Support Group Application Note
Number: 281
Issue:0.01
Author:NK

Acorn

Writing Toolbox Gadgets

Applicable Hardware :	Any RISC OS computer running RISC OS 3 (version 3.10) or later.	Related Application Notes:	280

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

Contents

Introduction

This document assumes a familiarity with the RISC OS Toolbox Environment and writing RISC OS relocatable modules. Details of the Toolbox may be found in the 'User Interface Toolbox Manual' which is part of the Acorn C/C++ product (SKB78). All of the examples given within this document are written in C and Acorn encourages developers to use the C language to write new Modules, however it is possible to write them in assembler and the Appendix at the end of this document will describe the SWI calls that are necessary to interface to the environment.

This application note describes how new Gadgets may be added to the RISC OS Toolbox environment. The Window module (version 1.29 and above) has a number of SWIs that allow new Gadget types to be added to the environment at run time. A new Gadget type is implemented with a RISC OS relocatable module which registers itself with the Window module and is then called to perform various services when the Window module encounters a Gadget of this type.

Most of the document describes how one of these modules can be written. The remainder explains how to modify !ResEd and !ResTest to allow the new Gadget(s) to be created and tested and finally an Appendix defines the Window SWIs required and the C library which uses them (GLib).

A simple Gadget

This section describes how a simple gadget (in this case a 'Tool' button that one might find on a Toolbar such as the drawing tools in !Draw) is implemented as a C relocatable module using GLib. It is essentially a sprite option button in that clicking on it with the mouse will change its state from one sprite to another. It will also raise a Toolbox event on the Gadget when the state changes.

The first thing to do is to write a public header file which defines the template for the Gadget type, the events associated with it and any other public information such as the valid flag bits. There should also be a private header to be used by the module which prototypes the handlers for instance.

ToolButton.h

```
/*--------- Tool Button ----------
 * ToolButton.h
 */


/* template definition */

typedef struct
{   GadgetHeader hdr ;
    char         *sprites;
    int          type;
    int          event;
} ToolButton;

#define ToolButtonValidFlags            0xC0000000

/* tool button methods */

#define ToolButton_Base                 0x49d0
/* none defined yet */

#define ToolButton_Type  (sizeof(ToolButton)) << 16    | ToolButton_Base

/* tool button events */

#define ToolButton_StateChanged          (GTest_00)

typedef struct
{       ToolboxEventHeader hdr;
        int                new_state;
} ToolButtonStateChangedEvent;

#define ToolButton_StateChanged_Adjust  0x00000001
#define ToolButton_StateChanged_Select  0x00000004
```

ToolBtnP.h

```
/*
 * ToolBtnP.h
 */

_kernel_oserror *toolbutton_add(ToolButton *template,int wimpw,int **icons,int **data);
_kernel_oserror *toolbutton_mclick(int *handle,WimpMouseClickEvent *click,ObjectId win,int
*claim);
```

After having set up a suitable CMHG file (in this case for a module GTest) the various Module handlers can be written. At module initialisation and on receiving the service call Service_WindowModuleStarting the GTest module needs to register itself with the Window module. Similarly at finalisation the module deregisters itself (Note the use of a finalisation handler rather than atexit).

The registration process works as follows: Any Gadget type has a set of 'features', which refer to the services that the module implementing it is capable of. In the example below the Toolbutton Gadget can be created, removed and can handle mouse clicks. This is identified by setting various bits in a *Features Mask*: Initially the mask is zero which implies that no features are defined for this type. By setting the add and mclick bits to PRIVATE_HANDLER it signifies that the module has handlers for adding and handling clicks on this type of Gadget.

The remove bits are set to DEFAULT_HANDLER which means the Window module will handle all the actions associated with Gadget removal, ie. deleting icons, freeing memory etc. Note that this may only be used if the Gadget uses the GLib functions (or their SWI equivalent) for icon creation and memory allocation. The obvious benefit here is that less code requires to be written and removal of Gadgets is quicker as the Window module does not need to call out to another module. Note that not all feature bits can be set to DEFAULT_HANDLER, for instance, there is no default add_gadget handler. The possible values for each of the feature bits is described in the GLib appendix.

The only other type of handler is NO_HANDLER which implies that this module provides no facilities for this operation and that the Window module should do nothing either. In this simple case the method bits are zero (ie. NO_HANDLER) and so this module provides no methods.

