Support Group Application Note
Number: 280
Issue: 0.00
Author:IDJ

Writing Toolbox Object Modules

Applicable Hardware :	All machines running RISC OS 3.10 and later	Related Application Notes:	281

Support Group
Acorn Computers Limited
Acorn House
Vision Park
Histon, Cambridge
CB4 4AE

Contents

1 Introduction

The User Interface Toolbox has been designed to be extensible to support any number of Object Classes by the creation of Object Modules which implement those Classes. ResEd (the resource file editor supplied with Acorn C/C++) was also designed so that editors for new Object Templates can be added without the need to alter the main ResEd binary.

This document describes how to write an Object Module, and how to interface a new editor for the Object Template with the ResEd Shell program.

2 Definition of an Object Module

Associated with each Object Class, there is a module which implements and manages that Class. Each such module is called an Object Module and must comply with the set of conventions laid down in the following sections.

The Toolbox itself knows only about generic Objects, and uses the Object Modules to implement the details of the Object. For example when a Window Object is created using SWI Toolbox_CreateObject, the Toolbox merely creates a new entry in its list of Objects for the current task, and returns a unique ID for that Object; it calls the Window Object Module to actually create the underlying Window Object from its Object Template description; it is the Window Object Module which will call SWI Wimp_CreateWindow to actually create the real Wimp window. Similarly when a Window Object is deleted using SWI Toolbox_DeleteObject, the Toolbox merely removes this entry from its list of Objects for the current task, and calls the Window Object Module to actually use SWI Wimp_DeleteWindow to delete the real Wimp window.

An Object Module must deal with the following (at least):

- registering with the Toolbox as an Object Module supporting a given Object Class
- dealing with methods on Objects of the given Class

- dealing with Wimp Events, Wimp Messages and Toolbox Events
- deregistering with the Toolbox on finalisation
- dealing with client tasks starting up

Typically, an Object Module will register itself with the Toolbox in its initialisation code using SWI Toolbox_RegisterObjectModule (see the section "Registering with the Toolbox as an Object Module"). This tells the Toolbox how to deal with methods on Objects of the given Class. When a method is applied to Objects of the given Class, the module's Class SWI will be called (see the section "Dealing with Methods on an Object"). When the Toolbox itself is initialised it sends round a service call Toolbox_Starting, and Object Modules should register themselves with the Toolbox (as above) when they receive this service call.

Whenever a new Toolbox task starts up, the Toolbox sends round a service call Toolbox_TaskBorn to inform Object Modules of this new task's Wimp task handle. An Object Module should typically keep a linked list of client tasks (remembering their task handles), where each task descriptor has a linked list of the Objects for that task. Note that when a Toolbox method is applied to an Object, the task handle of the calling task is passed to the Object Module thus allowing it to look down its list of task descriptors to find that task.

When a Toolbox task exits, the Toolbox sends round a service call Toolbox_TaskDied, quoting the task's Wimp task handle. This allows an Object Module to delete the task descriptor for that task, and also (more importantly) to delete any Objects which that task owns. the Toolbox will have removed its record of a dying task's Objects, but does not do this via Toolbox_DeleteObject, so Object Modules must tidy up their per-task data structures themselves. In particular, an Object Module should ensure that all memory associated with the dying task is freed, even if it encounters errors while cleaning up.

When the client application polls the Wimp, the Toolbox's Wimp prefilter and postfilter are called. An Object Module can choose to be put on the chain of Object Modules called in this prefilter by calling SWI Toolbox_RegisterPreFilter, specifying a SWI entry point (see the section "Handling Wimp Events, Wimp Messages and Toolbox Events"). Object Modules will use the prefilter mechanism to ensure that the client application has not masked out events which are needed by that Object Module. An Object Module can also elect to be called when specific Wimp Events, Wimp Messages or Toolbox Events are delivered to the client application on exit from SWI Wimp_Poll (ie during the Toolbox's postfilter). This is achieved by calling SWI Toolbox_RegisterPostFilter (see the section "Handling Wimp Events, Wimp Messages and Toolbox Events").

When an Object Module's finalisation code is entered (maybe due to RMKill or because another version of the module has been RMLoaded) it should free up its allocated memory, and call SWI Toolbox_DeRegisterObjectModule (see the section "Finalisation") to cancel its registration with the Toolbox. Note that an Object Module should refuse to finalise (and return a suitable error) if it still has active applications running.

3 Object IDs and Internal Handles

The Toolbox itself maintains a set of data structures associated with each Object for each client task; IDs for these Objects are unique within a task. Also associated with each Object is its "internal" handle; this is the value returned by the Object Module when an Object is created. An Object Module is passed this "internal" handle whenever a method has been applied to one of its Objects.

Given an Object ID, an Object Module can request its "internal" handle from the Toolbox, using SWI Toolbox_GetInternalHandle. In general this should not be necessary because the Toolbox passes the internal handle to an Object module in most cases where it is needed.

SWI Toolbox_GetInternalHandle

Entry:
R0 = flags
R1 = Object ID

Exit:
R0 = "internal" handle

Usage:
This SWI returns the "internal" handle which was passed back to the Toolbox by the relevant Object Module when the Object was created.

4 Registering with the Toolbox as an Object Module

Associated with each Object Class, there is a SWI number which will be called in order to provide the fundamental methods on Objects of that Class. This is referred to as the Class SWI.

When an Object Module starts up, it should register itself with the Toolbox, and declare its Class SWI using SWI Toolbox_RegisterObjectModule.

SWI Toolbox_RegisterObjectModule

Entry:
R0 = flags
R1 = Object Class Number
R2 = Class SWI Number
R3 -> filename where module's resources file is held (or 0 for none).

Exit:
R1-R9 preserved

Usage:
This SWI is called by an Object module to declare that methods on Objects of the given Class, should be done by calling the specified "Class SWI".

The resources held in the file whose name is pointed at by R3 are loaded by the Toolbox, and will be searched when the client makes Toolbox_CreateObject calls. This is primarily used by modules which implement a standard dialogue, and need to hold a Window template for creating the dialogue box itself. For example, the ProgInfo module has a resource file containing two Window Templates called _ProgInfo1 and _ProgInfo2, which it uses for calls to Toolbox_CreateObject to create its underlying Window Objects.

IMPORTANT NOTE: In the current implementation, the Class SWI Number and the Object Class Number must be the same.

5 Dealing with Methods on an Object

Whenever a method is called on an Object of a type for which an Object Module has registered itself with the Toolbox, that Object Module's Class SWI is called.

On entry to the Class SWI, registers R0-R4 have the following contents:

R0 = reason code telling which Toolbox method was called (eg 0 means Toolbox_CreateObject)
R1 = Object ID
R2 = "internal" handle passed back by this Class module when the Object was created
(will be 0 for Toolbox_CreateObject)
R3 = Wimp task handle of client application
R4 -> block containing the client's R0-R9

IMPORTANT: The Class SWI should only alter the contents of the client's register block for a call to SWI Toolbox_ObjectMiscOp (ie reason code 6); for all other methods, it is the Toolbox which fills in these return registers; there is no need to set the client's R0 to point to an error block, in case of an error, since this will be done by the Toolbox, from the error block passed back by the Class SWI.

Register usage in the client's block is as documented in the User Interface Toolbox Manual supplied with Acorn C/C++.

Note that in the list given below, reason code 2 is deliberately undefined.

In order to implement its methods, a Class SWI should have the functionality shown below. Note that any references to registers refer to the register block passed to the Object Module by the Toolbox NOT the user registers (which are held in the block pointed at by the Toolbox's R4), unless otherwise stated.

Reason Toolbox_CreateObject (0)

Exit:
R0 = "internal" handle of created Object.

Usage:
On entry to this method, the user's R1 register will contain a pointer to a template description (not its name). If the client specified a named template to SWI Toolbox_CreateObject, then before calling the Class SWI, the Toolbox replaces this with a real pointer to the template itself.

The Toolbox checks to see if the Object is a shared one which already exists, in which case **it does not call the Class SWI**, but just returns the appropriate Object ID, and increments the reference count.

This method should create a new instance of an Object of its class, and return an "internal" handle for the Object to the Toolbox via R0 (the Toolbox's R0 that is NOT the client's R0). This "internal" handle is the one which will be passed back to the Object Module when the client uses other Toolbox methods on the Object.

If the Object description contains references to other Object templates, then this method should call Toolbox_CreateObject for these, and record the returned IDs.

Reason Toolbox_DeleteObject (1)

Exit:

R1-R9 preserved

Usage:

This SWI should remove the data structures associated with the Object with the given "internal" handle. If this Object contains IDs for other Objects, then this SWI should call SWI Toolbox_DeleteObject for those Objects (unless bit 0 is set).

