, up-arrow and down-arrow keys) by filling in the **Before** and **After** fields of the gadget properties dialogues:


These fields contain the component ids of the two gadgets ‘before’ and ‘after’ the gadget. To help you fill these in, you can drag gadgets into them, or more typically you can use the **Link writables** option in the Edit submenu. This automatically fills in these fields for all the selected gadgets that support caret movement (writable fields, string sets and number ranges). The ordering imposed is left-to-right and top-to-bottom (as if you were reading a page of text).

## The Edit submenu




If you select one or more gadgets then, depending on the gadgets selected, some of the following edit options in the Edit submenu will be available:

**Delete** deletes the selection of gadgets.

**Properties...** opens the gadget properties dialogue box for the selected gadget. An alternative way to open this dialogue box is to double-click Select on the gadget itself.

**Snap to grid** snaps selected gadgets to the window grid (see *Snap to grid* on page 453). Note that this option is independent of the **Lock to grid** setting, and is operative even when the grid points are not displayed.

**Make radio group** makes any selected radio buttons into a radio group (see *Manipulating radio groups* on page 453).

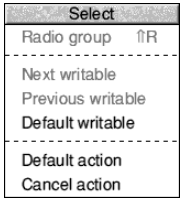
**Link writables** links the selected writable gadgets together so that they can be traversed with Tab, Shift-Tab, up arrow and down arrow keys (see *Moving the caret between writable gadgets* on page 451).

**Coordinates** allows gadget coordinates to be entered from the keyboard for precise positioning (see *The Coordinates dialogue* on page 454).

**Align** allows you to align gadgets with one another (see *The Align menu* on page 455).

**Select all** selects all the gadgets in the window.

**Select** leads to the Select submenu.




**Radio group** selects all the radio buttons in the radio group to which the selected radio button belongs (see *Manipulating radio groups* on page 453).

**Next writable** selects the gadget that is linked after the selected gadget.

**Previous writable** selects the gadget that is linked before the selected gadget.

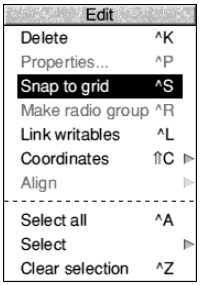
**Default writable** selects any gadget that is assigned as the ‘default input focus’ for the window.

**Default action** selects any action button that is assigned as the default action button.

**Cancel action** selects any action button that is assigned as the cancel action button.

**Clear selection** deselects all the gadgets in the window.

## Snap to grid




The **Snap to grid** operation on the **Edit** submenu makes each selected gadget move so that its alignment point is on the nearest gridpoint.

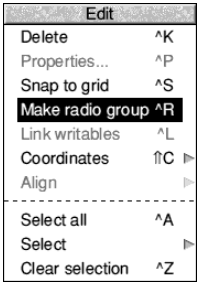
The 'alignment point' of a gadget is as follows:

- the Y-coordinate is always the centre of the gadget
- the X-coordinate is normally the lefthand side of the gadget.
  (the only exception is the label gadget; where the alignment point is on the lefthand side if the label is left-justified, on the righthand side if the label is right-justified, and in the centre if the label is centre-justified)

**Snap to grid** snaps each selected gadget independently (when the selection is moved under grid-lock, the relative positions of the gadgets are preserved).

If you drag a selection of gadgets into a window they will not be snapped to the grid in that window (even if **Lock to grid** were switched on). If they were snapped automatically to the grid it would alter their relative positions to each other, and this might not be desired. The gadgets remain selected when dragged into a window, so if you do want to snap them to the grid then you can just press Ctrl-S (for **Snap to grid**).

## Manipulating radio groups




When you drag radio buttons into a Window object template from the gadgets window, each one ends up in its own new radio group. You must then select and group them explicitly using the **Make radio group** option in the Edit menu.

The **Make radio group** option is faded unless the window's selection consists entirely of radio buttons. When you choose this menu entry, the selected radio buttons are placed into a single new radio group.

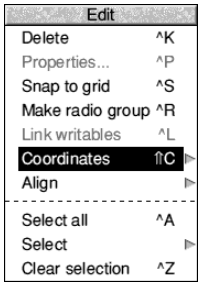
To select all members of a radio group, press Menu over one of them and choose **Radio group** from the Select submenu in the Edit menu. This enables you to see instantly the grouping relationship between radio buttons.

When a radio button is copied within a window by use of Shift-Drag, the copy is put into the same group as the original. So the easiest method to create a radio group is to drag a single radio button into the Window object template and make multiple copies of it using Shift-Drag Select.

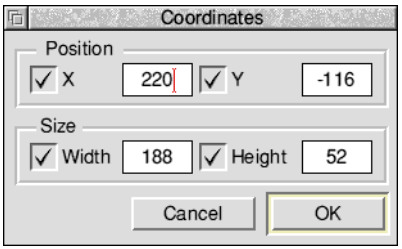
**Dragging a group of radio buttons between window templates**

Adding radio buttons to a window never adds them to a pre-existing group; but any radio groups added to a window remain as groups.

## The Coordinates dialogue




This dialogue box allows you to position or size selected gadgets by entering coordinates (in the window's work-area coordinate system) from the keyboard:

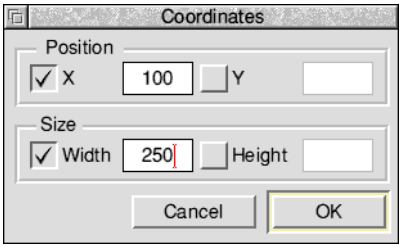



When a single gadget is selected, all four option buttons are switched on and the four writable fields are filled in with its position and size.

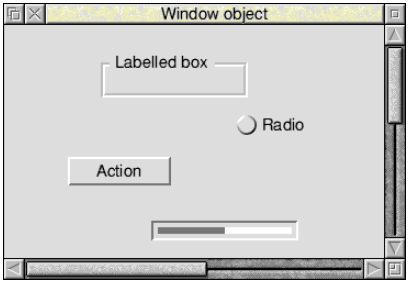
If you select more than one gadget, they are checked to see if they have common values for any of the four attributes. Those attributes with common values are filled in, and the corresponding option buttons switched on. Those attributes with differing values are faded, and the corresponding option buttons switched off. You may toggle the option buttons to alter the settings of any of the latter attributes.

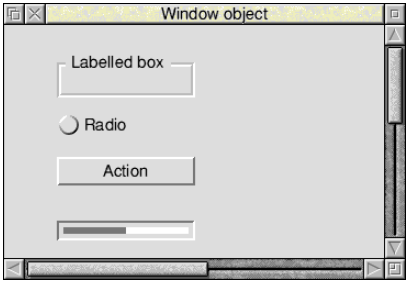
When you click **OK**, the attributes are set from those fields with the option buttons switched on. The attributes that have their option buttons off are left alone. Thus, it is possible to set several gadgets to have the same X position without altering their Y positions, and at the same time equalise the width of the selected gadgets:

selecting the four gadgets below, and setting Position and Size as opposite ...

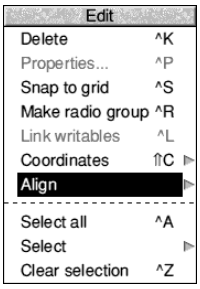



... would result in this



## The Align menu

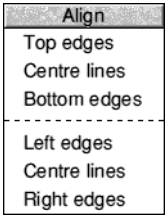



The Align menu allows you to align a group of selected gadgets in a window

**1** select one or more gadgets

**2** decide which gadget you want to align the other gadgets to and press Menu over it (this gadget does not need to be part of the selection)

**3** go into the **Align** menu and click on the required type of alignment:

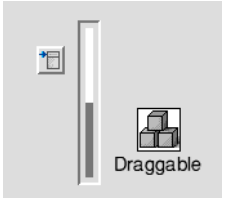



The gadgets are then moved to align with the nominated gadget.

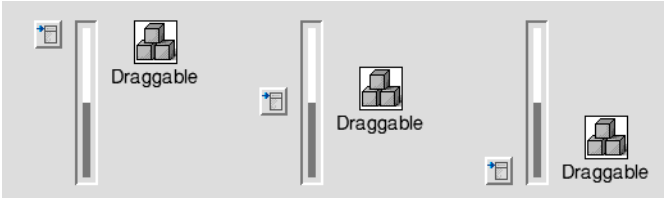
If you press Menu when the pointer is not over a gadget the Align menu will be faded. **Lock to grid** is ignored when aligning.

### Aligning gadgets from top to bottom

The top three options control how the gadgets will be aligned from top to bottom. In the following example the gadgets are aligned with the slider gadget:



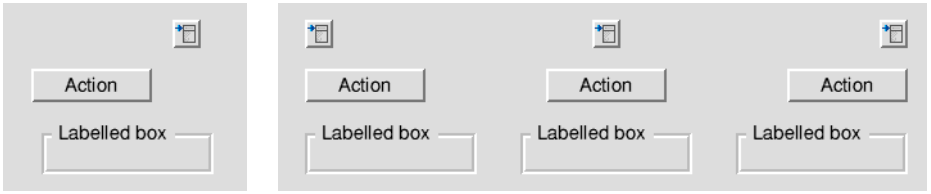

Before aligning




Top edges | Centre lines | Bottom edges

**Aligning gadgets from left to right**

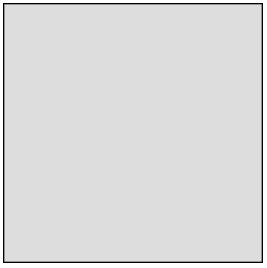
The bottom three options control how the gadgets will be aligned from left to right. In the following example the gadgets are aligned with the Labelled box gadget:




Before aligning | Left edges | Centre lines | Right edges

## Toolbar object template

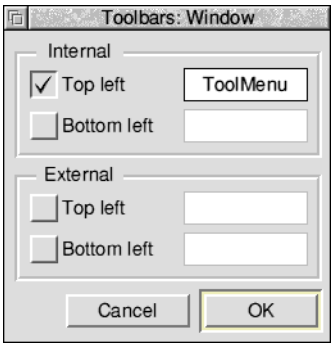
The toolbar object prototype is a window object template. Double-clicking on it inside a resource file display will display a blank editing window:



You can then edit this window, move it round the screen (using Ctrl-Shift-Drag Select), change its size (using Ctrl-Shift-Drag Adjust) and colour, drag gadgets into it etc, in exactly the same way as you would edit a window object template.