The registration also tells the Window module what SWI to call out to and what flags (from the Gadget header such as the faded bit) are valid from this type of Gadget. The latter simplifies the Gadget adding handler.

from ModHdr

```
title-string:                 GTest
help-string:                  Gadget_Test 0.01

initialisation-code:          GTest_init
finalisation-code:            GTest_final

swi-chunk-base-number:        0x49d00
swi-handler-code:             GTest_SWI_handler
swi-decoding-table:           GTest,ToolButton

service-call-handler:         GTest_services 0x82881
```

from main.c

```
static void register_gadgets(void)
{
    FeatureMask features ;

    features.mask = 0;
    features.bits.add    = PRIVATE_HANDLER;
    features.bits.mclick = PRIVATE_HANDLER;
    features.bits.remove = DEFAULT_HANDLER;
    register_gadget_type(0,ToolButton_Type,ToolButtonValidFlags
                          ,features.mask,GTest_ToolButton);

}

extern _kernel_oserror *GTest_init(char *cmd_tail, int podule_base, void *pw)
{
    IGNORE(cmd_tail) ;
    IGNORE(podule_base);
    IGNORE(pw);

    register_gadgets();
    return NULL;
}

extern void GTest_services(int service_number, _kernel_swi_regs *r, void *pw)
{
    IGNORE(pw) ;
    IGNORE(r);

    switch (service_number)
    {
        Service_WindowModuleStarting:
            register_gadgets();
            break;
        default:
            break;
    }
}

extern _kernel_oserror *GTest_final (int fatal, int podule, void *pw)
{
     IGNORE(fatal) ;
     IGNORE(podule);
     IGNORE(pw);

     deregister_gadget_type(0,ToolButton_Type,GTest_ToolButton);
     return NULL;
}
```

The Window module will now SWI the GTest module whenever a Toolbutton Gadget is created or clicked on. The SWI handler is as follows...

```
extern _kernel_oserror *GTest_SWI_handler(int swi_no, _kernel_swi_regs *r, void *pw)
{
    _kernel_oserror *e = NULL;

    IGNORE(pw);

    switch (swi_no) {
        case 0:
            switch (r->r[2]) {
              case GADGET_ADD:
                e = toolbutton_add((ToolButton *) r->r[3],r->r[5],
                         (int **) &(r->r[1]),(int **) &(r->r[0]));
                break;
              case GADGET_MCLICK:
                e = toolbutton_mclick((int *) r->r[3],
                         (WimpMouseClickEvent*) r->r[6], (ObjectId) r->r[4], &(r->r[1]));
                break;
              default:
                break;
           }
           break;
        default:
           break;
    }
    return e;
}
```

It simply calls the appropriate function according to the service required (given in R2 on entry to the SWI handler). The add and mouse click functions can be found in toolbutton.c:

```
static int my_icons[2] = {0,-1};

typedef struct {
  int event;
  ComponentId cid;
  char state;
  char type;
} PrivateTButton;

extern _kernel_oserror *
toolbutton_add(ToolButton *template,int wimpw,int **icons,int **data)
{
   WimpCreateIconBlock i;
   PrivateTButton *tb;

   tb = (PrivateTButton *) mem_allocate(sizeof(PrivateTButton)+
            string_length(template->sprites)+2);
   if (!tb) return out_of_memory();
   *data = (int *) tb;

   tb->event = template->event ? template->event : ToolButton_StateChanged;
   tb->state = 0;
   tb->cid   = template->hdr.component_id;
   tb->type  = template->type;

   i.window_handle = wimpw;
   i.icon.bbox = template->hdr.box;
   i.icon.flags = WimpIcon_Text+WimpIcon_Sprite+WimpIcon_Indirected+
                                    (WimpIcon_ButtonType * ButtonType_Click);
   i.icon.data.ist.buffer = "";
   i.icon.data.ist.validation = (char *) (1+tb);
   *i.icon.data.ist.validation = 's';
   string_copy (1+i.icon.data.ist.validation,template->sprites);

   my_icons[0] = glib_create_icon(&i);
   *icons = my_icons;
   return NULL;
}
```

The add handler is passed the template for the Gadget to be created and the wimp window handle (note that the SWI handler in main.c is also given the ObjectId for the window, but this is not generally required by Gadgets which are implemented using icons). From this information it is possible to create the Gadget - A data structure is allocated and filled in (Note that the memory is allocated in one chunk- this improves performance and reduces system memory fragmentation. It also means that a default remove handler may be used) and an icon is created. By using the GLib function to create an icon, initial fading and plotting of the Gadget (for !ResEd) will be handled on your behalf.