The Toolbox itself deals with reference counts for Shared Objects, so this method is only called when an Object's reference count reaches 0.

Reason Toolbox_ShowObject (3)

Exit:

If bit 1 of the flags word was set, then R0 should contain a value suitable for use as a Wimp Menu pointer.
R1-R9 preserved

Usage:

This method should display the given Object on the screen. Exact semantics will vary for different Object classes.

Reason Toolbox_HideObject (4)

Exit:

R1-R9 preserved

Usage:

This method should remove a previously displayed Object from the screen. Exact semantics will vary for different Object classes.

Reason Toolbox_GetObjectState (5)

Exit:

R0 = Object state

Usage:
This SWI should fill in R0 with state information for the given Object.

Reason Toolbox_MiscOp (6)

Exit:
R1-R9 preserved.
The user register block (pointed at by R4) should be filled in as appropriate for this particular method.

Usage:
This SWI is used to implement all of the Toolbox_MiscOp methods which are Object-specific.

6 Handling Wimp Events, Wimp Messages and Toolbox Events

The Toolbox registers a single PostFilter and a single PreFilter with the Wimp. Object Modules should register interest in Wimp Events, Wimp Messages and Toolbox Events with the Toolbox, so that their appropriate handler SWIs can be called in the PostFilter. This is done using SWI Toolbox_RegisterPostFilter as documented below. An Object Module can choose to be called in the PreFilter by using SWI Toolbox_RegisterPreFilter. This is useful for modifying the mask passed to SWI Wimp_Poll in case the client application has not enabled a particular Wimp event which is needed by this Object Module.

SWI Toolbox_RegisterPostFilter

Entry:
R0 = flags
bit 0 set means no longer interested in being called by post-filter, for the list of pairs pointed at by R4, and the given SWI number, and the given R2 value.
R1 = SWI number to be called by filter
R2 = one-word value to be passed to the postfilter SWI in R2
R3 = filter type
1 => Wimp Events
2 => Wimp Messages
3 => Toolbox Events
R4 = -> list of pairs of either:
Wimp Event code, Object Class
Wimp Message number, Object Class or
Toolbox Event code, Object Class
depending on the value of R3.
The list is terminated by two words containing -1.

Exit:
R1-R9 preserved

Usage:

This SWI is used to get Wimp Events, Wimp Messages or Toolbox Events delivered to an Object Module. It is used by that module to vet events before they are passed to the client application.

If an Object Module registers for a Wimp Message, then that message is added to those received by the client using SWI Wimp_AddMessages, unless the client was receiving all Wimp Messages anyway. NOTE: an Object Module should **not** call SWI Wimp_AddMessages itself.

Modules which have registered themselves to receive a particular event are called when that event occurs, when the Object ID field in the application's poll block is one for an Object of the specified Class (an Object Class value of 0 means that the Object Module is interested in events on underlying Wimp objects, and an Object Class value of -1 means that the Object Module is interested in events on Objects of ALL classes). For example, the Window Module registers interest in all Wimp Events which can happen on a Window, in order to fill in the Object ID of that Window in the client's ID block; this is done by registering for those Events on underlying Wimp objects using the special value 0, ie:

```
EventInterest   events_of_interest[] =
                {{ Wimp_ERedrawWindow          , 0},
                 {Wimp_EOpenWindow             , 0},
                 {Wimp_ECloseWindow            , 0},
                 {Wimp_EPointerLeavingWindow   , 0},
                 {Wimp_EPointerEnteringWindow  , 0},
                 {Wimp_EMouseClick             , 0},
                 {Wimp_EKeyPressed             , 0},
                 {Wimp_EUserDrag               , 0},
                 {Wimp_EScrollRequest          , 0},
                 {Wimp_ELoseCaret              , 0},
                 {Wimp_EGainCaret              , 0},
                 {Wimp_EUserMessage            , 0},
                 {Wimp_EUserMessageRecorded    , 0},
                 {- 1                          , -1}
                };
```

It also registers interest for Toolbox Events on Objects of the Window Class and Menu Class in order to implement composite Gadgets (eg the code to deal with Number Ranges needs to receive Adjuster_Clicked Events, and the code to deal with String Sets needs to receive Menu_Selection Events), so it uses the following block:

```
EventInterest   toolbox_events_of_interest[] =
          {{Window_AboutToBeShown       , Window_ObjectClass},
           {Adjuster_Clicked            , Window_ObjectClass},
           {Slider_ValueChanged         , Window_ObjectClass},
           {WritableField_ValueChanged  , Window_ObjectClass},
           {PopUp_AboutToBeShown        , Window_ObjectClass},
```

```
        {StringSet_AboutToBeShown    , Window_ObjectClass},
        {Menu_Selection              , Menu_ObjectClass},
        {- 1                         , -1}
       };
```

The SWI whose number is passed in R1 to SWI Toolbox_RegisterPostFilter should have the following behaviour:

Entry:
R0 = Wimp Event reason code returned from Wimp_Poll
R1 = pointer to event block as returned from SWI Wimp_Poll
R2 = value passed in R2 to SWI Toolbox_RegisterPostFilter.
R3 -> 6-word "ID block" as passed to Toolbox_Initialise by the client.

Exit:
R0 contains the Wimp Event reason code returned from Wimp_Poll. In the case of a Toolbox Event, this will be 0x200, and for a Wimp Message it will be the standard values User_Message (0x11) or User_Message_Recorded (0x12). In the case of a Toolbox Event, the event details are held in the event block passed in R1.

If the Wimp Event, Wimp Message or Toolbox Event is for an Object maintained by this Object module and it wishes to indicate which Object it is, this is done by filling in the "self" field in the client's "ID Block". If the Object module has updated the "ID block" then it **MUST** return a non-zero value in R0, to indicate that it has "claimed" the event, otherwise it should set R0 to zero. The Toolbox will deal with filling in the Parent and Ancestor fields in the client's "ID Block", so the Object Module should not alter these fields.

Only one module can claim a particular event on an Object of a particular type. If more than one claim is attempted, then an error is raised. Note, however, that after an event is claimed, that event is still passed on to other modules interested in the event.

The Toolbox will continue to repeatedly pass the event to Object Modules whose criteria passed to SWI Toolbox_RegisterPostFilter are met, and as long as one of these Modules indicates that it has updated the ID block. Great care must be taken here by Object Modules to avoid an infinite loop.

No changes can be made to the event block itself. If the module wishes to cause further events to happen this should be done using Wimp_SendMessage or by raising a Toolbox Event via SWI Toolbox_RaiseToolboxEvent. Before the Toolbox returns to the client, all original register contents are restored, and the Parent and Ancestor fields of the "ID Block" are filled in by the Toolbox.

Example:

The Iconbar module contains the following code:

```
EventInterest   messages_of_interest[] =
                {{ Wimp_MHelpRequest  , 0},
                 {- 1                 ,-1}
                };

EventInterest   events_of_interest[] =
                {{ Wimp_EMouseClick   , 0},
                 {- 1                 ,-1}
                };

EventInterest   toolbox_events_of_interest[] =
                {{ Iconbar_SelectAboutToBeShown,  Iconbar_ObjectClass},
                 { Iconbar_AdjustAboutTobeShown , Iconbar_ObjectClass},
                 {- 1                           ,   -1}
                };
```

These data blocks are passed to SWI Toolbox_RegisterPostFilter and mean that the Iconbar module is interested in being called in the PostFilter when a Help Request message is received on any Wimp object, when a Mouse Click event is received on any Wimp object, and when the Iconbar_SelectAboutToBeShown and Iconbar_AdjustAboutToBeShown Toolbox Events are received on Objects of the Iconbar Object Class.

Note: The Iconbar Module registers interest in the AboutToBeShown events to ensure that the client is delivered this event BEFORE the attached Object is shown. It does this by noting that the AboutToBeShown event has been delivered (in the PostFilter) to the client, and then actually showing the attached Object on the next call to the Iconbar Module's PreFilter. In the PreFilter we are thus assured that the AboutToBeShown event has been delivered, and that the client has called SWI Wimp_Poll since delivery of the event.