### Positioning the toolbar within a window


Once you have finished designing your toolbar you can open a window object template, go into the window menu for that template, and select the **Toolbars...** option. This will display the following box:

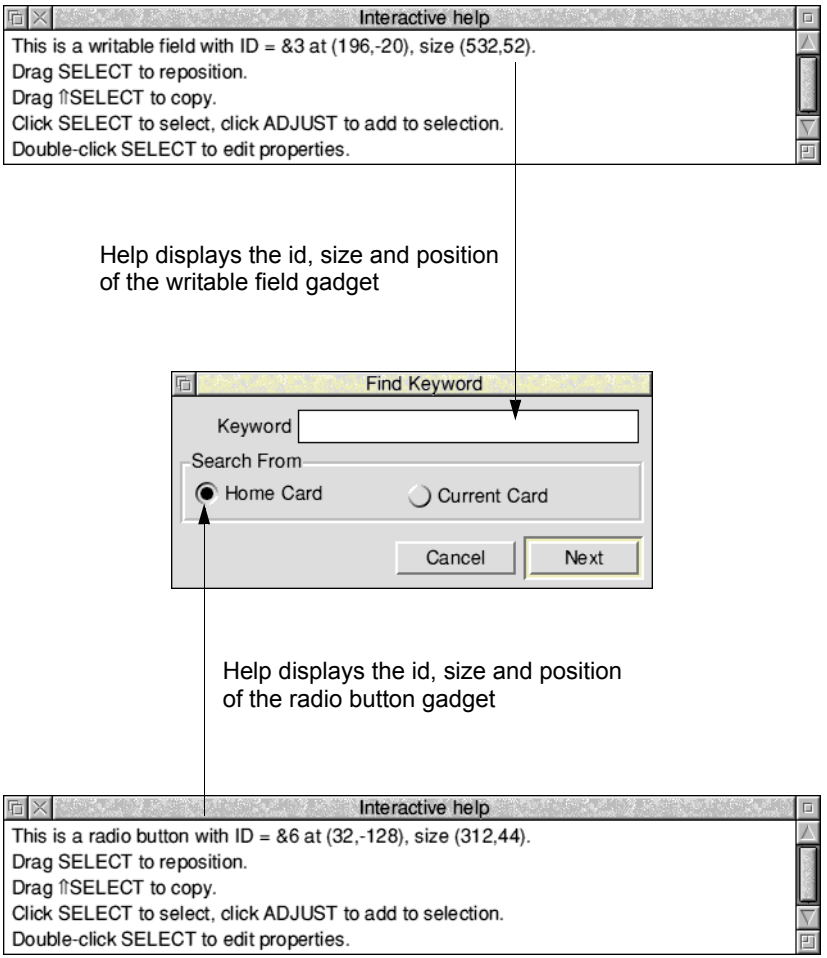



You can enter a toolbar object template name into a writable field after switching on the corresponding option icon (e.g. to the right of **Top left**), or drop a toolbar object template onto the writable field (or onto the associated option icon if the writable field is faded).

## Interactive help for gadgets

The Help window displays the id, size and position of a gadget in a window.

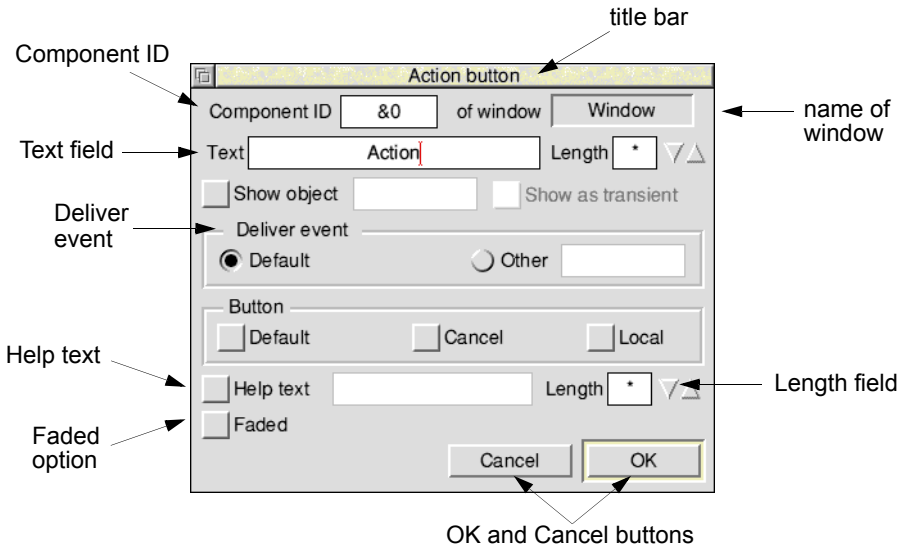
In the following example, a window has been customised as a Find dialogue box and the pointer has been moved over two of the gadgets in the window:




The customised window shown above is described in *Adding a find capability* on page 57 in the *User Interface Toolbox* manual.

## Common features in gadget properties boxes

Some features are common to several or all gadget properties boxes. These are described here rather than repeating their descriptions in each gadget section:




- The title bar contains a string describing the type of gadget being edited.
- The first field is always a writable icon containing the gadget's **Component ID**. Normally you do not have to enter anything into this field as a unique number is automatically assigned to it. If you need to, you can change a gadget's id by typing a new id into this icon. When OK is pressed, the gadget will be renumbered. Duplicate component ids are not allowed within a window; any attempt to set a component id to one already used by a gadget in the same window will be faulted. New gadgets dragged in from the gadgets window have a new unique component id chosen automatically.
- Next to the component id is a display field showing the name of the window object template that the gadget belongs to.
- Many of the dialogues have a **Text** field allowing you to type in a string which appears in the gadget.
- All gadgets have a **Help text** field. This is a writable icon for you to supply a suitable interactive help string for the Toolbox to send to !Help when the mouse pointer is over the gadget. If the **Help text** option icon is not selected, the underlying window will respond to !Help instead.
- All gadgets have a **Faded** option button. Setting this fades the gadget and makes it inactive to mouse clicks.

- Some string entry fields (including Help and Text) have an associated **Length** field. This is a writable number range which specifies the length of the buffer used to hold the text. For more details on how this field works see *Help messages* on page 426.
- Several of the dialogues feature a **Deliver event** section. This section allows you to specify whether or not you want an event to be returned, and what that event should be:
  - **Default** specifies that the default event should be returned.
  - **None** (if present) specifies that no event should be returned.
  - **Other** is used to specify a user event; you may enter event codes in either decimal or hex (by prefixing with '&').
- Every gadget properties dialogue has **OK** and **Cancel** buttons (see page 428 for more details).

**Opening a gadget properties box**

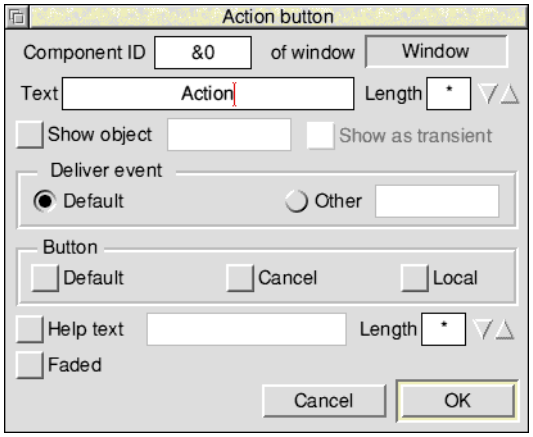
You can open the properties dialogue box for a gadget by double-clicking on the gadget in the Window editor.

The following sections describe in detail the layout and extra controls of each type of gadget properties dialogue:

## Action button properties

Action

The action button properties box is displayed as follows:




The **Show object** option controls whether pressing this button should cause another object to be shown automatically. You can enter the object template's name into the associated writable field, or drag the object template into this field (or onto the associated option icon if the field is faded). This mechanism may be used to make nested dialogues.

**Show as transient** selects whether the object will be shown as a transient or not.

The **Button** section allows you to specify the operation of the action button.

> **Default** controls whether this button is the default for the window it is in. If you select it, the button is given a highlighted border and is activated by any presses of the Return key within its window.
>
> **Cancel** controls whether this button is the cancel button for the window it is in. If this is selected, all clicks on the button cause the window to be closed. Also any Escape key presses when the parent window has the caret cause the Cancel button to be activated.
>
> When you make an action button into the Default or Cancel button for its window, that attribute is removed from the button that previously had it.
>
> If you drag an action button into another window, the editor checks that the strictures regarding Default and Cancel buttons are not violated (that there must be at most one of each). If necessary the previous ‘owners’ of these attributes are made into normal action buttons.

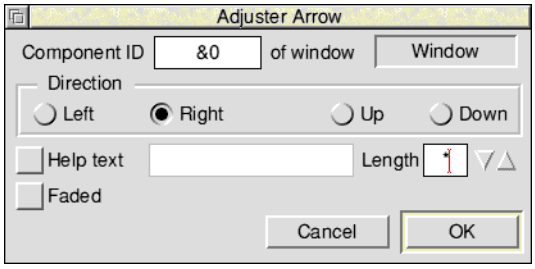
Whenever the Default attribute is added to an action button, its bounding box is automatically enlarged to include the special border, and when the attribute is removed, the bounding box is made correspondingly smaller.

**Local** makes an action button into a Local action button. Unlike a normal action button, activating it will not cause the parent window to be closed.

## Adjuster arrow properties



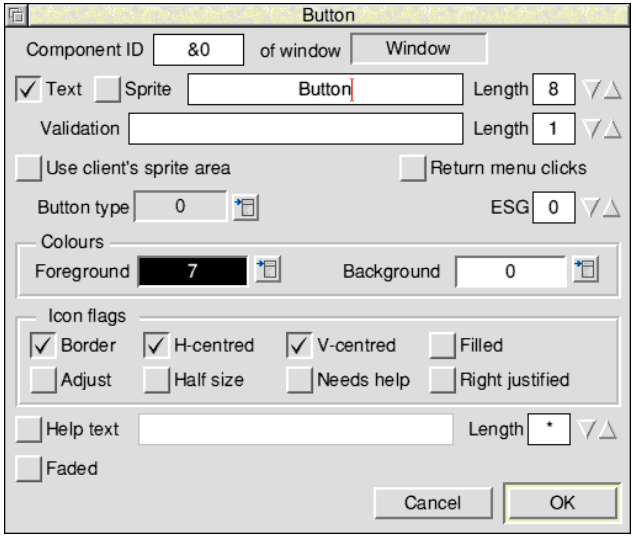
The adjuster arrow properties box is displayed as follows:




The **Direction** radio buttons control the direction that the arrow button is pointing in, and hence whether the button will return ‘up’ or ‘down’ events.

## Button properties


The Button gadget exposes most of the underlying Wimp icon, allowing you to create custom controls. The Button properties box is displayed as follows:



**Text** and **Sprite** are option buttons controlling the contents of the icon. By switching the two buttons on or off, or just switching one of them on, you can produce four combinations. The effects of these various combinations are described in the *RISC OS 3 Programmer's Reference Manual* on page 3-101. If necessary you can then specify a validation string in the **Validation** field. Note, however, that if you only switch on Sprite, then the pointer must be to a sprite name.

**Use client's sprite area** specifies that the Toolbox should first check on those areas set up by Toolbox_Initialise, rather than using the default Wimp Sprite area.

**Return menu clicks** specifies that a Menu click is returned to the client application (instead of being processed and acted upon by the Toolbox).

**Button Type** is a string set offering the sixteen possible Wimp button types:

0 Never8 Double/Drag
1 Always9Menu icon
2 Auto-repeat10Double/Click/drag
3 Click11 Radio
4 Release12Type 12
5 Double click13Type 13
6 Click/Drag14Write/Click/Drag
7 Release/Drag15Writeable

**ESG** is a writeable field for the input of the icon's Exclusive Selection Group number. This number is constrained to be between 0 and 31.

**Foreground** and **Background** offer the choice of the sixteen standard Wimp colours from a pop-up menu. The associated display field shows the chosen colour, as well as the Wimp colour number in a contrasting colour.