The add handler should fill in a static (as it is passed back) array with the list of icons that it has created terminated by -1. This allows the Window module to associate wimp events on an icon with a particular Gadget. The handler also returns a pointer to its data structure. This address is then passed to the other handlers which means that they do not have to worry about the more abstract Object/Component Ids and how these are associated with the Gadget's attributes.

Note the use of the string_length and string_copy functions (provided by the String32 library). These are equivalent to the strlen/strcpy functions in the standard C library but can cope with Ctrl terminated strings as the source. It is important that when a Gadget type is made available that it should be usable from any language such as BASIC in common with the rest of the Toolbox. It is conventional to store strings as Null terminated once inside the Toolbox, so that the String32 functions only need to be called for strings in Gadget templates and the method SWIs.

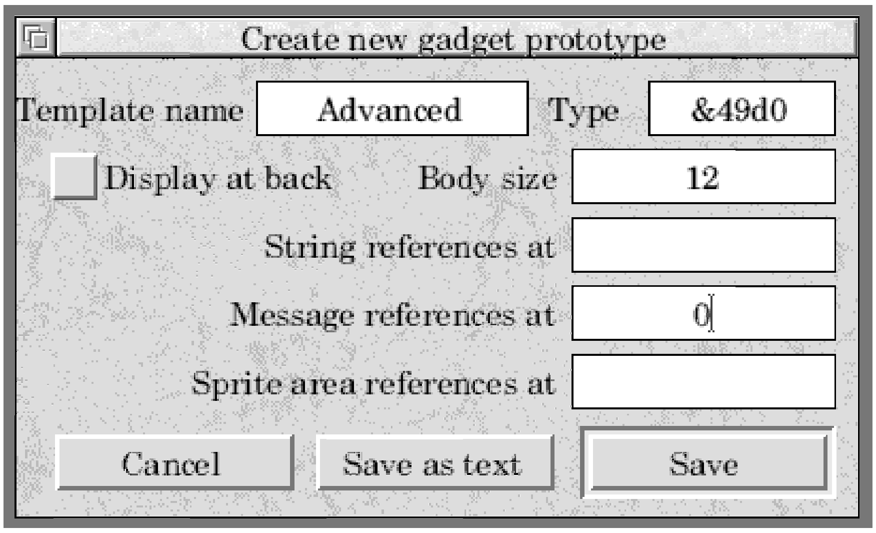
```
extern _kernel_oserror *
toolbutton_mclick(int *handle,WimpMouseClickEvent *click,ObjectId win,int *claim)
{
   WimpSetIconStateBlock state;
   ToolButtonStateChangedEvent event;
   PrivateTButton *tb;

   tb = (PrivateTButton *) handle;
   tb->state = 1- tb->state;

   state.window_handle = click->window_handle;
   state.icon_handle   = click->icon_handle;
   state.clear_word    = WimpIcon_Selected;
   state.EOR_word      = WimpIcon_Selected*tb->state;

   wimp_set_icon_state(&state);

   event.hdr.flags      = 0;
   event.hdr.event_code = tb->event;
   event.hdr.size       = sizeof(ToolButtonStateChangedEvent);

   event.new_state      = tb->state;

   toolbox_raise_toolbox_event(0,win,tb->cid,(ToolboxEvent *) &event);

   *claim = 1;
   return NULL;
}
```

When the wimp sends a click event to the task, the Window module uses the icon lists (returned from Gadget creation) to determine which gadget the click occurred on. For one of the Toolbutton Gadgets this will result in the GTest module being SWIed. The click handler uses the private data (that was allocated in the add handler) to determine the state of the button and toggle it. It then raises a Toolbox event to inform the Client task of the change. Note that the ObjectId is passed through to the handler, but the ComponentId needs to be remembered in the data structure.

By 'claiming' the click, the Window module marks the Wimp mouse click event as having take place on the given component. Generally though, the client will only be interested in the high level event.