For efficiency reasons, we realise that these events can only happen for an Iconbar Object, when that Object is actually showing on the Iconbar, so the Iconbar Object Module registers interest in the above events in its function which deals with showing an Iconbar Object, ie:

```
    regs.r[0] = 0;
    regs.r[1] = Iconbar_PostFilter;
    regs.r[2] = (int)iconbar_icon;
    regs.r[3] = Toolbox_RegisterPostFilter_WimpEvent;
    regs.r[4] = (int)events_of_interest;

    if ((e = _kernel_swi (Toolbox_RegisterPostFilter, &regs, &regs)) != NULL)
        return e;

    regs.r[0] = 0;
    regs.r[1] = Iconbar_PostFilter;
    regs.r[2] = (int)iconbar_icon;
    regs.r[3] = Toolbox_RegisterPostFilter_WimpMessage;
    regs.r[4] = (int)messages_of_interest;

    if ((e = _kernel_swi (Toolbox_RegisterPostFilter, &regs, &regs)) != NULL)
        return e;

    regs.r[0] = 0;
```

```
    regs.r[1] = Iconbar_PostFilter;
    regs.r[2] = (int)iconbar_icon;
    regs.r[3] = Toolbox_RegisterPostFilter_ToolboxEvent;
    regs.r[4] = (int)toolbox_events_of_interest;

    if ((e = _kernel_swi (Toolbox_RegisterPostFilter, &regs, &regs)) != NULL)
        return e;
```

Then in the function which deals with hiding an Iconbar Object, the PostFilters are deregistered:

```
    regs.r[0] = Toolbox_RegisterPostFilter_Remove;
    regs.r[1] = Iconbar_PostFilter;
    regs.r[2] = (int)iconbar_icon;
    regs.r[3] = Toolbox_RegisterPostFilter_WimpEvent;
    regs.r[4] = (int)events_of_interest;

    if ((e = _kernel_swi (Toolbox_RegisterPostFilter, &regs, &regs)) != NULL)
        return e;

    regs.r[0] = Toolbox_RegisterPostFilter_Remove;
    regs.r[1] = Iconbar_PostFilter;
    regs.r[2] = (int)iconbar_icon;
    regs.r[3] = Toolbox_RegisterPostFilter_WimpMessage;
    regs.r[4] = (int)messages_of_interest;

    if ((e = _kernel_swi (Toolbox_RegisterPostFilter, &regs, &regs)) != NULL)
        return e;

    regs.r[0] = Toolbox_RegisterPostFilter_Remove;
    regs.r[1] = Iconbar_PostFilter;
    regs.r[2] = (int)iconbar_icon;
    regs.r[3] = Toolbox_RegisterPostFilter_ToolboxEvent;
    regs.r[4] = (int)toolbox_events_of_interest;

    if ((e = _kernel_swi (Toolbox_RegisterPostFilter, &regs, &regs)) != NULL)
        return e;
```

In the registration calls, the SWI Iconbar_PostFilter will be called when any of the specified events happen on Objects of the specified classes. The value passed in R2 to SWI Iconbar_PostFilter is actually the internal handle of the Iconbar Object; in the Iconbar Module we chose to have one handler registered for each Object since in general, each task will not have a great number of Iconbar Objects showing at any one time. When Iconbar_PostFilter is called it does the following check to see if the event is actually for this Iconbar Object or not; if not it just returns without dealing with the event:

```
extern _kernel_oserror *events_postfilter (_kernel_swi_regs *r)
{
    /*
     * called from the main Toolbox postfilter, when an event happens which
     * this module has expressed an interest in.
     * R0 = Wimp event reason code
     * R1 ->client's Wimp event block
```

```
 * R2 = pointer to icon object
 * R3 ->6-word "ID block" as passed to Toolbox_Initialise
 *
 */

/*
 * This function gets a pointer to an Iconbar Object in
 * R2 (since this was the value passed to Toolbox_RegisterPostFilter).
 * If the event is dealt with by this module (ie ID block gets updated).
 * then set R0 to non-null before return.
 */

int              event_code =  r->r[0];
WimpPollBlock   *block      = (WimpPollBlock *)r->r[1];
IdBlock         *id_block   = (IdBlock *)r->r[3];
_kernel_oserror *e          = NULL;
Object          *icon       = (Object *)r->r[2];

r->r[0] = 0;   /* default is "we haven't handled the event" */

/*
 * for safety we return if NULL handle passed in
 */

if (icon == NULL)
    return NULL;

/*
 * deal with the event
 */

switch (event_code)
{
    case Wimp_EMouseClick:
        if (block->mouse_click.window_handle != WimpWindow_Iconbar)
           break;

        /*
         * first find if it's one of our icons.  If so process the mouse click.
         */

        if (icon->wimp_icon_handle == block->mouse_click.icon_handle)
            e = events__button_click (icon, block, id_block, r);

        break;

   etc............
```

In the **events__button_click** function we actually deal with the mouse click, and update the ID Block with the Toolbox Object ID of this Iconbar Object, as shown below:

```
static _kernel_oserror *events__button_click (Object *icon,
                                              WimpPollBlock *block,
                                              IdBlock *id_block,
                                              _kernel_swi_regs *r)
{
    _kernel_oserror *e = NULL;

    if (block->mouse_click.buttons & Wimp_MouseButtonMenu)
    {
        /*
         * if iconbar object has attached menu, then show it.
         */

        if (icon->menu != NULL)
            if ((e = events__show_menu (icon, block->mouse_click.x, block->mouse_click.y))
                 != NULL)
              return e;

        /*
         * update the id block to say which object the original click was
         * on, so return non-zero R0 to show this.
         */

        id_block->self_id = icon->id; /* we have remembered the object id here */
        id_block->self_component = NULL_ComponentId;

        r->r[0] = 1;
    }
    else if (block->mouse_click.buttons &
            (Wimp_MouseButtonSelect | Wimp_MouseButtonAdjust))
    {
        IconbarClickedEvent clicked_event ;
        _kernel_swi_regs    regs;

        if (((icon->flags & Iconbar_GenerateSelectClickedEvent) &&
             (block->mouse_click.buttons & wimp_SELECT_BUTTON))
              ||
             ((icon->flags & Iconbar_GenerateAdjustClickedEvent) &&
             (block->mouse_click.buttons & wimp_ADJUST_BUTTON))
            )
        {
            /*
             * raise Toolbox Event to client (default is Iconbar_Clicked)
             */

            clicked_event.hdr.size = sizeof(IconbarClickedEvent);
            clicked_event.hdr.event_code = Iconbar_Clicked;

            clicked_event.hdr.flags = block->mouse_click.buttons & 0x0f;

            if (icon->select_event != 0 &&
```

```
            (block->mouse_click.buttons & wimp_SELECT_BUTTON))
            clicked_event.hdr.event_code = icon->select_event;
        else if (icon->adjust_event != 0 &&
            (block->mouse_click.buttons & wimp_ADJUST_BUTTON))
            clicked_event.hdr.event_code = icon->adjust_event;

        regs.r[0] = 0;
        regs.r[1] = (int)icon->id;     /* object id of this iconbar icon */
        regs.r[2] = NULL_ComponentId;  /* no component id */
        regs.r[3] = (int)&clicked_event;

        if ((e = _kernel_swi (Toolbox_RaiseToolboxEvent, &regs, &regs)) != NULL)
            return e;
    }


    /*
     * if there is an Object specified to be shown on this click, either
     * raise a Toolbox event to get it shown, or show it immediately
     * depending on flags settings for iconbar Object
     */

    if ((block->mouse_click.buttons & wimp_SELECT_BUTTON) &&
        icon->select_show != NULL)
    {
        if (icon->flags & Iconbar_GenerateSelectAboutToBeShown)
            e = events__raise_about_to_be_shown (icon,
                                                 icon->select_show,
                                                 Iconbar_SelectAboutToBeShown);
        else
            e = events__show_attached_object (icon, Iconbar_SelectAboutToBeShown);
    }
    else if ((block->mouse_click.buttons & wimp_ADJUST_BUTTON) &&
             icon->adjust_show != NULL)
    {
        if (icon->flags & Iconbar_GenerateAdjustAboutToBeShown)
            e = events__raise_about_to_be_shown (icon,
                                                 icon->adjust_show,
                                                 Iconbar_AdjustAboutToBeShown);
        else
            e = events__show_attached_object (icon, Iconbar_AdjustAboutToBeShown);
    }

    if (e != NULL)
        return e;

    /*
     * update the id block to say which object the original click was
     * on, so return non-zero R0 to show this
     */
```

```
        id_block->self_id = icon->id;
        id_block->self_component = NULL_ComponentId;

        r->r[0] = 1;
    }

    return NULL;
}
```

A different approach was taken in the Window Module, since many Windows may be created and shown at any one time; in the Window Module, there is a single PostFilter registered which scans its linked list of Windows whenever an event is delivered to see which Window the event is for. Note that the calling of an Object Module's PostFilter is via a SWI, so this decision was taken for efficiency reasons.

SWI Toolbox_RegisterPreFilter

Entry:

R0 = flags

bit 0 set means no longer interested in being called by pre-filter, for the given value in R2.

R1 = SWI number to be called by filter

R2 = one-word value to be passed to the SWI (in R2).