The option buttons under **Icon flags** are used to set the remaining icon flag bits that are not implicitly defined by the above settings. The correspondence between buttons and icon flag bits is as follows (see the *RISC OS 3 Programmer's Reference Manual* entry for Wimp_CreateIcon on page 1-93 for more details):

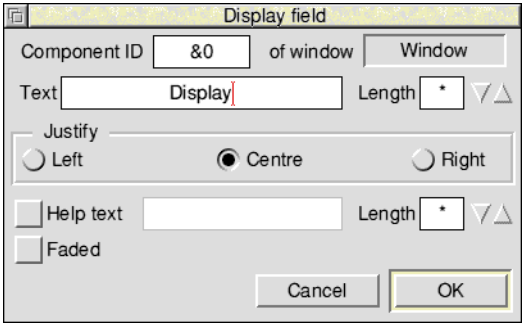
| **Button** | **Bit** |
|---|---|
| Border | 2 |
| H-centred | 3 |
| V-centred | 4 |
| Filled | 5 |
| Adjust | 10 |
| Half size | 11 |
| Needs help | 7 |
| Right justified | 9 |

There are three icon flag bits that are pre-set which you cannot change:

| Bit | Set to |
|---|---|
| 6 | always set to system font |
| 8 | always indirected |
| 21 | always unselected when first displayed |

## Display field properties

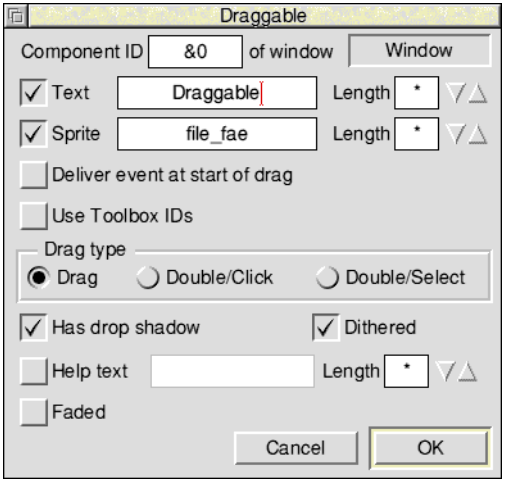

The display field properties box is displayed as follows:




The **Justify** radio buttons are used to choose whether the contents are positioned to the left, right or centre of the gadget.

## Draggable properties


The draggable properties box is displayed as follows:



The Draggable gadget may have a writable text string, a sprite, or both, as chosen by relevant option buttons. At least one of these must be on.

The **Deliver event at start of drag** option allows you to control delivery of the Draggable_DragStarted event.

**Use Toolbox IDs** allows you to specify that object/component id pairs of the drag destination will be reported, rather than Wimp window handle/icon handle pairs.

The **Drag type** radio buttons allow you to select the behaviour of the draggable.

**Drag** provides drag behaviour equivalent to dragging a standard Save As box.

**Double/Click** is equivalent to Icon button type 10.

**Double/Select** is equivalent to Icon button type 8.

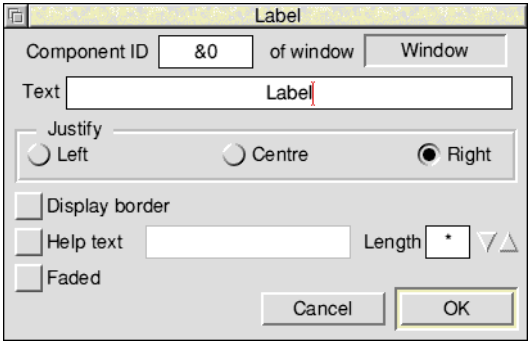
**Has drop shadow** allows you to specify whether the draggable has a grey drop shadow when dragged.

**Dithered** allows you to specify whether the draggable is displayed as semi-transparent when dragged.

## Label properties

Label

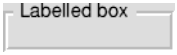
The label properties box is displayed as follows:



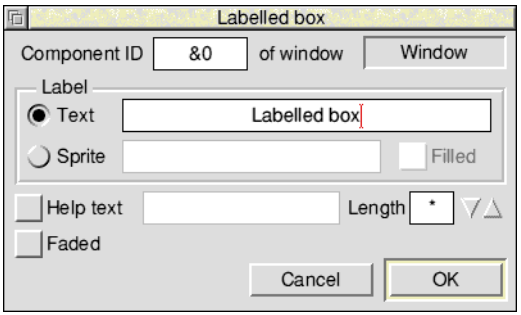

The **Justify** radio buttons are used to choose whether the contents are positioned to the left, right or centre of the gadget.

**Display border** controls whether the gadget's bounding box is drawn or not.

## Labelled box properties




The labelled box properties box is displayed as follows:

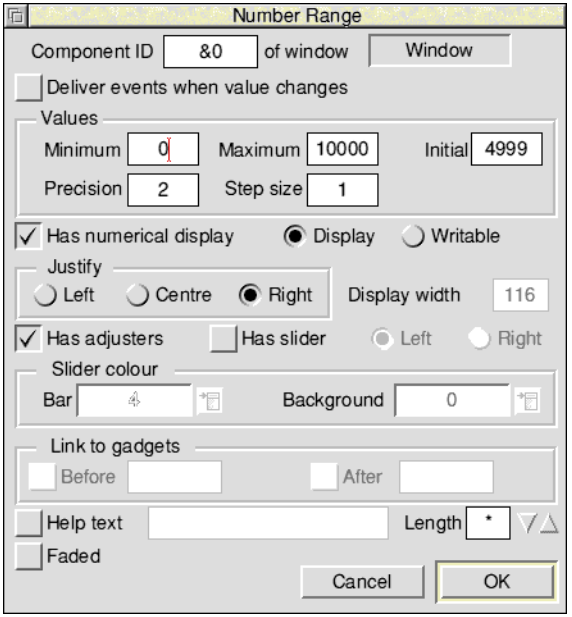



The labelled box can have either a textual or sprite label, but not both. This is chosen using the **Text** and **Sprite** radio buttons. The text entry field next to the unselected radio button is faded.

**Filled** allows you to specify that the background to the sprite is set to grey.

## Number range properties


The number range properties box is displayed as follows:



**Deliver events when value changes** controls whether the application receives NumberRange_ValueChanged events when the contents of the writable change.

**Initial**, **Minimum**, **Maximum** and **Step Size** are writable fields in which you specify the main parameters of the number range. They are always specified as integers.

**Precision** controls the display of a decimal point; its value is the number of digits to be displayed to the right of the point (thus if precision is 2, the value 2.34 is specified as 234). To display integers, set Precision to 0.

**Has numerical display** controls whether any numbers are displayed.

> **Display** and **Writable** select whether the display area may be typed into. If **Writable** is on, the **Link to gadgets** section allows you to specify which gadgets the caret should be moved to when the Tab, Shift Tab, up-arrow and down-arrow keys are pressed. If you drag a gadget into the **Before** or **After** writable fields (or their associated option icons) its component id is entered into the field automatically. Normally, however, you would use the **Link writables** option in the Edit menu to determine the path taken by the caret. See *Moving the caret between writable gadgets* on page 451 for more details.

The **Justify** radio buttons are used to choose whether the numeric value is positioned to the left, right or centre of the numerical display field.

**Display width** allows you to specify the width (in multiples of 4 OS units) of the field that displays the number (only if **Has slider** is switched on).

**Has adjusters** controls whether adjuster arrows are displayed; if selected, they will appear as a pair of buttons to the right of the display area (or, if there is a slider, at either end of the slider).

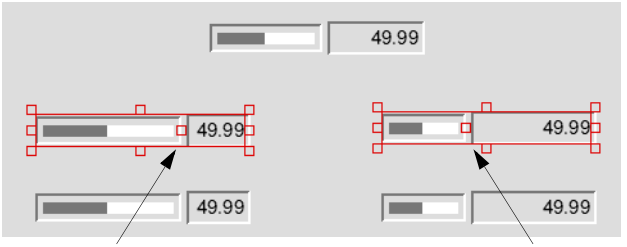
**Has slider** controls the presence and positioning of the gadget's associated slider. The slider is always placed 8 OS units away from the display area, and may be to the left or right of it. The slider will be interactive only if the writable radio button is selected.

The **Slider colour** section allows you to specify the colours of the slider:

> **Bar** is a display field showing the colour of the slider's bar. The colour is set by specifying a Wimp colour number from the attached pop-up menu.
>
> **Background** is a display field showing the background colour of the slider's bar. The colour is set by specifying a Wimp colour number from the attached pop-up menu.

### Altering the size of the numerical field

As well as the normal eight resize handles, number range gadgets which display a slider and numerical display have an additional handle. You can drag this handle to the left or right to adjust the size of the numerical display field:

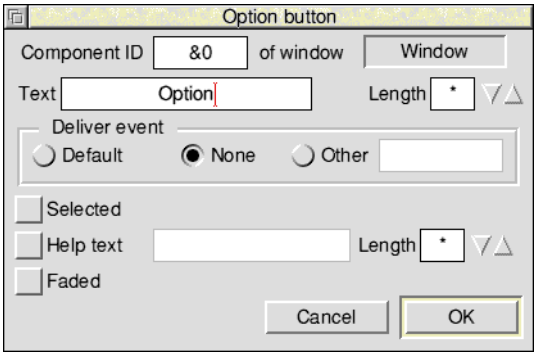



Note: You can only alter the size of the numerical field on one number range gadget at a time. If you try and resize this field on a selection of number range gadgets only the gadget you are actually resizing will be resized.

## Option button properties

Option

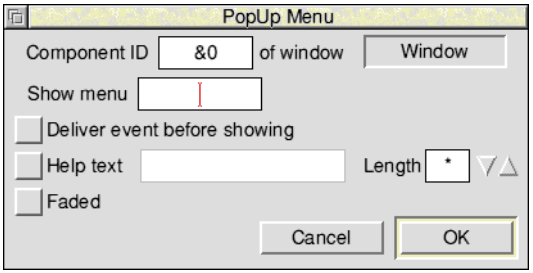
The option button properties box is displayed as follows:




**Selected** chooses whether this button is initially switched on or not.

## Pop-up menu properties

The pop-up menu properties box is displayed as follows:

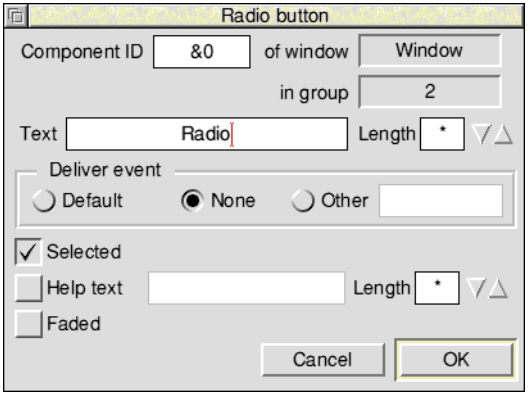



**Show menu** controls whether a menu will be automatically shown when the menu button is clicked. The template name of the menu to be attached may be filled in by dragging a Menu object template to this field. If no Menu object template is supplied, the application will be expected to create it at run-time in response to the PopUp_MenuAboutToBeShown event.

**Deliver event before showing** controls whether the client application will receive a PopUp_MenuAboutToBeShown event when the object is about to be shown.

## Radio button properties


The radio button properties box is displayed as follows:



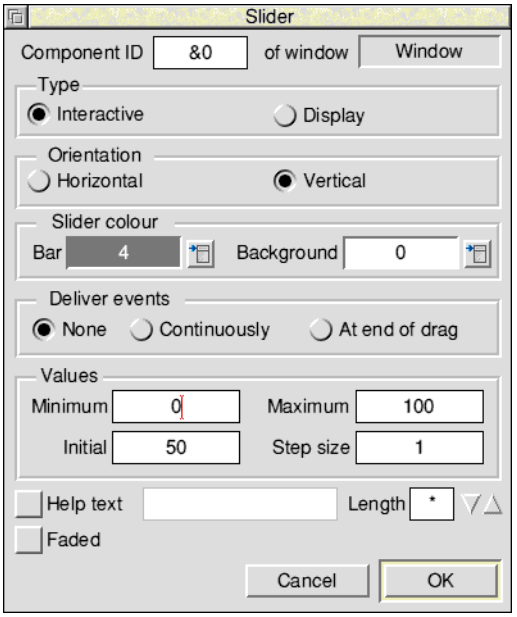

Each radio button is a separate gadget and belongs to a ‘radio group’, this group being the set of radio buttons with which it is mutually exclusive. The radio group is implemented by means of a ‘Group Number’ (see *Radio buttons* on page 387) in the Toolbox data structure that describes the gadget; the group number is not the same as the

Wimp's ESG (which the Toolbox does not use). You cannot specify the group number explicitly, instead you must use the **Make radio group** option in the Edit menu; however, the group number assigned by ResEd is always displayed in the **in group** field.