Creating and editing new Gadget types

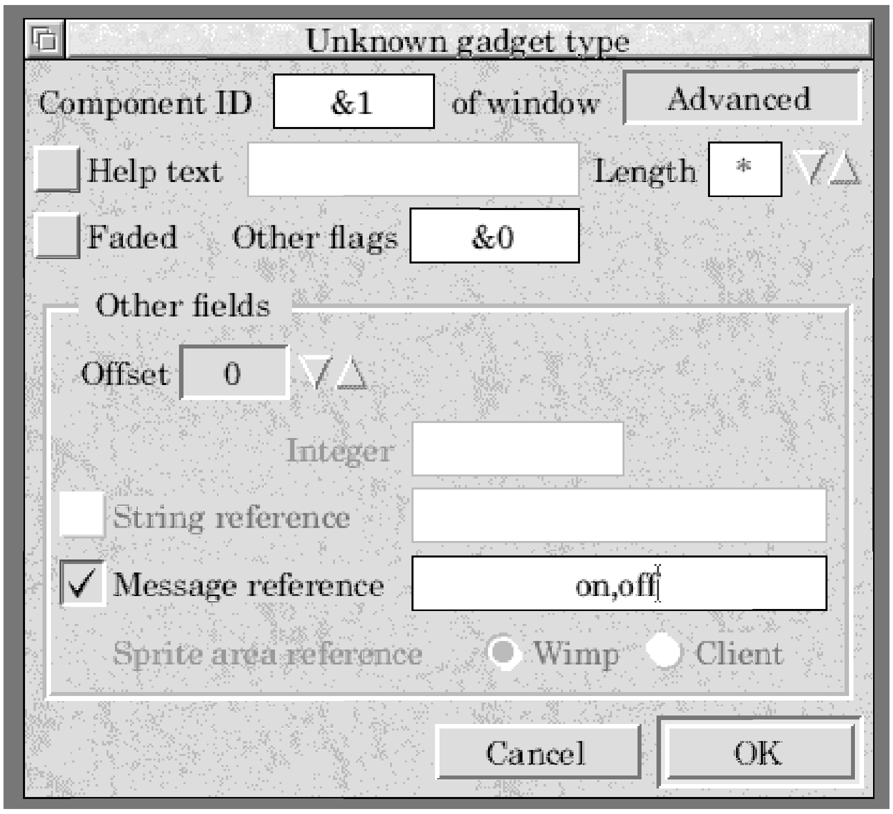
!ResEd comes with an application !ResCreate which as its name implies creates resource files. It will create either a single object or a single gadget (contained in a Window object). When the 'Create gadget' option is chosen from the menu, the following dialogue appears:-


Create new gadget prototype

Template name: Advanced
Type: &49d0
Display at back
Body size: 12
String references at:
Message references at: 0
Sprite area references at:

The 'Template name' is the the name of the Window object template to be created and the type is the allocated Gadget type (preceded by a '&' if entered in hex). The body size is the size of the gadget template, not including the header, so for the simple example above this would be 12. Comma separated lists of offsets should then be entered into the 'reference fields' - for the simple gadget '0' would be entered into the message reference writable (See the User interface toolbox manual for a description of references within templates). The template can then be saved and edited by !ResEd. Note that the Gadget will be created 'empty' so that strings and messages appear NULL and integer fields are zero.

Double Clicking on the 'unknown' Gadget results in a dialogue like the one below:-



The various attributes of the Gadget are modified by using the offset number range to tell !ResEd what attribute is being edited and then !ResEd unfades the relevant writable below. In the Simple or Advanced example this means for offset zero that the message reference is unfaded. Moving the offset to four or eight unfades the integer field.

Modifying !ResTest

Now that we have a new Gadget type, we'll want to try it out. After having created a Toolbutton Gadget with the modified !ResEd it can be loaded into !ResTest. Creating and showing its parent window will display a window containing a Toolbutton. Clicking on this will change its state from one sprite to another. Opening the event log though will show that an unknown event is being raised. !ResTest can be modified (without rebuilding) so that unknown events can become known and as with the standard Toolbox events, like Window_AboutToBeShown, Certain aspects of the event itself can also be displayed.