Exit:

R1-R9 preserved.

Usage:

This SWI is used indicate that a particular SWI is to be called before the Toolbox PreFilter allows SWI Wimp_Poll to continue.

The given SWI is called with register contents as they will be when Wimp_Poll is called. The SWI can enable any Wimp events which it needs, by zero-ing appropriate bits in R0 (the Wimp_Poll mask). R2 is set to the value passed in R2 to Toolbox_RegisterPreFilter. Note that the value of R0 on return from the given SWI will be passed on to the next SWI in the chain, and will finally be passed to Wimp_Poll.

The purpose of this SWI is to allow Object Modules to be informed when the application calls SWI Wimp_Poll. It may be used to free up memory which is only needed between calls to SWI Wimp_Poll, or to set internal state appropriately.

IMPORTANT NOTE: if an Object Module enables a particular Wimp event which the client was not expecting, then if this event happens, it is delivered to all postfilters, but is NOT passed back to the client. This is important since the client may have problems caused by receiving an event it thinks it masked out!

7 Finalisation

During an Object module's finalisation code, it should do the following:

- refuse to finalise if it has tasks active, otherwise:
- close its messages file
- deregister its messages file from ResourceFS
- deregister its resources file from ResourceFS (if it has one)
- deregister itself from the Toolbox using SWI Toolbox_DeRegisterObjectModule
- free up all its allocated memory.

SWI Toolbox_DeRegisterObjectModule

Entry:
R0 = flags
R1 = Object Class

Exit:
R1-R9 preserved

Usage:
This SWI deregisters a previously registered Object module.

8 Toolbox Service Calls

The Toolbox issues service calls to keep Object Class modules informed of the state of the system. All such modules should listen for these service calls and take appropriate action.

Toolbox_Starting (0x44ec1)

Entry:
R1 = 0x44ec1 (Toolbox_Starting)

Exit:
Do not claim.

Usage:
This service call is passed round when the Toolbox is initialised; it allows Toolbox object modules to call SWI Toolbox_RegisterObjectModule.

Toolbox_TaskBorn (0x44ec2)

Entry:
R0 = Wimp Task handle of newly created task.
R1 = 0x44ec2 (Toolbox_TaskBorn)

Exit:
Module should have noted task's handle.
Do not claim.

Usage:
This service call is passed round after the Toolbox has called Wimp_Initialise on behalf of the client application. It can be used by Object modules to create a new data structure for a client task.

It is also passed round when Toolbox_RegisterObjectModule is called, for each task which the Toolbox currently knows about.

Toolbox_TaskDied (0x44ec3)

Entry:
R0 = Wimp Task handle of dead task.
R1 = 0x44ec3 (Toolbox_TaskDied).

Exit:
Module should have deleted its per-task info.
Do not claim.

Usage:
This service call is passed round when the Wimp_CloseDown service call is received for a Toolbox application.

9 Object Class Numbers and Toolbox Event Codes

Each Object Class has its own Class Number. These numbers are centrally allocated by Acorn; in fact a Class Number is the Class SWI of the Object Module which implements Objects of that Class. For example the Iconbar Module has a SWI chunk starting at 0x82900, and so defines the following SWIs:

```
#define Iconbar_SWIChunkBase    0x82900
#define Iconbar_ObjectClass     Iconbar_SWIChunkBase
#define Iconbar_ClassSWI        (Iconbar_SWIChunkBase + 0)
#define Iconbar_PostFilter      (Iconbar_SWIChunkBase + 1)
#define Iconbar_PreFilter       (Iconbar_SWIChunkBase + 2)
```

Iconbar_ClassSWI is the SWI which will be called for all methods on Objects of the Iconbar Class. Iconbar_PostFilter and Iconbar_PreFilter are the two SWIs which wil be registered with the Toolbox to be called during the Toolbox's PostFilter and PreFilter respectively.

Note that a single Object Module may implement more than one Class, by registering more than one Class SWI from its chunk with the Toolbox, for example the DCS Module implements both the DCS and Quit objects by specifying the following SWIs:

```
#define DCS_SWIChunkBase    0x82a80
#define DCS_ObjectClass     DCS_SWIChunkBase
#define DCS_ClassSWI        (DCS_SWIChunkBase + 0)
#define DCS_PostFilter      (DCS_SWIChunkBase + 1)
#define DCS_PreFilter       (DCS_SWIChunkBase + 2)


#define Quit_SWIChunkBase    0x82a90                /* &10 + DCSSWIChunk */
#define Quit_ObjectClass     Quit_SWIChunkBase
#define Quit_ClassSWI        (Quit_SWIChunkBase + 0)
#define Quit_PostFilter      (Quit_SWIChunkBase + 1)
#define Quit_PreFilter       (Quit_SWIChunkBase + 2)
```

Toolbox Events should also be based on the Class Number. For example the Iconbar Module defines the following Toolbox Events:

```
#define Iconbar_Clicked              (Iconbar_SWIChunkBase + 0)
#define Iconbar_SelectAboutToBeShown (Iconbar_SWIChunkBase + 1)
#define Iconbar_AdjustAboutToBeShown (Iconbar_SWIChunkBase + 2)
```

Error numbers will also be allocated at the same time as the Class Number.

In order to be allocated a Class number you will need to supply:

- details of the Object's Template format
- the API used for Toolbox_MiscOp methods for your Object
- the API for any extra SWIs which the Object Module implements
- the Object Module binary

During development you can use one of the available User SWIs as your Class Number.

The IPR for an Object Module resides with its author, but we encourage free sharing of such Object Modules to ease application development.

10 Interfacing with ResEd, ResCreate and ResTest

ResEd

From version 0.33 onwards, when ResEd is given a template for an Object Class for which it does not have a registered Class Specific Edtior (CSE - see Appendix 1 for more details), it displays a generic Object editing dialogue box which allows the editing of fields of the Object Template using word offsets from the beginning of the Object's body. The Object Template's Class and name will appear in the title bar of the dialogue box used to edit that Template.

If a CSE is required which gives a more WYSIWYG style of editing, then one can be written by following the protocols laid down in Appendix 1, however in most cases this should not prove necessary.

Note that ResEd 0.28 (released with Acorn C/C++) will fault Templates of unknown Object Class. Note also that in future versions of ResEd, we intend to make editing of Object Templates table-driven, thus making the "word offset" approach described above more user-friendly.

It is conventional to use names which start with an underscore for Object Templates which are used by an Object Module (eg the SaveAs module has Window Objects called "_SaveAs1" and "_SaveAs2). Normally !ResEd will fault Template names which do not start with a letter; by changing the !ResEd !Run file to pass the "-private" flag on the command line to !ResEd's !RunImage, underscores are permitted, ie:

```
Run <ResEd$Dir>.!RunImage %*0 -private
```

ResCreate

ResEd is purely an editor of Object Templates and cannot be used to create new Templates; the palette of Object Templates is just a read-only resource file which is created by merging all files named 'Palette' found in its CSE directories.

ResCreate is an application which allows the user to create a "blank" Object Template. Such a Template will have null StringReferences and MsgReferences (see the Resource File Format section of the Acorn C/C++ User Interface Toolbox Manual). This will allow ResEd to be used to edit the Template once it has been created. Typically, a creator of a new Object Module will use this bootstrap mechanism to add knowledge of new Object Template prototypes to ResEd. This can be done as follows:

1. Use ResCreate to create a blank Object Template in a Resource File.

2. Load this file into ResEd, and set up a suitable 'prototype' Object Template, saving this back to the Resource File you have just created.

3. Merge this Resource File into the ResEd !Misc.Palette file.

4. Add a line to ResEd's !Misc.!Config file for the new Object Template of the form:
   <class number>,<template name>,<sprite name>
   where sprite name is the name of a sprite to use in ResEd's prototype window.

5. Create a sprite suitable for this Template and merge it into ResEd's !Misc.!Icons and !Misc.!Icons22 sprite files. The sprite should be called "obj_<name>" following the convention used for existing Object Classes.

ResTest

ResTest can display Toolbox Events for new Object Classes by modifying its Standard and TBlockMess files. This process is explained in Application Note 281 (Writing Tooolbox Gadgets) in the section "Modifying !ResTest".

11 A Generic Object Module

Included with this Application Note is a floppy disc containing an example "Generic" Object Module which doesn't actually implement a new Object Class, but is the skeleton for such a Module. It deals with initialisation, service call handling, SWI dispatching, event handling, task handling, registering with

Toolbox, resource file handling, and message file handling.