**Selected** chooses whether the button is initially on or off; only one button in the group may be on at once, and switching another on will turn off the previously-on button.

## Slider properties

The slider properties box is displayed as follows:




The **Type** radio buttons select between a read/write slider and a read-only one.

The **Orientation** radio buttons select whether the slider is horizontal or vertical. When a slider's orientation is changed, it is rotated through 90 degrees about its centre point.

**Slider colour Bar** is a display field showing the colour of the slider's bar. The colour is set by specifying a Wimp colour number from the attached pop-up menu.

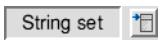
**Slider colour Background** is a display field showing the background colour of the slider. The colour is set by specifying a Wimp colour number from the attached pop-up menu.

The **Deliver events** buttons control when the application will receive Slider_ValueChanged events.

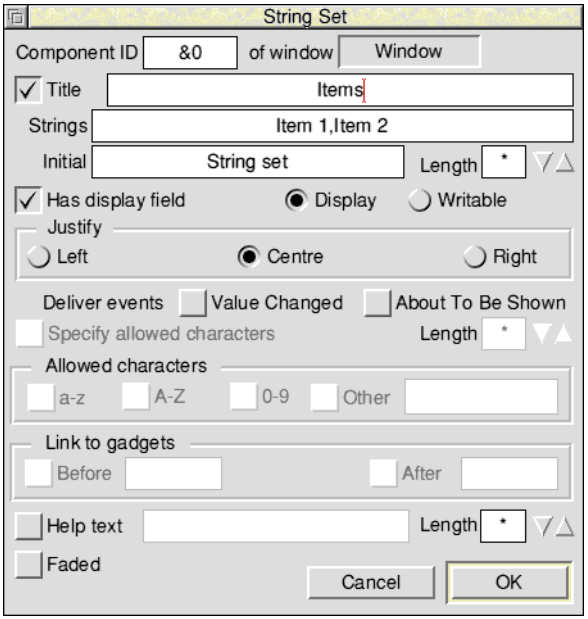
**Minimum** and **Maximum** are the signed integer bounds of the slider's range.

The **Initial** value and **Step size** are constrained to be valid given the current minimum and maximum settings.

## String set properties




The string set properties box is displayed as follows:




To set up a string set, enter the list of available strings into the **Strings** writable field. The list is comma-separated; to include a comma in one of the strings, precede it with a backslash. To include a literal backslash, use two backslashes.

The **Initial** writable field is for entering the string whose value will be used as the initial contents of the string set. This string does not have to be one of the list of available strings.

**Has display field** controls whether any text is displayed.

> **Display** and **Writable** select whether the display area may be typed into. If **Writable** is switched on, the display area of the string set will be writable and the user may enter any desired string into it – not just one of the predetermined choices. Switching on **Writable** also enables you to fill in the **Specify allowed characters** section.

The **Justify** radio buttons are used to choose whether the contents are positioned to the left, right or centre of the display area.

**Deliver events Value Changed** controls whether the application receives StringSet_ValueChanged events when the contents of the writable change.

**Deliver events About To Be Shown** controls whether the client application will receive a StringSet_AboutToBeShown event when the object is about to be shown.

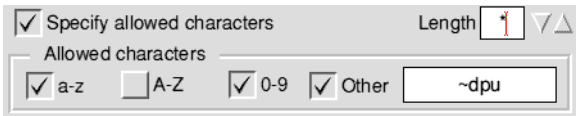
The **Specify allowed characters** section allows you to specify what characters may be typed into the display area. If you do not switch on this option any character will be accepted (before you can fill in the **Specify allowed characters** section you must first switch on **Writable**).

**Length** determines the size of buffer allocated to the validation string.

**Allowed characters** accepts a pattern for the characters that should be allowed in the gadget.

- The three option buttons marked **a-z**, **A-Z** and **0-9** enable you to specify the lower-case letters a-z, the upper-case letters A-Z and the digits 0-9.
- The **Other** option allows you to enter a pattern as for the Wimp's icon validation string ‘A’ command (for more information on the A command see the *RISC OS 3 Programmer's Reference Manual* entry for Wimp_CreateIcon on page 3-102).

For example, if you wanted to specify that the only characters allowed were the digits 0-9 and the lower-case letters a-z, except for ‘d’, ‘p’ and ‘u’, you would fill this section in as follows:

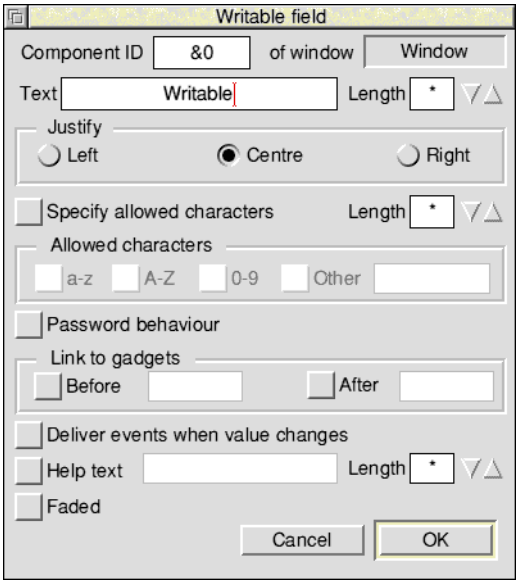



The **Link to gadgets** section allows you to specify which gadgets the caret should be moved to when the Tab, Shift Tab, up-arrow and down-arrow keys are pressed. If you drag a gadget into the **Before** or **After** writable fields (or into the associated option icon if the writable field is faded) its component id is entered into the field automatically. Normally, however, you would use the **Link writables** option in the Edit menu to determine the path taken by the caret. See *Moving the caret between writable gadgets* on page 451 for more details.

## Writable field properties

Writable

The writable field properties box is displayed as follows:




The **Justify** radio buttons are used to choose whether the contents are positioned to the left, right or centre of the gadget.

The **Specify allowed characters** section allows you to specify what characters may be typed into the display area. **Length** determines the size of buffer allocated to the validation string. **Allowed characters** accepts a pattern for the characters that should be allowed in the gadget as for the Wimp's icon validation string ‘A’ command. For a full description of allowed characters see the section on allowed characters on the previous page.

If **Password behaviour** is switched on, then any characters entered will be displayed as minus signs.

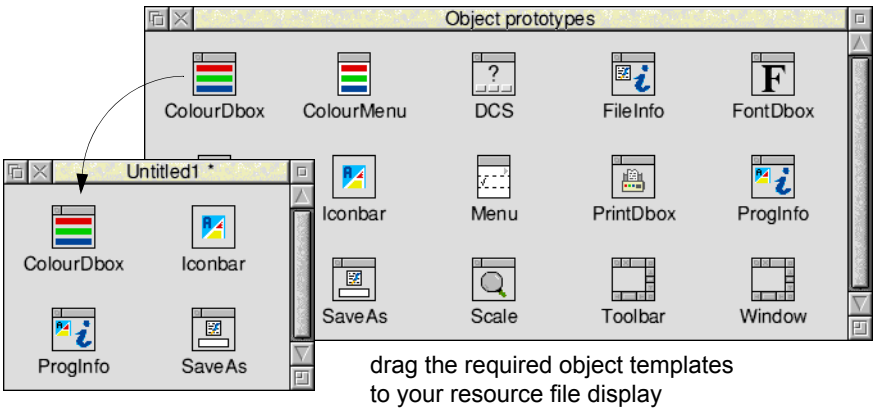
The **Link to gadgets** section allows you to specify which gadgets the caret should be moved to when the Tab, Shift Tab, up-arrow and down-arrow keys are pressed. If you drag a gadget into the **Before** or **After** writable fields (or into the associated option icon if the writable field is faded) its component id is entered into the field automatically. Normally, however, you would use the **Link writables** option in the Edit menu to determine the path taken by the caret. See *Moving the caret between writable gadgets* on page 451 for more details.

**Deliver events when value changes** controls whether the application receives WritableField_ValueChanged events when the contents of the writable change.

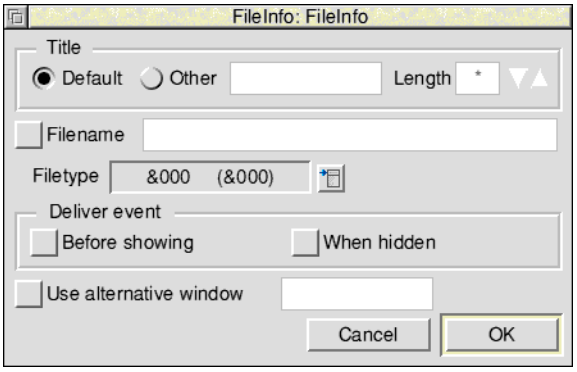
## Editing other classes

There are three stages in editing any of the remaining object templates.

1 Display the object prototypes window and drag the required object templates from the object prototypes window into your resource file display:




2 Edit each object template by double-clicking on its icon in the resource file display. An editing window for that object template will then be opened.
For example, the File Info object template:

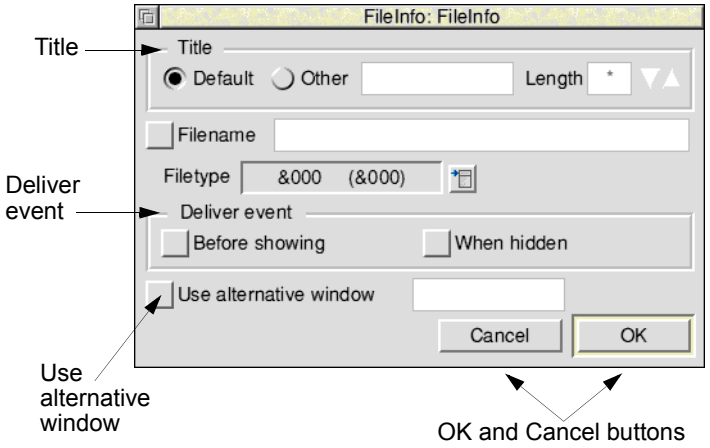



In general the editing dialogue boxes for these remaining object templates are not WYSIWYG representations of the underlying objects.

3 Close the editing window with the **OK** button to confirm the changes you have made. If you close the editing window with the **Cancel** button, the modified data is discarded.

## Common features in standard dialogue boxes and menus

Some features are common to several or all standard dialogue boxes or standard menus. These are described here rather than repeating their descriptions in each individual section:



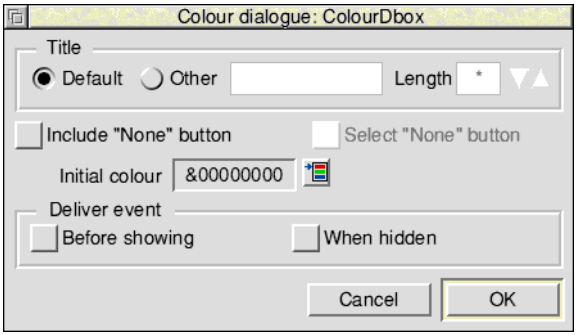

- **Title** is the title string to appear in the title bar of the dialogue box or menu. If this is set to **Default**, the module will provide a suitable default. If it is set to **Other**, the accompanying writable fields are unfaded for you to specify an initial title and its maximum length.
- **Deliver event** controls the following:

  **Before showing** controls whether the client application will receive a DialogueAboutToBeShown event when the object is about to be shown.