Two files control !ResTest's output: The 'Standard' file has tags for all the event codes, so we add the line:

```
m00049D00:ToolButton_StateChanged
```

In general this is just <mCCCCCCCC:string> and on encountering the given event code, !ResTest will print (if enabled) the string which is associated with that code via the file. The other useful file is TBlockMess, this allows the event block itself to be printed. To this we add the line:

```
E49d00 "new state = %d" 4 2
```

The '4 2' at the end means print the fourth word as an integer - looking at the ToolButton.h file will show that the new state is stored in the first word after the event block header (which is four words long). Any number of lines may be associated with each event code, eg. Draggable_DragEnded has four lines enabling the destination window, icon and coordinates to be printed. The syntax for each line is shown below:

Eccccc <format string> offset type [extra]	where ccccc is the event code <format string> is a printf like format string, but Containing at most one % operator - see below.

offset is the offset from the start of the event block in either bytes or words depending on the type and [extra] is an optional parameter again depending on the type:-

type	offset in	extra	% operators	effect
1	words		d x s c	address of
2	words		d x s c	contents of
3	words	divisor	f	contents of / divisor
4	bytes		d x s c	contents of
5	bytes	divisor	f	contents of / divisor

For example to print a string which starts at the fourth word of the event block we would use:

```
Eccccc "The string is %s" 4 1
```

The equivalent C for such an event would be like this:-

```
typedef struct {
   ToolboxEventHeader hdr;
   char               string[64];
} myevent;
```

To print a coordinate which starts at the fourth word of the event block we would use:

```
Eccccc "Coordinate: x = %d" 4 2
Eccccc "            y = %d" 5 2
```

The equivalent C for such an event would be like this:-

```
typedef struct {
   ToolboxEventHeader hdr;
   int                x;
   int                y;
} myevent;
```

And to print a value from 0-255 as a percentage which starts at the fourth word of the event block we would use:

```
Eccccc "The value is %2.2f %%" 16 5 2.55
```

The equivalent C for such an event would be like this:-

```
typedef struct {
   ToolboxEventHeader hdr;
   unsigned char      value;
} myevent;
```

More advanced Gadget features

Once a Gadget has been created it is typical that an application will want to operate on it. In the Toolbox environment this is done via a *Toolbox Method.* Adding methods to a Gadget is fairly similar to providing the handlers for adding and clicking that were introduced in the simple Gadget example GTest. Again the relevant bits of the Feature mask (bits.method) need to be set to PRIVATE_HANDLER. The Window module will now SWI the module when a task tries to use a method on the Gadget. As an example GTest can be modified so that it has a method to change the sprites of a Toolbutton. The Appendix on GLib gives details on how the handler will be called. Handling fading is done in a similar way.

Some Gadgets feature attached objects, for example the ActionButton can have a 'Show Object' attached to it. This is done by putting the name of the attached object (ie. the name of the template from which is should be created) inside the Gadget template and then the Gadget add handler will create this object (using the GLib function create_object) and store its handle in the data structure. The Gadget is then free to show and hide the object. Note that when showing, the Gadget should be given as the parent component. Also when the Gadget is removed, it should delete any attached objects according to the recurse bit (see GLib appendix) with Toolbox_DeleteObject. This means that the default remove handler can not be used in this case.

If using glib_create_icon and mem_allocate, then the Window module will handle the plotting of Gadgets (for !ResEd) automatically. It is possible to specify a custom plot handler - this might be desirable if the Gadget is intricate and takes a long time to redraw. Again it is implemented by set the features mask bits and providing a suitable handler - the GLib appendix describes the call mechanism. It's worth noting that providing a plot handler will improve !ResEd's performance. A plot handler should check the faded bit of the Gadget template's flags and modify the plot accordingly - ordinarily, glib_create_icon will have taken care of this.

The plot handler is passed a template from which it should render the Gadget. Note that this template is the same as the one stored in the resource file that is used for creation. One important feature therefore is that the bounding box in the header is in work area coordinates. This will not be a problem if the Gadget's plot handler uses Wimp_PlotIcon to render a template, but for screen relative renderers, access to the window state is required. Although this information is private to !ResEd, a service call is raised as !ResEd enters its redraw loop to allow such renderers access to this information.

Gadgets which contain a writable element can have the focus set to them. For such a Gadget to belong to set of linked writables, it must have a set focus handler which should put the caret in the icon or Gadget which is writable. If the Gadget is faded, it should 'pass on' the focus to the next member of the list. The Glib appendix describes the calling parameters for this handler. It is also possible to move a Gadget after creation - there is a default handler for Gadgets that consist solely of icons, but for Gadgets that can be resized as well as moved or composite Gadgets (see below) a private handler should be provided if the Gadget_Move method is to be supported.