The example has a makefile for use when building it. Also on the disc is a program called ResGen (in the library directory) which you should copy onto your Run$Path. This program creates an AOF file given an arbitrary input file, containing an AOF Area and a function to call by which to reference that area. The return value of this function can then be used to register a file with ResourceFS. The Generic Object Module contains an empty Resource File into which an Object Module which uses underlying Objects could place its Templates. For example the SaveAs Module uses a Resource File to store its underlying Window Templates. Many Object Modules will not need an embedded Resource File (eg Iconbar, Menu etc), so this can be removed from such Modules if they are based on Generic.

The code for Generic is supplied "as is" and no guarantees are given as to its correctness, it is purely there to serve as supporting material for this text.

Note: the Object Modules which were released with Acorn C/C++ use a module called TinyStubs to reduce their size by sharing a single copy of the "Stubs" object file, and sharing library static data. This requires a special version of the linker, and may be made available to developers at a later date.

Appendix 1: Adding a Class Specific Editor to ResEd

The ResEd shell provides facilities for creating, copying and moving objects, but does not directly provide facilities for displaying and altering the class-specific data attached to objects. This work is delegated to a number of Class Specific Editors (CSEs). A CSE is a small and specialised WIMP application that provides editing operations appropriate to one or more ToolBox object classes. This architecture has several advantages over the simpler approach of having one big program: each program is manageable in terms of size and complexity; the programs can be loaded incrementally; and new CSEs can be added later as new classes become available.

ResEd communicates with its CSEs by means of a simple message protocol, described fully later in this section. As far as the user is concerned, the separation between the ResEd shell and the various CSEs is not apparent; CSEs are started automatically as required, the intention being to make CSEs look like windows and dialogue boxes of the main ResEd application. As such, CSEs have certain differences from normal WIMP applications:-

- Started automatically by ResEd
- No icon-bar icon
- No explicit loading and saving of data

When the user double-clicks on an object's icon in the document display, ResEd first looks up the object's class number. It generates a command to run the correct CSE based on this number (see later). This command is passed to Wimp_StartTask. ResEd caches the task ID of the resulting task, so that future attempts to edit objects of this class can be passed to the same invocation of the CSE.

ResEd then asks the CSE to edit the object. It does this by sending a message to the CSE; see later for details. The data format used for transferring an object to and from the CSE is identical to the data that would represent the object within a Resource file on disk.

On receiving the object data, the CSE displays whatever editing window or dialogue box is appropriate for the object. As the user makes changes, the CSE informs ResEd that the object has been modified by means of a message. ResEd uses this information to record which objects are out of date in its own data structures. The CSE does not send the updated object back to ResEd (unless explicitly requested to do so by the Shell) until the user clicks on its OK button (or closes the window, for CSEs that feature "direct manipulation" of the object). Transfer back to the ResEd shell is in Resource file format using the same message protocol that was used for loading the object.

Each CSE is capable of editing more than one object at a time. To keep track of objects in the system, the ResEd shell associates an opaque "Document ID" with each loaded Resource file, and an opaque "Object ID" with each object. These IDs are quoted in all messages and together uniquely identify an object in the message protocols between ResEd shell and CSE.

9.1 Document Modified flag

CSE windows fall into two categories according to the style of interaction they use. Simple CSEs are merely dialogue boxes with OK and Cancel buttons. The object data is sent back to the shell when the user clicks OK. Until then, the document window's modified flag is not set. If the user invokes a Save operation from the shell while there are unconfirmed changes in the dialogue box, it is the old copy of the data that gets saved.

More complex CSEs offer direct manipulation of an object's representation (for example, the Window CSE). To give the illusion of the document being changed as the user works, the CSE sends a message to the shell when the first change is made and the shell updates the document's title bar to show the "modified" flag. The modified data is not normally transmitted back to the shell until the user closes the CSE window. However, if the user invokes a Save operation from the shell while there are unconfirmed changes in the window, the shell retrieves the latest version of the object automatically.

It is noted that there is inconsistency between these two categories of CSE window. It is hoped the user's perception that the first type is a "dialogue box" and the second type a "window" will minimise confusion.

9.2 Sprite relocations

Sprite Area Reference relocations in the Resource file format indicate which words in the data need to be set to point to the application's private sprite area when loaded by the ToolBox. CSEs like the Window CSE that allow the user to specify an indirected sprite icon should include a "Private Sprite Pool" option button which controls where the sprite is to be found. If the button is on, the sprite is to be sought in the application's sprite area before looking in the Wimp sprite pool.

If the button is on, the CSE sets the sprite area reference to 0; if it is off, a NULL reference (-1) is stored. In both cases, a corresponding sprite area reference relocation is recorded. When loading the Resource file, the Toolbox will replace a non-NULL sprite area reference by a pointer to the client application's sprite area, and will replace a NULL reference by 1 (meaning the Wimp Sprite pool).

9.3 Multiple versions of classes

The standard object header includes a version number for the object's class. This is included so that future releases of a class module can remain compatible with old object data. The CSE should check the version number of each object it loads, and interpret the object data accordingly, thus offering the same degree of backwards compatibility as the class module. If the object version is more recent than the CSE, the latter should issue an error and refuse to load the object. It is then up to the user to obtain and install a more recent version of the CSE.

9.4 Message Protocols

The interaction between the ResEd shell and the CSEs is managed by a simple message protocol.

The ResEd shell starts CSEs on demand. The first time it needs to edit an object of a particular class, it starts the appropriate CSE as a new task. It records the task ID of the CSE in a table mapping class IDs to CSE task IDs. When the user subsequently double-clicks on another object that can be edited by the CSE, the shell sends the editing request to the same CSE task rather than starting a new one. Thus CSEs must be prepared to edit more than one object at a time.

A CSE can provide editing facilities for a number of classes. This is useful where several different classes have similar characteristics, as they can share a common editor thus reducing memory usage, etc.

Data transfer from the shell to the CSE

RESED_OBJECT_LOAD (Recorded delivery)

R1+0	length of block
R1+4	task handle of sender (filled in by Wimp)
R1+8	my_ref (filled in by Wimp)
R1+12	your_ref (0)
R1+16	RESED_OBJECT_LOAD
R1+20	Flags: Bit 0 set: force re-loading even if object currently open All other bits reserved (should be zero)
R1+24	Document ID: opaque handle chosen by shell
R1+28	Object ID: opaque handle chosen by shell; unique within a document
R1+32	Object's class
R1+36	Object's version
R1+40	Object's address: position of object data in sender's address space
R1+44	Object size: length of object data in bytes
R1+48	Object name (NUL-terminated)

This message is sent by the shell to a CSE to ask it to load an object for editing. This message is sent directly to the CSE (after the the shell has started it, if necessary). The message is sent recorded delivery, and the CSE is expected to reply. If the message bounces, the shell assumes that the CSE has died; it posts a warning message for the user but does not attempt to restart the CSE (to prevent possible looping).

The Document ID and Object ID are opaque handles that uniquely identify the object to the CSE. These IDs are used to identify objects in the following messages, rather than object names.

On receiving this message the CSE should allocate a block of memory of sufficient size, and then obtain the data using Wimp_TransferBlock with the address and size information provided. It should make a note of the document and object IDs so that the object can be looked up in response to future messages. Note in particular that the CSE does NOT use the object's name for this purpose, as this would fail when objects are renamed.

If the CSE receives this message when it already has the specified object loaded, then it should merely raise its editing window to the top of the window stack. However, if bit 0 of the flags word is set, it should re-load the object's data, and update its data structures and windows to reflect this. This is because the shell needs to flush changes after the "edit messages" option is used - see later.

The object name is for the CSE's information only - it may display the name in window titlebars, etc, but must not provide a user-interface for changing the name.

When the CSE has loaded the object data, it should reply with the message RESED_OBJECT_LOADED (even if it merely raised an existing window). If for any reason it was unable to load the data, it should still reply with RESED_OBJECT_LOADED, setting flags bit 0 as described below.

RESED_OBJECT_LOADED (Normal message)

R1+0 length of block
R1+4 task handle of sender (filled in by Wimp)
R1+8 my_ref (filled in by Wimp)
R1+12 your_ref (copied from my_ref of RESED_OBJECT_LOAD)
R1+16 RESED_OBJECT_LOADED
R1+20 Flags:
Bit 0 set: load failed, error code is at R1+32
All other bits reserved (should be zero)
R1+24 Document ID: as quoted in RESED_OBJECT_LOAD
R1+28 Object ID: as quoted in RESED_OBJECT_LOAD

If flags bit 0 is set, the following additional field applies:-
R1+32 Error code

When the CSE has successfully loaded an object in response to RESED_OBJECT_LOAD, it should reply with RESED_OBJECT_LOADED with flags bit 0 clear. This message tells the shell that the transfer was successful.