  **When hidden** specifies that the client application will receive a DialogueCompleted event when the object is hidden.
- **Use alternative window** is an option button which controls the availability of the writable field next to it. If the option is switched on, you may enter the name of a Window object template to be used as the prototype for creating the relevant object template, instead of the standard one (alternatively you can drag a window object template icon from the resource file display into the writable field – or into the associated option icon if the writable field is faded). This enables any standard dialogue or menu to be given a custom appearance. The custom window must contain gadgets similar to those used in the default module window; see the relevant chapter on the particular module for details.
- Every dialogue box and menu has **OK** and **Cancel** buttons.

## Colour Dialogue class

The Colour Dialogue object template is displayed as follows:



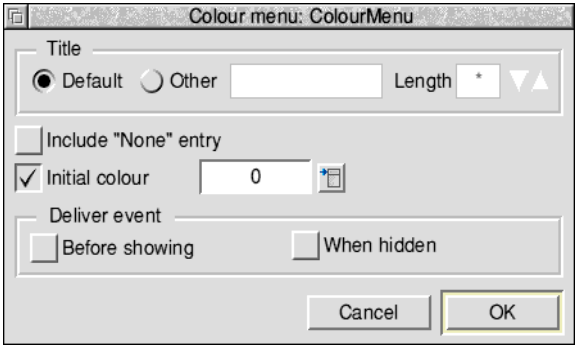

**Include "None" button** is an option button that decides whether the dialogue will allow the choice of ‘no’ colour.

**Select "None" button** specifies that the **None** button is selected by default.

**Initial colour** is a display field that shows the RGB value of the selected colour. Next to it is a pop-up button which summons a colour picker from which the initial colour may be chosen.

## Colour Menu class

The Colour Menu object template is displayed as follows:

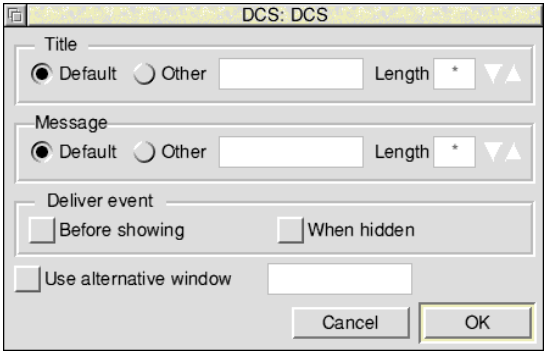



**Include "None" entry** is an option button that controls the presence of an entry for ‘no colour’ (i.e. **None**) on the menu.

The **Initial colour** display field shows the initially-ticked colour, and the pop-up menu to the right of it is itself a colour menu enabling the initial colour to be chosen. The option icon controls whether any value is ticked or not.

## DCS class

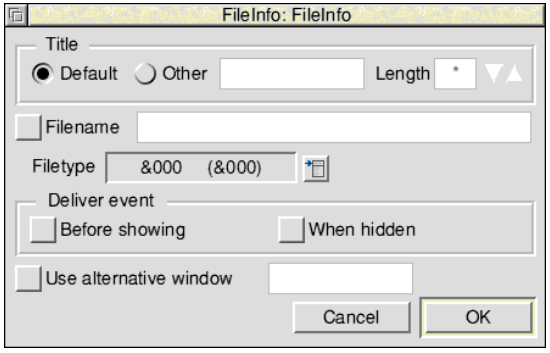
The DCS (Discard, Cancel, Save) object template is displayed as follows:



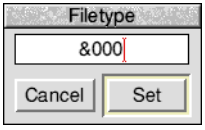

**Message** is a writable field for entering the message to be displayed in the centre of the window. Its behaviour is similar to that of the **Title** field.

## File Info class

The File Info object template is displayed as follows:




**Filename** is a writable field containing the initial contents of the filename display.

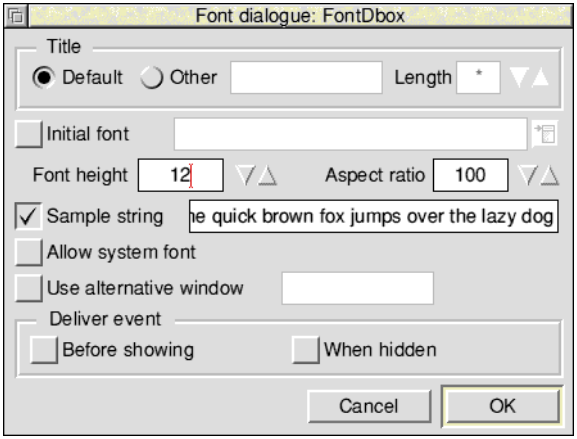



**Filetype** is a display field showing the initial filetype's name and hex value. Next to it is a pop-up menu button which displays a list of filetypes for you to choose from. If you want to specify a filetype not on this list you can go to the Filetype dialogue box (via the **Other** menu option) and fill in the writable field with any filetype name or number. The number must be in decimal unless preceded with '&'. The two special filetypes ‘directory’ (&1000) and ‘application’ (&2000) may also be entered.

Note that no interface is provided for setting the ‘filesize’, ‘modified’ and ‘date’ fields of the File Info object template because these cannot be known when the template is being created. They must be filled in by the application at run-time.

## Font Dialogue class

The Font Dialogue object template is displayed as follows:




**Initial font** is a writable field for you to type in the initial font name to be put into the font dialogue. Alternatively, you can select a font from the pop-up menu next to the writable field. Note that it is possible that the initial font will not be available at run-time; if so, a default will be substituted by the module (as will be the case if the option icon is not switched on).

**Font height** is a number range giving the initial contents of the object's font height setting. You can change the integer value using the adjuster arrows, or type a new value in yourself.

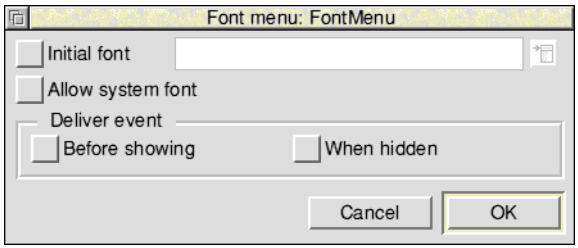
**Aspect ratio** is a number range giving the initial contents of the object's aspect ratio setting. You can change the integer value using the adjuster arrows, or type a new value in yourself.

**Sample string** is a writable field that lets you specify the test string to be displayed when the Font Dialogue's Try button is pressed. If the option icon is not switched on, the module will substitute a default.

The **Allow system font** option button controls whether System Font will be selectable using the Font Dialogue object.

## Font Menu class

The Font Menu object template is displayed as follows:

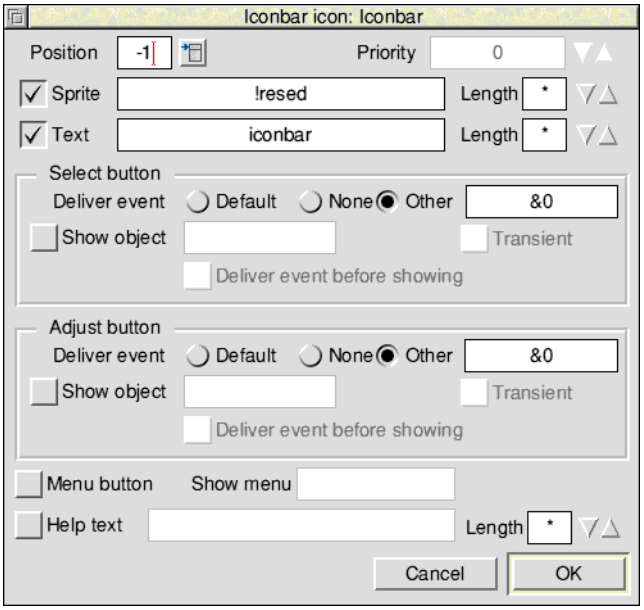



**Initial font** is a writable field for you to type in the initial font name. Alternatively, you can select a font from the popup menu next to the writable field. Note that it is possible that the initial font will not be available at run-time; if so, a default will be substituted by the module (as will be the case if the option icon is not switched on).

The **Allow system font** option button controls whether System Font will be on the menu. If you switch this option on, the **Initial font** menu has System Font on it too.

## Iconbar icon class

The Iconbar icon object template is displayed as follows:




**Position** and **Priority** control where on the iconbar the icon will appear. You can select the position from the adjoining pop-up menu or enter a value directly into the writable field.

| Value | Position |
|---|---|
| -1 | ✓ Right side of iconbar |
| -2 | Left side of iconbar |
| -3 | Left of specified icon |
| -4 | Right of specified icon |
| -5 | Left side, scanning from left |
| -6 | Left side, scanning from right |
| -7 | Right side, scanning from left |
| -8 | Right side, scanning from right |

- Types -3 and -4 require a Wimp icon handle to be passed into the call to Toolbox_ShowObject to specify which icon the position is relative to.

  They are also incompatible with the object's auto-show bit being set, as they depend on a Wimp icon handle being specified in the call to Toolbox_ShowObject. The editor does not force this bit to be clear in these cases; the effect of setting it is undefined.

- Types -5, -6, -7 and -8 require an integer Priority to be specified in the writable field provided. The priority level is as documented in the *RISC OS 3 Programmer's Reference Manual* entry for Wimp_CreateIcon on page 1-93. The Priority field is faded when Position is not set to one of -5 through -8. Priority is normally a decimal integer, but a hex value may be entered by preceding it with an '&'.

**Sprite name** is a writable field where you can enter the name of the sprite to be displayed in the icon. If the icon is to display text as well, you should switch on the **Text** option button. This unfades the two writable fields next to it, enabling you to enter the initial string and maximum length. Switching this option button on sets bit 0 of the object's flags word.

Grouped under **Select button** and **Adjust button** are the controls for specifying what should happen when the user clicks on the icon with the appropriate mouse buttons:

**Deliver event** is a writable field for the input of an event code to be delivered to the application.

**Show object** is a writable field that takes the name of an object template to be shown. You can enter the name of the object template by typing or by dragging an object template into the writable field (or into the associated option icon if the writable field is faded). It is possible to ask for both an event to be delivered and an object to be shown.

The **Transient** option selects whether the object will be shown as a transient or not.

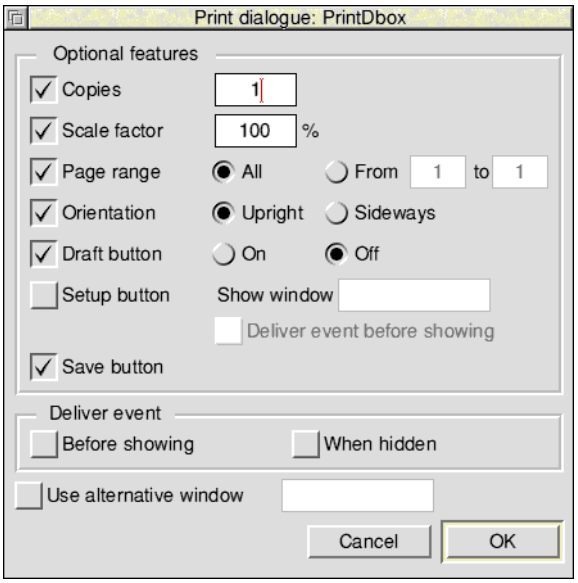
**Deliver event before showing** controls whether the client application will receive an Iconbar_DialogueAboutToBeShown event when the object is about to be shown.

**Show menu** is a writable field for you to specify the name of a menu to be shown when the user clicks in the icon with the Menu mouse button. If the associated option button is turned off, the field is faded and no menu will be shown. You can enter the name of the menu by dragging a Menu object template from the resource file display into the writable field (or into the associated option icon if the writable field is faded).

The writable field next to **Help text** allows you to supply a suitable interactive help string for the Toolbox to send to !Help when the mouse pointer is over the object. If **Help text** is switched off then no help text will be sent.