Some Gadgets will require other wimp events such as key presses, drag ended or even wimp messages to function. In these cases the extension module will need to register with the toolbox directly 'to express an interest'. Application note 280 describes the use of these SWIs as they are fundamental to writing Toolbox object modules. This note should be read as there are many subtle aspects to Toolbox filters and failure to comply can lead to unexpected behaviour and in particular poor performance. Composite Gadgets also require them and the example demonstrates this.

Composite Gadgets

This section describes composite Gadgets - these are ones that are made up of other Gadgets, for example the StringSet and NumberRange Gadgets contain no icons and are implemented with other Gadgets. This results in less code duplication as the NumberRange for instance does not require a copy of the slider code. It is even possible to create Gadgets which consist both of icons and other Gadgets!

Writing Composite Gadgets is very similar to icon based ones except that Gadgets are created instead of icons and Toolbox events are used instead of mouse clicks. It is this latter aspect which adds to their complexity. In the same way as keypresses above, Toolbox filters are required to 'hear' the events on the underlying gadgets. GLib provides a simple interface to the Toolbox filters that should be sufficient for most Gadgets.

Redrawable Gadgets

The Gadgets that have been discussed up to now consist of Icons or Gadgets that have been created. In other words the graphic or text that they depict are handled by the wimp. In some instances it is necessary to have a more complex graphic which can not be represented as text or a sprite and so requires an external source to redraw it. This can be achieved by catching wimp redraw requests in a PostFilter and entering a redraw loop. There are a number of aspects though which need considering with this approach:-

- A Window containing such a Gadget will need auto-redraw switched off.
- The filter should (in wimp terms) claim the event as the client will not be expecting the redraw request.
- If the Client did want redraw requests on the window, ie. the window contains one of these gadgets and the client renders into the window, then this approach can not be used. In these cases the client would have to call the Gadget module from within its own redraw loop.

Type Registration

Since Gadgets have a type, a method base and possibly an event code, up to three distinct numbers (or ranges of numbers) must be associated with each type. They must be distinct as there is no guarantee as to what combination of Gadgets will be available in any system. For the purposes of development the following ranges should be used:-

GadgetType	0xssssffxx	where ssss is the size of the template and xx can be 00-ff
MethodBase	0x00ffxx00	
EventBase	0x000ffxx0	

Gadgets which are released outside an organisation MUST be registered with Acorn and in return official types and bases will be allocated. Composite Gadgets require a tag, but as these do not need to be distinct, they do not require registration. Acorn will however maintain a list.

GLib - The Gadget extensions Library

This section describes the GLib function calls that are available to Gadget extension modules, It also describes the underlying SWIs and service calls. Note SBZ is an abbreviation for 'Should be zero'.

Service calls

The Window module provides a number of service calls for various events. Generally, it is just the starting and dying calls that will be of interest.

Service_WindowModuleStarting 0x82881

R1 = 0x82881

```
#define Service_WindowModuleStarting    (Window_SWIChunkBase + 1)
```

This service call is sent by the Window module when it starts up *and is ready to receive SWI calls*. On receipt of this, a Gadget extension module should register itself with the Window Module.

Service_WindowModuleDying 0x82882

R1 = 0x82882

```
#define Service_WindowModuleDying       (Window_SWIChunkBase + 2)
```

This service call is sent by the Window module when it is finalising. On receipt of this call a module may need to clear up and free any memory that has been allocated, though of course it need not die itself. Note that any memory allocated with mem_allocate will already have been freed **and should not be freed again**.

Service_GadgetRegistered 0x82883

R0 = type
R1 = 0x82883
R2 = SWI number of handler
R3 = feature mask

```
#define Service_GadgetRegistered        (Window_SWIChunkBase + 3)
```

This service call is sent by the Window module whenever a Gadget extension module registers itself. R0 identifies what type is being registered with R2 and R3 giving details of how such Gadgets will be handled.. It might be useful to a module which relies on the existence of a particular Gadget type.

Service_GadgetDeregistered 0x82884

R0 = type
R1 = 0x82884
R2 = SWI number of handler

```
#define Service_GadgetDeregistered      (Window_SWIChunkBase + 4)
```

This service call is sent by the Window module when a Gadget type has successfully been deregistered. It might be useful to a module which relies on the existence of a particular Gadget type. It is possible that an extension module may register/deregister several times, so a reference count should be used to determine if any handler is present.