If the transfer was unsuccessful, the CSE should still reply with this message, setting bit 0 of the flags word to inform the shell that the load failed. The error code may be used by the shell to determine how to handle the error. Valid error codes are:-

0 - out of memory
1 - CSE cannot handle the requested version of this object
2 - invalid or corrupt data
3 - non-fatal internal error
4 - fatal internal error

The CSE should also display an error message to the user explaining why the object was not loaded. The shell will not display any error message in this instance.

Data transfer from the CSE to the shell

Transfer of object data back from the CSE to the shell may be initiated by either party.

i) Initiated by the CSE

RESED_OBJECT_SENDING (Recorded delivery)

R1+0 length of block
R1+4 task handle of sender (filled in by Wimp)
R1+8 my_ref (filled in by Wimp)
R1+12 your_ref (0)
R1+16 RESED_OBJECT_SENDING
R1+20 Flags:
Bit 0 set: cannot send object, error code is at R1+32
All other bits reserved (should be zero)
R1+24 Document ID: opaque handle chosen by shell
R1+28 Object ID: opaque handle chosen by shell; unique within a document
R1+32 Address of object data in CSE's address space
R1+36 Size of object data

If flags bit 0 is set, the following additional field applies:-
R1+40 Error code

When the user clicks the OK button or the Close icon of a CSE window, the CSE needs to transmit the object back to the shell (if it has changed). The CSE sends RESED_OBJECT_SENDING to the ResEd shell task that owns the object, quoting the correct document ID and object ID, and the address and size of the object data. Flags bit 0 of the message shall be clear.

The shell determines which object is being updated. It allocates a buffer of the size quoted in the RESED_OBJECT_SENDING message, and then transfers the data with Wimp_TransferBlock. If this is successful, it then de-allocates the old data associated with the object and stores the newly-received data instead. It clears its internal modified flag for the object and sets the modified flag on the document, placing a "*" in the titlebar.

The shell then replies to the RESED_OBJECT_SENDING message with RESED_OBJECT_LOADED so that the CSE is aware that the transfer was successful. The CSE can now clean up its data structures, close its window etc. If the CSE does not receive a reply to the message, it should just deallocate any temporary storage that it allocated for the transaction.

If the shell is unable to load the object - for example, if it cannot allocate memory - it replies with RESED_OBJECT_LOADED, setting flags bit 0 and inserting a suitable error code into the reply. The shell then displays the error message. This will typically say "There was not enough memory for this operation - please free some memory and try again". On receipt of RESED_OBJECT_LOADED with the error indication, the CSE tidies up any temporary store allocated for the transaction, but does not close the object.

ii) Initiated by the shell

When the shell needs to save the document (or part thereof) it must reclaim the latest version of any objects that are currently being modified in CSEs. This is only done for objects which are known to have been modified (ie RESED_OBJECT_MODIFIED has been received), or for certain "force-loaded" objects (see section 5.2.3). The shell requests the updated object using the following message:-

RESED_OBJECT_SEND (Recorded delivery)

R1+0 length of block
R1+4 task handle of sender (filled in by Wimp)
R1+8 my_ref (filled in by Wimp)
R1+12 your_ref (0)
R1+16 RESED_OBJECT_SEND
R1+20 Flags:
Bit 0 set: delete after successful send (see section 5.2.3)
All other bits reserved (should be zero)
R1+24 Document ID: opaque handle chosen by shell
R1+28 Object ID: opaque handle chosen by shell; unique within a document

The CSE should reply to this message using RESED_OBJECT_SENDING sent recorded delivery, passing the address and size of the object data back to the shell. The transfer then proceeds as detailed in 5.1.1. The shell will reply with RESED_OBJECT_LOADED sent as a normal message (not recorded delivery). The shell should clear it's modified flag for the object, and set the modified flag for the document. When the CSE receives the RESED_OBJECT_LOADED message, it should clear its internal "modified" flags for the object, and deallocate any temporary storage that was used for the transfer. If bit 0 was set in the original RESED_OBJECT_SEND message, the object should now be deleted from the CSE; if not, no further action is necessary (and the window for the object should be left open).

There are several points at which the above transaction could fail.

a) If the CSE is unable to honour the shell's RESED_OBJECT_SEND message, it should reply with RESED_OBJECT_SENDING (sent as a normal message, not recorded delivery). It should set flags bit 0 of this message, and insert an error code. The CSE should also display an error message to the user. The error code may be used by the shell to determine how to handle the error. Valid error codes are:-

0 - out of memory
1 - object is unknown
3 - non-fatal internal error
4 - fatal internal error

On receipt of this message, the shell cancels the operation that caused the object recovery to take place (save, export messages, etc).

b) If the shell is unable to honour the CSE's RESED_OBJECT_SENDING message, it should reply with RESED_OBJECT_LOADED setting flags bit 0 and inserting an error code as described in section 5.1. The shell should then display an error message to the user and cancel the transaction. The CSE should deallocate any temporary storage that was used for the transfer. The CSE may use the error code to determine what went wrong, but it should not display an error message because the shell has already done so.

c) If the shell's RESED_OBJECT_SEND message bounces, it is probably because the CSE has died. It should move on to the next object requiring recovery; tidying up after the dead CSE is done in response to the Wimp_TaskCloseDown message.

d) If the CSE's RESED_OBJECT_SENDING message bounces, it should just deallocate any temporary storage that it allocated for the transaction.

Note: Dialogue-box style CSEs that do not send the message RESED_OBJECT_MODIFIED will only receive RESED_OBJECT_SEND under the circumstances described in section 5.2.3 below.

iii) Importing revised messages

If an object's messages have been changed as a result of importing a revised messages file, then that object must be processed by its CSE to ensure that all associated length fields are consistent with the new messages.

For example, suppose a window's 'helpmessage' field has been changed from "Hello" to "Welcome to my world"; the associated length field 'maxhelp' must now be at least 20, whereas previously 6 bytes were sufficient.

The protocol associated with this process is as follows:

The Shell sends RESED_OBJECT_LOAD to the CSE with flag bit 0 set (force load).

If the CSE already holds a copy of the object, it replaces that copy by the new version and refreshes the associated editing window or main dialogue box, and refreshes or hides any subsidiary dialogue boxes.

If the CSE does not already hold a copy of the object, it loads the object but does not display it (ie no editing window or main dialogue box is opened).

Note that any length fields are modified to be consistent with their associated message fields as part of this (re)loading process.

The CSE replies with a RESED_OBJECT_LOADED message.

If the object concerned was, in any case, being edited by the CSE at the time of message import, no further action is necessary other than for the shell to note that its copy of the object may now be out-of-date.

If, on the other hand, the object was force-loaded into the CSE just to check its length fields, it must now be immediately retrieved - so the shell sends RESED_OBJECT_SEND to the CSE with flag bit 0 set (delete after sending).

The CSE replies with a RESED_OBJECT_SENDING message. It should set the your_ref field of this message from the my_ref field of the RESED_OBJECT_SEND message which was just received.

The shell recovers the revised object data, and replies with a RESED_OBJECT_LOADED message.

The CSE deletes the object, and sends a RESED_OBJECT_CLOSED message to the shell.

Upon receipt of this message the shell marks the object as not being edited and the protocol is complete.

Object status messages

RESED_OBJECT_RENAMED (Normal message)

R1+0 length of block
R1+4 task handle of sender (filled in by Wimp)
R1+8 my_ref (filled in by Wimp)
R1+12 your_ref (0)
R1+16 RESED_OBJECT_RENAMED
R1+20 Flags: all bits reserved (should be zero)
R1+24 Document ID: as quoted in RESED_OBJECT_LOAD
R1+28 Object ID: as quoted in RESED_OBJECT_LOAD
R1+32 New name: NUL-terminated string

Sent to the CSE by the shell when the user renames an object. If the CSE is holding state for this object it should update its record of the object's name, and redraw titlebars etc as necessary. Despite the presence of this message, the CSE should not provide a user-interface to renaming objects, as this is handled centrally by the shell.

RESED_OBJECT_DELETED (Normal message)

R1+0 length of block
R1+4 task handle of sender (filled in by Wimp)
R1+8 my_ref (filled in by Wimp)
R1+12 your_ref (0)
R1+16 RESED_OBJECT_DELETED
R1+20 Flags: all bits reserved (should be zero)
R1+24 Document ID: as quoted in RESED_OBJECT_LOAD
R1+28 Object ID: as quoted in RESED_OBJECT_LOAD

Sent to the CSE by the shell when the user deletes an object. If the CSE is holding state for this object it must delete any windows, de-allocate memory, etc. It should do this silently, without asking about unsaved changes, as this is the responsibility of the shell.