## Print Dialogue class

The Print Dialogue object template is displayed as follows:




Listed under **Optional features** are a number of option buttons that select which of the optional controls will be present on the dialogue box. Some of these option buttons control the availability of further parameters.

**Copies** selects whether the dialogue box will allow the user to specify the number of copies to be printed. If this is selected, the writable field to its right is unfaded for the initial value of the number of copies to be specified.

**Scale factor** selects whether the dialogue box will allow the user to specify a scale factor for the print job. If this is selected, the writable field to its right is unfaded for the initial value of the scale factor to be specified.

**Page range** selects whether the dialogue box will allow the user to specify the range of pages to be printed. If you switch this option on, the two radio buttons to its right are unfaded for you to specify the default page range. Selecting **All** means that the default will be for all pages to be printed. Selecting **From** means that only a specified range of pages will be printed; this range is specified using the two writable fields (which are faded until **From** is selected.)

**Orientation** selects whether the Print dialogue box will include a choice of **Upright** (portrait) or **Sideways** (landscape) mode. The radio buttons to the right of it are faded unless you switch on this option, and enable you to choose what the default orientation will be.

**Draft button** selects whether the Print dialogue box has a Draft option button or not. The associated radio buttons choose the initial state of the Draft button.

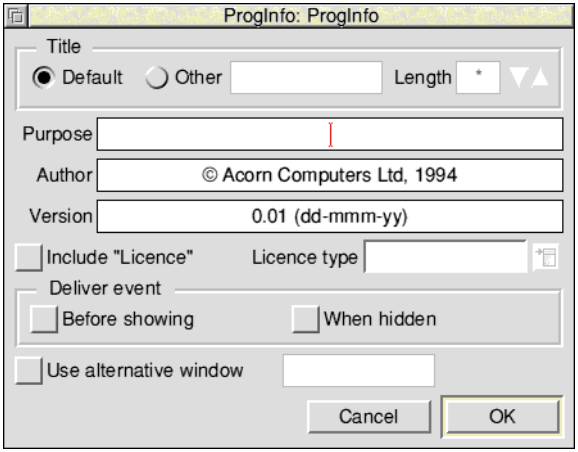
**Setup button** selects whether the dialogue box has a **Setup** button. If you switch this option on, the fields underneath and to the right are unfaded to enable the specification of the following parameters:

> **Show window** is the name of the Window object template to be used for the Setup dialogue. You can enter this by typing, or by dragging a Window object template into the writable field (or into the associated option icon if the writable field is faded).
>
> **Deliver event before showing** is an option button that controls whether a Print_SetUpAboutToBeShown event will be delivered before the Setup dialogue is shown.

**Save button** selects whether the Print dialogue box has a **Save** action button for saving the current printing setup.

## Prog Info class

The Prog Info object template is displayed as follows:

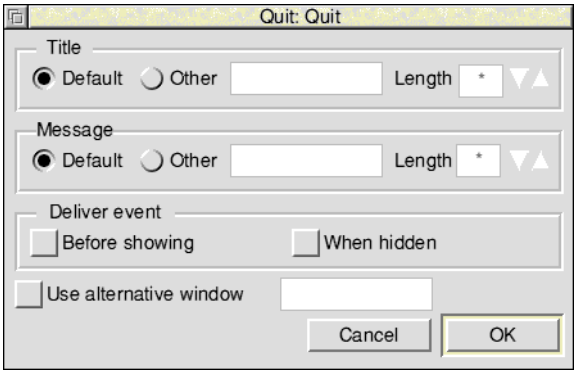



**Purpose**, **Author** and **Version** are writable fields that allow you to specify the contents of the corresponding parts of the Prog Info dialogue box.

**Include "Licence"** is an option button which controls whether the Prog Info dialogue box has a **Licence type** field. If you switch on this option, you can select the licence type from the pop-up menu next to the writable field. The licence types available are **Public domain**, **Single user**, **Single machine**, **Site**, **Network** and **Authority**.

## Quit Dialogue class

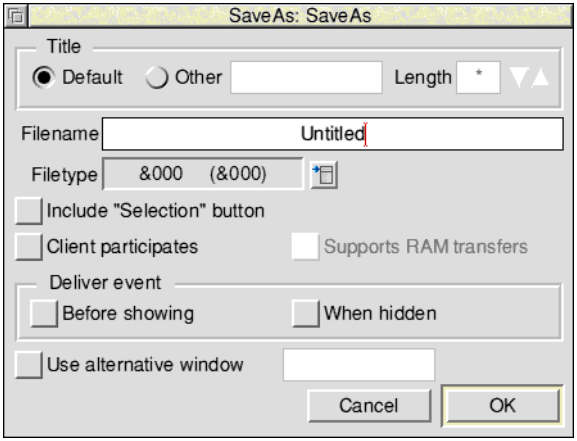
The Quit Dialogue object template is displayed as follows:



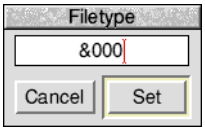

**Message** is a writable field that allows you to enter the message to be displayed in the centre of the window. Its behaviour is similar to that of the **Title** field.

## Save As class

The Save As object template is displayed as follows:




**Filename** is a writable field for you to enter the default filename to be displayed in the dialogue.




**Filetype** is a display field showing the current filetype's name and hex value. Next to it is a pop-up menu button which displays a list of filetypes for you to choose from. If you want to specify a filetype not on this list you can go to the Filetype dialogue box (via the

**Other** menu option) and fill in the writable field with any filetype name or number. The number must be in decimal unless preceded with '&'.The two special filetypes ‘directory’ (&1000) and ‘application’ (&2000) may also be entered.

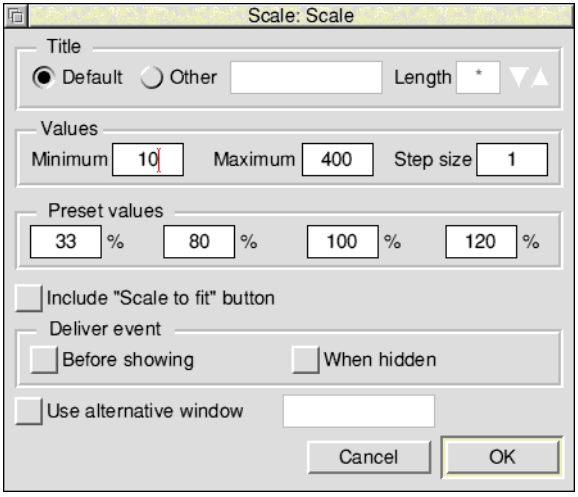
**Include "Selection" Button** is an option button that allows you to control the presence or absence of the Save As dialogue's **Selection** option.

If the **Client participates** option button

- is off, the Save As module will itself handle all data saving on behalf of the client, and the **Supports RAM transfers** option button remains faded.
- is on, the Save As module will involve the client in data saving, using the RAM transfer protocol only when the **Supports RAM transfers** option button is on.

## Scale Dialogue class

The Scale Dialogue object template is displayed as follows:




**Minimum**, **Maximum** and **Step size** are writable integer fields for entering the constraints to be placed on user-specified scale factors.

**Preset values** is a list of four writable fields allowing you to specify the scale factors on the preset size local action buttons.

**Include "Scale to fit" button** is an option button that allows you to control the presence or absence of a **Scale to fit** action button in the Scale Dialogue object.

## Exporting and importing messages

For some purposes, especially internationalisation, you may want to edit the user-visible messages held in a resource file en masse. Rather than manually stepping through every object template in the file, it is useful to be able to edit all the messages in one place. You can do this using the **Export messages** menu item (see page 423). This menu item leads to a Save as box containing a Textfile icon. If you drag this icon into a Filer window or a text editor, ResEd generates a file of messages in MessageTrans format (see the *RISC OS 3 Programmer's Reference Manual* for details).

The file produced contains the messages from each object template in turn. Because these do not have specific tags, a unique tag is generated automatically for each message. These tags take the form:

`<object name>|<number>:`

where

<*object name*>is the name of the object template
<*number*> is the number of the message within that object

You can then edit the resulting message file, and drag it back into the resource file display. A warning is displayed, and you must click on **Import** to proceed.

The messages are matched to their respective objects by use of the information stored in the tags. So, for example, the message

```
SetColours|5:This is the setcolours dialogue
```

will replace the fifth message in the object template whose name is 'Setcolours'. This means that you should take extra care when editing a resource file after its messages have been exported, and before they have been imported back again. Objects should not be renamed, and gadgets within window object templates must not be deleted. On the other hand it is safe to add new templates, or to add new gadgets, or move existing gadgets within a window.

**Note**: it is important that you do not alter any of the tags while editing the messages.

When revised messages are imported, to an object that is currently being edited it is forcibly re-loaded to ensure that its editor is kept up-to-date with the changes. Thus there is potential for you to lose changes made while editing, so care should be exercised when importing message files. Indeed, it is best, before exporting or importing messages, to ensure that there are no unconfirmed changes in any dialogue boxes associated with the file.

## Keystroke equivalents

On occasions, it can be quicker when you are working in ResEd to use the keyboard instead of the mouse, especially when you are familiar with ResEd.

### In the resource file display

| Keystroke | Effect |
|---|---|
| Ctrl-O | open the **Object flags** dialogue box for the selected objects |
| F3 | display a Save As dialogue box |

### In the Window editor

| Keystroke | Effect |
|---|---|
| Ctrl-W | open the **Main properties** dialogue box |
| Ctrl-E | open the **Extents** dialogue box |
| Shift-K | open the **Keyboard shortcuts** dialogue box |
| Ctrl-T | open the **Toolbars** dialogue box |
| Ctrl-G | open the **Gadgets** dialogue box |
| Ctrl-P | open the **properties** dialogue box for the selected gadget |
| Shift-C | open the **Coordinates** dialogue box for the selected gadget |
| Shift-G | open the **Grid** dialogue box |
| Ctrl-S | snap the selected gadgets to the grid |
| Ctrl-R | make the selected radio buttons into a radio group |
| Ctrl-L | link the selected writable gadgets together |
| Ctrl-F2 | close this window |
| Shift-R | show all members of the radio group to which the selected radio button belongs |

### In the Menu editor

| Keystroke | Effect |
|---|---|
| Ctrl-M | open the **Menu properties** dialogue box for editing the top-level characteristics of a menu |
| Ctrl-P | open the **Menu entry properties** dialogue box for the selected menu entry |

### When editing in general

| Keystroke | Effect |
|---|---|
| Ctrl-A | select all entries, gadgets or objects |
| Ctrl-K | delete selected entries, gadgets or objects |
| Ctrl-Z | clear current selection |

## Mouse behaviour

The following mouse actions work on individual menu entries, gadgets or object templates or selections of the same.

Object prototype windows, gadget windows and menu entry windows behave in the same manner as described below, except that, as they are non-editing windows, they do not allow operations such as deletion or repositioning.

### In the Window editor

| Mouse action | Effect | Page |
|---|---|---|
| Double-click | on a gadget to open its properties dialogue box | 460 |
| Drag Select | on a gadget to move it around the window | 450 |
| | or to copy it from one window to another | |
| | or on the resize handle of a gadget to resize it | 451 |
| Drag Adjust | on the resize handle of a gadget to move it in one direction only | 451 |
| Shift-Drag Select | on a gadget to make a copy of it within the window | 450 |
| | or move it from one window to another (deletes the original) | |
| Ctrl-Shift-Drag Select | on a window (with or without a titlebar) to move it around the screen | 443 |
| Ctrl-Shift-Drag Adjust | on a window (with or without an Adjust size icon) to change its size | 443 |

### In the Menu editor window

| Mouse action | Effect | Page |
|---|---|---|
| Double-click | on a menu entry to open its properties dialogue box | 430 |
| Drag Select | on a menu entry to reposition it within the list of menu entries | 433 |
| | or to copy it from one menu to another | |
| Shift-Drag Select | on a menu entry to make a copy of it within the list of menu entries | 433 |
| | or move it from one menu to another (deletes the original) | |

### In the resource file display

| Mouse action | Effect | Page |
|---|---|---|
| Double-click | on a window, toolbar or menu object template to open its editor | 422 |
| | on any other object template to open its properties dialogue box | |
| Drag Select | on an object template to copy it from one resource file display to another | 422 |
| Shift-Drag Select | on an object template to make a copy of it within the resource file display | 422 |
| | or move it from one resource file display to another (deletes the original) | |

### Box selection

The mouse can be used in two ways to select a group of object templates:

- Dragging a box around a group of object templates will select any object template partly or wholly within the Select box.
- Dragging a box around a group of object templates while holding down Shift will select only object templates wholly within the Select box.

Groups of gadgets (in the Window editor) or groups of menu entries (in the Menu editor) can be selected in a similar way.

# 18 ResTest

Having constructed a resource file you may wish to experiment with the interface to ensure that the proper links have been made between the different objects in the file. The resource file test application (ResTest) allows you to

- check the appearance and behaviour of all the objects in your resource file
- monitor the flow of Toolbox and Wimp event codes inside an event log window and, if required, save this event log to a file.

## Starting ResTest

Start ResTest in a similar way to other RISC OS applications, by double-clicking on its application icon. Then drag your resource file (or a selection of object templates from ResEd) to the ResTest iconbar icon.

ResTest will read the resource file and register it with the Toolbox. If your resource file contains any objects marked as auto-create they will be created automatically; any objects marked as auto-create and auto-show will be created and displayed. Thus certain objects in the resource file may appear immediately (e.g. iconbar icons). If these objects are linked to other objects, they will also be created, and these will be shown when you perform the appropriate action. For example, if an iconbar icon is linked to a menu, the menu will be shown when you press the Menu button on the icon. Then if the menu itself is linked to submenus, these will be shown when you traverse the submenu arrows.

## The iconbar menu


Once you have dragged your resource file to the ResTest icon then you can click Menu on the iconbar icon and the ResTest menu will be displayed.

**Info** displays an Info dialogue box.

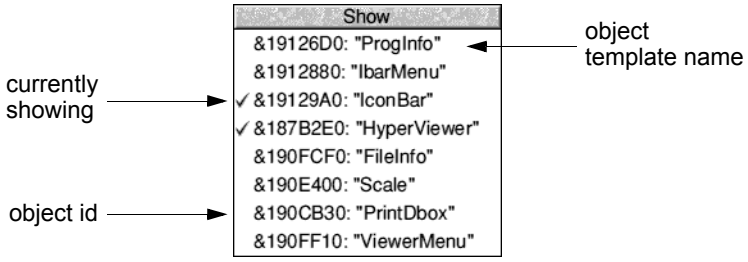
**Create** displays all the object template names in the resource file. Choosing an entry calls Toolbox_CreateObject on that template and creates the object. Shared objects which have already been created are shaded to indicate that they cannot be created more than once.

**Show** displays all the objects that have been created from the object templates. If you go to this submenu immediately after dragging your resource file to ResTest, only two types of object will be displayed:

- those objects marked auto-create
- other objects referenced from those objects (see *Attached objects* on page 11).

So, for example, if the only object marked auto-create was an iconbar icon object, then that object would be displayed, plus the menu object referenced by the iconbar icon object, plus any objects referenced by that menu object. Other objects are added to the Show list as you create them from the **Create** submenu.

Each entry shows the run-time generated object id and the name, or the object template from which it was created. For example:




Entries which are currently showing are ticked. You can cause an unshown object to be shown by clicking Select on its entry, and cause a shown object to be unshown by unticking. Click with the Adjust button causes an object to be shown transiently, and the menu tree will not stay open.

**Delete** displays all the objects that have been created. You can call Toolbox_DeleteObject on an object by clicking on its entry. If the object has unshared children then they are deleted too (a shared object will only be deleted when all its uses are deleted – see *Deleting an object* on page 7).

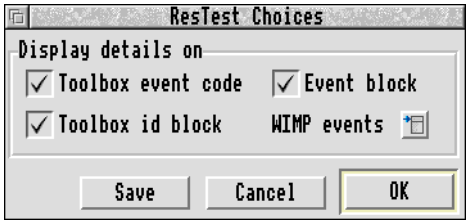
Note: If you delete one or more objects created by a menu object (i.e. attached to the menu object), and then try and delete the menu object itself, you may see the following ResTest error displayed (you should not worry about this error):

```
Invalid Object Id (object id)
```

*`object id`* is the object id of the attached object that was deleted before the menu object was deleted

So, in the example displayed of a Show menu (taken from the example application constructed in the chapter *Building an application* on page 39), if the Scale object were deleted, and then ViewerMenu were deleted (ViewerMenu is the menu object that created the Scale object), then the above error message would be displayed and the object id would be that of the Scale object.

**Choices** displays the following dialogue box:



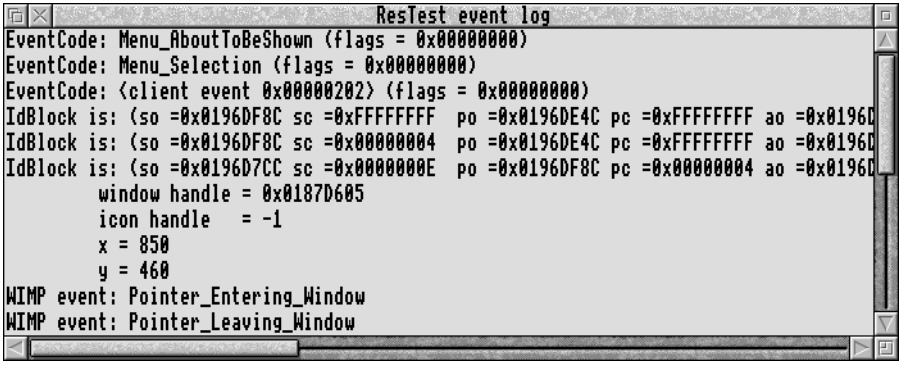

This box allows you to select what information is displayed in the event log window. The options are fully described in the following section *The event log window*.

**Quit** shuts down ResTest, removes all its windows from the screen, and deletes any objects that were created in that session.

## The event log window

If you click Select on the ResTest iconbar icon, the event log window is displayed. This window contains a log of the events received from the Toolbox. You can use this to verify that the proper assignment of events to user actions has been made.

The output in the log window displays four sets of information, depending on what options you have selected from the Choices box in the ResTest menu:




### Toolbox event code

This displays the event code (including client-specified events) and the flags value of the event block. It is always preceded by 'EventCode:'

```
EventCode: Menu_AboutToBeShown (flags = 0x00000000)
EventCode: Menu_Selection (flags = 0x00000000)
EventCode: (client event 0x00000202) (flags = 0x00000000)
```

### Toolbox id block

This displays the contents of the id block. It is always preceded by 'IdBlock:'

```
IdBlock is: (so =0x0196DF8C sc =0xFFFFFFFF  po =0x0196DE4C pc =0xFFFFFFFF ao =0x0196D
IdBlock is: (so =0x0196DF8C sc =0x00000004  po =0x0196DE4C pc =0xFFFFFFFF ao =0x0196D
IdBlock is: (so =0x0196D7CC sc =0x0000000E  po =0x0196DF8C pc =0x00000004 ao =0x0196D
```

where

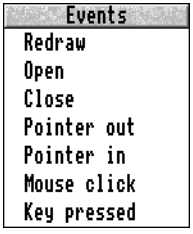
| | | |
|---|---|---|
| so | = | self object |
| sc | = | self component |
| po | = | parent object |
| pc | = | parent component |
| ao | = | ancestor object |
| ac | = | ancestor component |

### Event block

Once an event has occurred (e.g. DragEnded), information about that event is returned in the event block. This information is always displayed indented by eight spaces (how much information is displayed depends on the event):

```
window handle = 0x0187D605
icon handle   = -1
x = 850
y = 460
```

### WIMP events




This option allows you to select various types of Wimp events from the attached pop-up menu. The information displayed is always preceded by 'WIMP event:'.

The following example shows the Wimp events reported when **Pointer in** and **Pointer out** have been selected from the pop-up menu:

```
WIMP event: Pointer_Entering_Window
WIMP event: Pointer_Leaving_Window
```

### The ResTest menu


If you click Menu in the log window the ResTest menu is displayed.

**Save** leads to a **Save as** dialogue allowing you to save the text in the log window to a file.

**Clear** removes any text in the log window.

# 19 DrawFile

DrawFile is a module that renders Draw files.

## Differences between DrawFile output and !Draw output

The following are some small differences between the output of the DrawFile module and !Draw.

### Text

A text line that uses a font which can't be found will be rendered (in system font) at a size to fit its bounding box.

### Transformed text

Transformed text lines in system font are supported. A transformed text line that uses a font which can't be found will be rendered (in system font) at a size to fit its bounding box. The transformation will be ignored.

### Text areas

In a text area, if you change (for example) the margin size (\M command), the change doesn't take effect until the next output line. In Draw, this refers to printable characters: but in DrawFile, it includes colour and font change commands as well (this is because DrawFile uses the Font Manager to remember the current font and colours). This means that line breaks can happen at slightly different places when using DrawFile.

The following commands cause output to occur:

```
B C U V <digits>
```

The following do not:

```
! ; A D F L M P
```

By preceding the former with the latter, the problem can be avoided.

### Sprite colours

For a sprite without a palette, the colours used are the WIMP colours, found by using Wimp_ReadPalette.

# SWI DrawFile_Render

## On entry

R0 = flags:
bit 0 set means render bounding boxes (as dotted red rectangles)
bit 1 set means do not render the objects themselves
bit 2 set means R5 is used as the flatness parameter
R1 = pointer to Draw file data
R2 = size of Draw file in bytes
R3 = pointer to transformation matrix
0 □≠ use identity
R4 = pointer to clipping rectangle in OS units
0 ≠ no clipping rectangle set up
R5 = flatness with which to render lines (if bit 2 of R0 set)

## On exit

All registers preserved

## Use

This SWI renders a Draw file at a given screen position where that position is defined as screen position 0, 0 with the x- and y-translations as specified in the transformation matrix. Hence to render a non-rotated 1:1 Draw file at x, y (screen coordinates in OS units) the transformation matrix is:

$$\begin{pmatrix} 1 << 16 & 0 \\ 0 & 1 << 16 \\ 256*x & 256*y \end{pmatrix}$$

The effects of calling the module with the matrix not of the form:

$$\begin{pmatrix} f & 0 \\ 0 & f \\ x & y \end{pmatrix}$$

(which is a translation and a magnification). If R3 = 0, then unit transformation matrix is assumed (i.e. the Draw file is rendered with its bottom left corner at screen coordinates (0, 0)).

The clipping rectangle is typically a redraw rectangle returned by the Wimp on a redraw window request. If R4 = 0, then the whole Draw file is rendered. If non-zero, only objects which intersect the clipping rectangle are rendered.

### C veneer

```
extern _kernel_oserror *drawfile_render       (int flags, void *data,
                                               int size, Transform *trfm,
                                               BBox *clip,int flatness);
```

## SWI DrawFile_BBox

### On entry

R0 = flags (must be 0)
R1 = pointer to Draw file data
R2 = size of Draw file in bytes
R3 = pointer to transformation matrix
    0 ≠ use identity
R4 = pointer to 4-word buffer to hold the bounding box of the Draw file
    (x0, y0, x1, y1) in Draw units

### On exit

All registers preserved

Buffer pointed at by R4 holds the bounding box of the Draw file (x0, y0, x1, y1) in Draw units

### Use

This SWI is used to determine the bounding box (in Draw units) of the given Draw file, as if it were plotted with the transformation given.