This message is also sent when the user moves a resource from one document to another, or when the document containing the resource is closed. As far as the CSE is concerned, this is indistinguishable from deletion.

The given (Document ID, Object ID) pair will not be re-used.

RESED_OBJECT_MODIFIED (Normal message)

R1+0 length of block
R1+4 task handle of sender (filled in by Wimp)
R1+8 my_ref (filled in by Wimp)
R1+12 your_ref (0)
R1+16 RESED_OBJECT_MODIFIED
R1+20 Flags: all bits reserved (should be zero)
R1+24 Document ID: as quoted in RESED_OBJECT_LOAD
R1+28 Object ID: as quoted in RESED_OBJECT_LOAD

This message allows the shell to track the state of objects when they are altered in a CSE. The shell uses this information to decide whether it needs to recover the object data when the document (or the object's messages) is saved. The message is sent from the CSE to the shell when the user modifies the CSE's copy of an object. The shell records this fact in its data structure describing the object, and also sets the modified flag for the whole document. The CSE should only send this message when the object's state goes from "unmodified" to "modified".

Note: CSEs which have "dialogue box-like" behaviour should never send this message. The transfer of object data from such CSEs is always initiated by the user clicking OK, and the document's "modified" flag is not set until this is done.

RESED_OBJECT_CLOSED (Normal message)

R1+0 length of block
R1+4 task handle of sender (filled in by Wimp)
R1+8 my_ref (filled in by Wimp)
R1+12 your_ref (0)
R1+16 RESED_OBJECT_CLOSED
R1+20 Flags: all bits reserved (should be zero)
R1+24 Document ID: as quoted in RESED_OBJECT_LOAD
R1+28 Object ID: as quoted in RESED_OBJECT_LOAD

Sent from the CSE to the shell when the object's editing window is closed. This also happens when the user clicks SELECT on the "Cancel" button or presses ESCAPE, and also when the user SELECT-clicks on the "OK" button. In the latter case the message should be sent in reply to the RESED_OBJECT_LOADED message sent by the shell.

The CSE should not send this message in response to RESED_OBJECT_DELETED, as the shell expects the window to be closed in response to that message.

Other messages

RESED_SPRITES_CHANGED (Broadcast)

R1+0 length of block
R1+4 task handle of sender (filled in by Wimp)
R1+8 my_ref (filled in by Wimp)
R1+12 your_ref (0)
R1+16 RESED_SPRITES_CHANGED
R1+20 Flags: all bits reserved (should be zero)

Broadcast by the Shell when the user has loaded a Sprites file by dragging it to the iconbar icon. CSEs that are displaying user-specified sprites should redraw their windows on receipt of this message.

RESED_OBJECT_NAME_REQUEST (Normal message)

R1+0 length of block

R1+4 task handle of sender (filled in by Wimp)
R1+8 my_ref (filled in by Wimp)
R1+12 your_ref (copied from my_ref of DataSave message)
R1+16 RESED_OBJECT_NAME_REQUEST
R1+20 Flags: all bits reserved (should be zero)
R1+24 Document ID of requesting object
R1+28 Object ID of requesting object
R1+32 Window handle of destination window
R1+36 Icon handle in destination window (-1 for none)

RESED_OBJECT_NAME (Normal message)

R1+0 length of block
R1+4 task handle of sender (filled in by Wimp)
R1+8 my_ref (filled in by Wimp)
R1+12 your_ref (copied from my_ref of RESED_OBJECT_NAME_REQUEST)
R1+16 RESED_OBJECT_NAME
R1+20 Flags:
Bit 0 set: request refused
All other bits reserved (should be zero)
R1+24 Document ID of requesting object (PRESERVED)
R1+28 Object ID of requesting object (PRESERVED)
R1+32 Window handle of destination window (PRESERVED)
R1+36 Icon handle in destination window (-1 for none) (PRESERVED)
R1+40 Class of dragged object
R1+44 Object name: NUL-terminated string

Many CSE dialogue boxes offer a writable icon for the input of an object name (for example the name of an object to open when an action button is pressed). The user can fill these in by dragging the object's icon from the shell's document display window into the writable icon (or onto an associated option icon if the writable is faded).

When such a drag is initiated by the user, the shell does not know that the destination is such a field, so the shell sends a DataSave message as normal. If the CSE spots that the drag was to an object name writable field, it replies with RESED_OBJECT_NAME_REQUEST rather than the usual DataSaveAck or RamFetch messages. When the shell receives this message, it replies with RESED_OBJECT_NAME. It then considers the DataSave operation complete.

The shell will reply with flags bit 0 set after displaying an error message under the following circumstances:-

- the document ID quoted in RESED_OBJECT_NAME_REQUEST is different from that of the document the object was dragged from
- more than one object was dragged

In this case the shell will abort the DataSave operation, and the CSE should leave the writable field unaltered.

The CSE may check the class field of the reply to ensure that the dragged object was of a suitable type. It may also check the Object ID field to guard against self-referential links.

The "Document ID", "Object ID", "Window handle" and "Icon handle" fields are for the CSE's use - the shell must preserve the contents of these fields when building its reply.

RESED_KEYCUT_DETAILS (Normal message)

R1+0 length of block
R1+4 task handle of sender (filled in by Wimp)
R1+8 my_ref (filled in by Wimp)
R1+12 your_ref (0)
R1+16 RESED_KEYCUT_DETAILS
R1+20 Flags:
Bit 0 set: this keyboard shortcut raises an event
Bit 1 set: this keyboard shortcut shows an object
Bit 2 set: any object shown by this keyboard shortcut is shown transiently
All other bits reserved (should be zero)
R1+24 Task id of shell
R1+28 Window handle of destination window
R1+32 Key code
R1+36 Event code (valid only if flag bit 0 set)
R1+40 Key name: NUL-terminated string
Object name: NUL-terminated string - required only if flag bit 1 set

This specialised message is sent from the Window CSE to the Menu CSE after the user has dragged a keyboard shortcut entry from the Window editor's Keyboard shortcuts scrolling pane to the Menu editor's Menu entry properties dialogue box (see the relevant Functional Specifications for more details of these windows). The purpose of the message is to transmit details of a keyboard shortcut from the Window CSE to the Menu CSE, so that it is easy to make sure that the actions associated with a keypress on a window are the same as those associated with a menu choice.

9.5 CSE start-up

The CSEs are stored inside a subdirectory of the ResEd application. Each contains a file called !Config which the shell uses to determine which CSEs offer editing facilities for which classes.

A CSE is started automatically by the shell the first time it is required. The CSE is started by calling Wimp_StartTask, passing as a parameter the task ID of the shell expressed as a decimal numeric string.

If Wimp_StartTask returns 0, it was not possible to start the task, possibly because there is insufficient free memory. In this case the shell issues a warning to the user and aborts the operation.

Otherwise the value returned by Wimp_StartTask is the task ID of the newly-started CSE. The shell immediately sends RESED_OBJECT_LOAD to the task and remembers the task ID for future editing requests on this class (or any of the other classes that the CSE can handle).

When a CSE is executed, it initially has very little work to do. It first makes a note of the task ID of its parent shell (passed in on the command line). It will need this information to ensure that it only responds to messages from its parent task. Next it calls Wimp_Initialise to register itself as a Wimp task. It next performs any initialisation that it needs for its own internal data structures, and then calls Wimp_Poll. At this point the ResEd shell regains control and is told the task ID of the CSE.

CSEs should use an OS variable of the form <CSE{class }$Dir> to communicate their application directory name from the !Run file into the application, and the application should expand the value and store it before first calling Wimp_Poll. It should use this expanded value rather than the variable itself when referring to the contents of its application directory. This precaution is needed to ensure that CSEs from two different versions of ResEd can co-exist.

9.6 Multiple-class CSEs

To avoid needless code duplication, CSEs may provide editing facilities for more than one class. The shell uses the information in the CSE's !Config file (see Implementation Notes) to determine which classes it can edit.

9.7 CSE death

When the shell receives the broadcast message Wimp_TaskCloseDown, it checks the task ID against all the CSE task IDs it has cached. If the ID matches any of these, then one of the CSEs has unexpectedly died.

The shell removes references to this task ID from its lookup table and resets its private "being edited" flag on any objects that were currently being edited by that task. Subsequent attempts to edit objects of the class handled by the dead CSE will result in the task being restarted. Another circumstance when the shell detects and act on CSE death is when an attempt to send a message to the CSE returns an error.

9.8 Shell death

CSEs should check any Wimp_TaskCloseDown messages they receive. If the task that exited was their parent, they should immediately exit silently. This is the only mechanism to ensure that CSEs do not outlive the shell.