### C veneer

```
extern _kernel_oserror *drawfile_bbox          (int flags, void *data,
                                                int size, Transform *trfm,
                                                BBox *box);
```

# SWI DrawFile_DeclareFonts

## On entry

R0 = flags
    bit 0 set means do not download font (passed to PDriver_DeclareFont)
R1 = pointer to Draw file data
R2 = size of Draw file in bytes

## On exit

All registers preserved

All fonts used by the document have been declared

## Use

If a printer requires font declarations, this SWI must be called for each Draw file to be printed, between the calls to PDriver_SelectJob and PDriver_DrawPage.

All fonts are declared as ‘kerned’, since this includes the non-kerned case.

## C veneer

```
extern _kernel_oserror *drawfile_declare_fonts  (int flags, void *data,
                                                 int size);
```

# Appendix A: Resource File Formats

This appendix describes the resource file format, which is intended to replace the Wimp Template file format, allowing you to specify the appearance of not only window definitions, but also menu definitions and dialogue boxes.

## Terminology

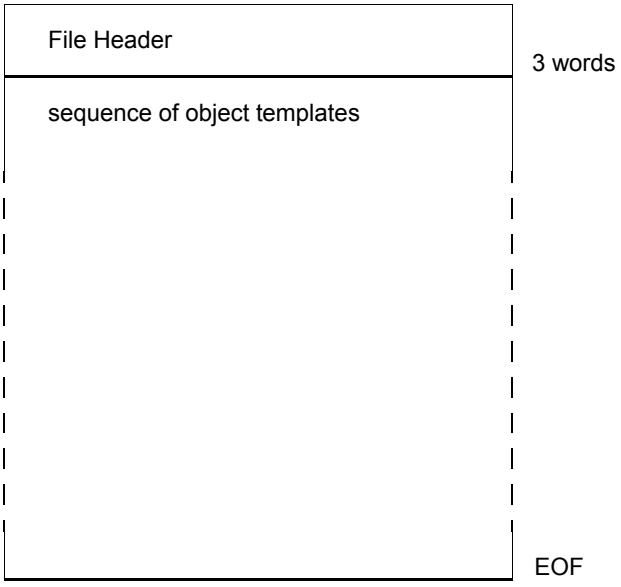
The following terms are used throughout this appendix:

| Term | Meaning |
|---|---|
| word | 4 bytes stored in a file in 'little-endian' format; that is the least significant byte of the word is stored first. |
| resource file | consists of a fixed size header, followed by a contiguous set of user interface object templates or 'objects'. An object consists of a fixed size header followed by the variable size 'body' of the object, followed by 3 tables:<br>string table<br>message table<br>relocations table<br>All object headers are word-aligned. Unless otherwise explicitly stated, all occurrences of a 'word' in this appendix are assumed also to be aligned on a 4-byte address. |
| string | is a sequence of ASCII characters terminated by a NUL character. There is one table per object which holds all such strings.<br>A 'string reference' is given by its byte offset from the start of the strings table.<br>A null string reference is represented by -1.<br>typedef int StringReference; |
| message | is some textual information which is visible to the user. All such messages for an object are held in its Messages Table.<br>A null message reference is represented by -1.<br>typedef int MsgReference; |

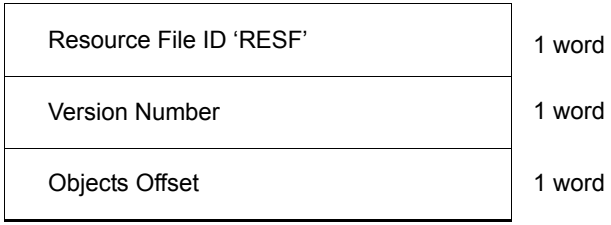
# Resource file format

## Diagrammatic representation

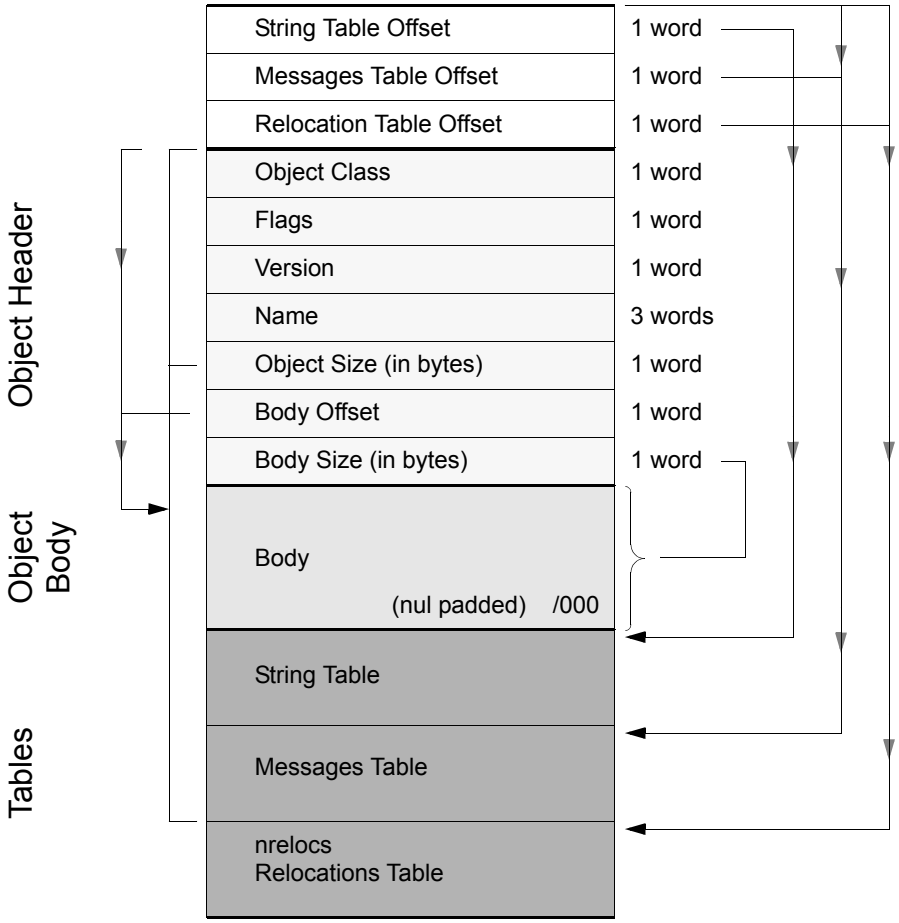
Diagrammatically, a resource file is as follows:




where the file header is:



A resource file containing no objects has an objects Offset of -1 where an object template is:




A String Table Offset of -1 is used to denote an Object Template which has no String Table.

A Messages Table Offset of -1 is used to denote an Object Template which has no Messages Table.

A Relocation Table Offset of -1 is used to denote an Object Template which has no Relocation Table, and hence the nrelocs must always be > 0, if the Relocation Table exists.

When the Resource File is loaded by the Toolbox, the body offset field is always relocated to be a real pointer (but this is not specified as a relocation in the relocation table).

## Resource File Format Description

A resource file begins with a standard fixed size header which has the format:

```
'RESF'                  1 word
Version number          1 word    (* 100, e.g. 109 means 1.09)
Objects Offset          1 word
```

The current version number is 1.01

The objects Offset gives the byte offset from the beginning of the file where the object templates begin.

```
typedef struct
{
    int        file_id;
    int        version_number;
    int        objects_offset;
} ResF_FileHeader;
```

The rest of the file starts with a contiguous sequence of object templates where each template has 3 words giving the byte offsets from the beginning of the template of each of the string, messages and relocations tables, followed by a standard fixed size header, followed by the body of the object, followed by its tables. All object headers are word-aligned.

Where the object header is:

| Field | Type |
|---|---|
| Class of object | 1 word |
| Flags | 1 word |
| Version of the class module required | 1 word |
| Object name | 3 words |
| Total size of object in bytes | 1 word |
| Offset of object body from start of object header | 1 word |
| Total size of object body in bytes | 1 word |

Note that the name of an object is limited to 12 bytes including a terminating NUL character.

'Total size' of object refers to the total size of the object header, the object body and the string and message tables.

'Body size' refers only to the size of the object's body (i.e. without its string and message tables).

```
typedef struct
{
    int                     class;
    int                     flags;
    int                     version;
    char                    name[12];
    int                     total_size;
    int                     body_offset;
    int                     body_size;

} ObjectTemplateHeader;


typedef struct
{
    int                     string_table_offset;
    int                     messages_table_offset;
    int                     relocations_table_offset;
    ObjectTemplateHeader    hdr;

} ResourceFileObjectTemplateHeader;
```

The use of a body_offset field is to allow expansion in the header, without losing backwards compatibility.

## Relocations at Load Time

When the resource file is loaded into memory, the relocations table for each object is used to relocate any string, message, sprite area references and object offsets which appear in the object's body.

This means that the file can be loaded in one operation into memory, and when relocation has been done, the memory can be used directly to create an object.

## Table Formats

There are three tables which optionally appear at the end of an object template: strings table, messages table, and relocations table.

### Strings table

The string table contains all strings which are not visible to the user which are referenced elsewhere in the object. A string is a sequence of ASCII characters terminated by a NULL character.

### Messages Table

The messages table contains a list of strings consisting of text strings which will be visible to the user at run-time, and which are referred to by the object template.

### Relocations Table

The first word of the relocations table gives the number of relocations in the table.

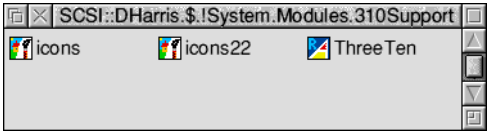
The relocations table contains entries which give the byte offset of a word in the object which should be relocated at load time; this is an offset from the base of the object's body. Each entry is two words long: the byte offset, and a relocation directive. Possible relocation directives are:

| Relocation Directive | Value | Meaning |
|---|---|---|
| StringReference | 1 | add the address of the base of the strings table to this word |
| MsgReference | 2 | add the address of the base of the messages table to this word |
| SpriteAreaReference | 3 | enter the address of the Sprite area into which the client's Sprites file has been loaded |
| ObjectOffset | 4 | add the address of the object's body to this word |

# Appendix B: Support for RISC OS 3.10

This appendix describes the support provided for RISC OS 3.10.

RISC OS 3.10 support is located in `System.Modules.310Support`:




RISC OS 3.10 has the following restrictions which would affect Toolbox applications:

- basic 3.10 does not have 3D icons as standard (e.g. option buttons and radio buttons)
- fading icons on 3.10 is not always consistent (e.g. text label will gain a white box behind the text)
- deleting a window while a ‘slabbed’ button is pressed in will cause a crash.

The ThreeTen module addresses the above restrictions. It is automatically loaded by the Window module when running on a RISC OS 3.10 machine, and also looks for a new version of DragASprite and BorderUtils. It is able to co-exist with New Look.

# Index

# D

# E



# F

# G



# H



# I

# L



# M



# N

# O



# P

# Q



# R

# S

# T

# U



# W

# Reader's Comment Form

*User Interface Toolbox, Issue 1*
*0484,231*

We would greatly appreciate your comments about this Manual, which will be taken into account for the next issue:

**Did you find the information you wanted?**

**Do you like the way the information is presented?**

**General comments:**

If there is not enough room for your comments, please continue overleaf

How would you classify your experience with computers?

☐ **Used computers before** ☐ **Experienced User** ☐ **Programmer** ☐ **Experienced Programmer**

*Cut out (or photocopy) and post to:*

Dept RC, Technical Publications
Acorn Computers Limited
Acorn House, Vision Park
Histon, Cambridge CB4 4AE
England

**Your name and address:**

This information will only be used to get in touch with you in case we wish to explore your comments further