9.9 Drag and drop

Drag-and-drop within a particular CSE can be provided as the author of the CSE sees fit. The only drag-and-drop interaction supported between the shell and the CSE is the dragging of an object from the document window into a CSE dialogue box in order to enter the object's name in a writable field. The messages RESED_OBJECT_NAME_REQUEST and RESED_OBJECT_NAME should be used, as described in an earlier section.

9.10 Standard message protocols

The ResEd shell and its CSEs respond to the following standard message protocols as documented in the PRM:-

- Quit protocol
- Desktop Save protocol
- Shutdown protocol

- Interactive Help protocol

9.11 Implementation notes

i) Filetypes and message numbers

The filetype registered for a Resource file is &fae.

Registered message numbers are:

&83340	MESSAGE_RESED_OBJECT_LOAD
&83341	MESSAGE_RESED_OBJECT_LOADED
&83342	MESSAGE_RESED_OBJECT_SEND
&83343	MESSAGE_RESED_OBJECT_SENDING
&83344	MESSAGE_RESED_OBJECT_RENAMED
&83345	MESSAGE_RESED_OBJECT_DELETED
&83346	MESSAGE_RESED_OBJECT_MODIFIED
&83347	MESSAGE_RESED_OBJECT_CLOSED
&83348	MESSAGE_RESED_SPRITES_CHANGED
&83349	MESSAGE_RESED_OBJECT_NAME_REQUEST
&8334a	MESSAGE_RESED_OBJECT_NAME
&8334b	MESSAGE_RESED_KEYCUT_DETAILS

ii) Directory structure

The structure of !ResEd's directory hierarchy is as follows:-

```
!ResEd     Top-level application run by user
        !Boot                   !Boot file for the shell
        !Help                   !Help file for the shell
        !Run                    !Run file for the shell
        !RunImage               !RunImage file for the shell
        Templates               Wimp templates for the shell
        Messages                Messages and interactive help for the shell
        !Sprites, !Sprites22    Resource filetype and application icon sprites
        Sprites, Sprites22      Sprites used by the shell
        CSE                     Directory containing CSEs
                !Menu           CSE for class Menu
                        !Help           its !Help file
                        !Run            its !Run file
                        !RunImage       its !RunImage file
                        !Config         CSE description file - see below
                        !Palette        Palette entries for the shell - see below
                        !Icons, !Icons22  Sprites file containing class icon(s) - see below
                        Templates       Wimp templates for the CSE
                        Sprites         its sprites (if any)
                        Messages        its messages and interactive help
                        Palette         Palette entries for its palette (if any)
                !Window         CSE for class Window
```

<as for !Menu>
!Misc CSE for other classes
<as for !Menu>

!ResEd.Sprites contains all the sprites used internally by the shell. It does not contain the sprites to be used for the various object's icons; these are loaded from the individual CSEs.

The !ResEd.CSE directory contains all the CSE applications. The names assigned to these are arbitrary. CSE apps contain three special files used by the Shell to determine information about the CSE before it has been started. The shell walks the CSE directory and reads the following files from each:-

!Config: a text file describing this CSE. The file contains one line for each class handled by the CSE, containing three comma-separated fields: the hexadecimal class ID, the human-readable class name, and the sprite name to be used when looking up the class' icon sprite.

eg 0x828c0,Menu,obj_menu

!Icons: sprites file containing the icon sprites for the class(es) edited by the CSE. The shell loads this with *iconsprites.

!Palette: a Resource file containing prototype objects for the classes edited by the CSE. The shell merges the !Palette files from all the CSEs to create its palette.

9.12 General behaviour and standards

ResEd's user interface adheres to the recommendations in the second edition of the RISC OS Style Guide. Also, its editing capabilities are designed to encourage the creation of Style Guide compliant applications.

ResEd's selection model and drag-and-drop system operate according to the following Support Group application notes:-

The RISC OS Selection Model and Clipboard
The RISC OS Drag and Drop System

In particular (and in addition) the following general dialogue box ("dbox") behaviour is supported wherever possible by both the Shell and all CSEs:

Clicking ADJUST on the default action button (eg on "OK") applies a dbox but does not remove it from the screen.

Clicking ADJUST on the cancel action button (eg on "Cancel") resets the state of the dbox to that which it had when it was last opened (or when ADJUST was last clicked on its default action button).

Wherever option or radio buttons control the relevance of other icons in the dbox, these icons are unfaded only when it is sensible to use them; for example, the writable into which Help Text is written is unfaded only when the corresponding option button is ticked.

When a dbox has the input focus:

TAB/SHIFT-TAB or DOWN/UP CURSOR keys will cycle the caret through any unfaded writables.

Pressing RETURN has the same effect as clicking SELECT on the default action button.

Pressing SHIFT-RETURN has the same effect as clicking ADJUST on the default action button.

Pressing ESCAPE has the same effect as clicking SELECT on the cancel action button.

Pressing SHIFT-ESCAPE has the same effect as clicking ADJUST on the cancel action button (but note that the Wimp does not allow this for transient dialogue boxes).

Clicking in any dbox gives it input focus (even if it has no writables).

When a dbox is first opened, input focus is given to it and the caret is placed in the first unfaded writable, if any.

When the state of an option or radio button is changed in such a way that some corresponding writable containing the caret is faded, then the caret is removed from that faded icon and placed in the next unfaded writable, if any.

If an option or radio button associated with a writable is switched on, then the caret is immediately placed in that writable.

When a dialogue box is closed, the input focus is returned to its "parent" window.

Numeric values can normally be entered in either hexadecimal or decimal: the former is indicated by an initial "&". Many values, such as event codes and component ids, are displayed in hexadecimal, this being indicated once more by an initial "&".

Any message or string field inside an object template contains either a string or a NULL value. In some cases there is no difference in behaviour between a NULL value and an empty string, and in such situations ResEd will always save a NULL value (rather than an empty string) since this occupies less space in the Resource File. When there *is* a difference, the CSE will provide an option or radio button whose state determines whether the field is set to an explicit string value or to NULL.

Many object templates include message or string fields which have length fields associated with them. For example, a window object template has a "helpmessage" field with an associated "maxhelp" field. In such cases, the length field determines the size of buffer allocated for the text field whenever an object is created from the template; in other words, it determines the size of the longest string that can ever be assigned to the text field of that object at run-time. Continuing the example, if "maxhelp" is set to 20, then any string set by the Window_SetHelpMessage method must always be at most 19 characters long (one byte must be allowed for the NULL terminator).

In many cases, the client application will never wish to change the text field at run-time, and so the buffer length should be chosen so that the fixed string just fits. To make this easy to do, each CSE allows the user to specify "*" as the value of a length field - and will then set the corresponding field of the object template according to the length of the associated text field value. The precise rules are as follows:

Let N be the value of the length field inside the object template,
and let L, the length of the corresponding text field, be defined as follows:

If the text field is NULL, L = 0.
If the text field contains the string s, then L = strlen(s) + 1.

(Note that it is possible to have a NULL text field with a non-zero length field.)

During the editing process, the value of N can be an integer >= 0 or can be the special value "*". But "*" cannot be represented inside the object template itself, and so translation has to take place when the object is loaded into the CSE, and when it is returned to the shell.

When an object is loaded into a CSE for editing, N is set to "*" if and only if N = L.

When an object is returned to the shell, any "*" value is replaced by 0 (if the text field is NULL) or by strlen(s) + 1 (if the text field contains a non-NULL value s).

The loading process also makes sure that every length field is at least large enough to contain its corresponding string: if N < L, then N is set to "*". (This is how CSEs restore consistency after message import.)

When editing a length field in a dialogue box, the user may enter any positive integer or an asterisk. When he applies the dbox (by clicking on OK) the CSE again ensures that the field is large enough for the corresponding string: any numerical value entered by the user that is too small is replaced by the minimum value necessary (ie strlen(s) + 1).

Other user interface behaviour common to all components of ResEd includes:

Pressing ESCAPE will cancel any user drag or "lassoo" operation, and will hide any menu on display.

Keyboard shortcuts are inoperative during a drag or lassoo interaction, and will cancel any menu on display.

Pressing any one of the four cursor keys during a drag or lassoo interaction will "nudge" the pointer in the corresponding direction by 4 OS units.

All coordinates (position and size) are forced to be exact multiples of 4 OS units.

"Auto-scrolling" is supported for all drag and lassoo operations. If the mouse pointer remains close to the edge of a window for long enough, the pointer changes shape and the window will scroll in that direction; the speed of scrolling increases as the pointer gets closer to the edge of the window. This means that the user must drag decisively in order to move an object from one window to another.

Clicking on an adjuster arrow changes the associated numerical value by a small amount (usually 1); if SHIFT is held down at the same time, the associated value is changed by a larger amount (usually 10).