Service_RedrawingWindow 0x44EC6

R0 = window handle
R1 = 0x44EC6

```
#define Service_RedrawingWindow      (Toolbox_SWIChunkBase + 6)
```

This service call is sent (in particular) by !ResEd so that Gadget plot handlers can gain access to information about the window that !ResEd is currently updating.

Registration

When registering for a particular Gadget type a mask of features (with all reserved/unused bits zero) must be supplied:-

```
typedef struct {
   int add:2,            can be PRIVATE_HANDLER or NO_HANDLER¹
   remove:2,             can be PRIVATE_HANDLER, DEFAULT_HANDLER² or NO_HANDLER³
   reserved0:2,          SBZ
   method:2,             can be PRIVATE_HANDLER or NO_HANDLER
   reserved:2,           SBZ
   mclick:2,             can be PRIVATE_HANDLER or NO_HANDLER
   reserved2:4,          SBZ
   plot:2,               can be PRIVATE_HANDLER or DEFAULT_HANDLER
   setfocus:2,           can be PRIVATE_HANDLER or NO_HANDLER
   move:2,               can be PRIVATE_HANDLER, DEFAULT_HANDLER or NO_HANDLER
   fade:2;               can be PRIVATE_HANDLER or NO_HANDLER
} FT;

                         ¹Not useful
                         ²Only if GLib mem_allocate and create_xxx used.
                         ³Not a good idea as no clear up will happen

typedef union {
   int mask;
   FT  bits;
} FeatureMask;
```

SWI Window_RegisterExternal 0x82885

Entry:	R0 =	flags (SBZ)
	R1 ->	types (see below)
	R2 =	Gadget SWI number

types is an array of GadgetExtensionRecords, ie.:-

offset	value
+0	type
+4	valid flags
+8	features mask
+12n	-1 (terminator)

or in C terms:-

```
typedef struct {
  int type;
  int validflags;
  FeatureMask features;
} GadgetExtensionRecord;
```

Exit: All registers preserved

This SWI registers the new Gadget type(s) with the Window module. The given SWI number will be called whenever the Window module encounters the given Gadget type and the features mask indicates that there is a handler available. See the registration section on valid types.

Two C functions are provided for registration, one taking an array like the SWI and the other taking separate parameters enabling registration for a single type:-

```
extern _kernel_oserror *register_gadget_types
                                     (unsigned int flags,
                                      GadgetExtensionRecord *rec,
                                      int SWIno);

extern _kernel_oserror *register_gadget_type
                                     (unsigned int flags,
                                      int type,
                                      unsigned int valid,
                                      unsigned int mask,
                                      int SWIno);
```

Deregistration

SWI Window_DeregisterExternal 0x82886

Entry:	R0 =	flags (SBZ)
	R1 =	type of Gadget to deregister
	R2 =	SWI that handled it.

Exit: All registers preserved

This SWI deregisters a Gadget type. Note that if a module dies without deregistering, the Window module will automatically deregister it on the next SWI call. This behaviour should not be relied on though - it is there to prevent unexpected behaviour with the rest of the system. A C function is provided by GLib:-

```
extern _kernel_oserror *deregister_gadget_type
                                     (unsigned int flags,
                                      int type,
                                      int SWIno);
```

Module support

SWI Window_SupportExternal 0x82887

Entry:	R0 =	flags (SBZ unless otherwise stated)
	R1 =	Reason code
	R2...	Code specific
Exit:	depends on reason code	

This SWI provides various function calls that are of use to extension modules.

CreateIcon (0)

Entry:	R1 =	0 (CreateIcon)
	R2 ->	wimp icon definition
Exit:	R0 =	Icon handle or -1 if failed to create

This call creates a wimp icon from the given definition except when the Window module is in 'plotting' mode. A C equivalent is available:

```
int glib_create_icon(WimpCreateIconBlock *i);
```

CreateObject (2)

Entry:	R0 =	flags	
	R1 =	2 (CreateObject)	
	R2 ->	depends on flag bit 0:	
		clear	R2-> template name
		set	R2-> in memory object definition
Exit:	R0 =	ObjectId or zero if failed to create	

This call creates a toolbox object and returns its ObjectId. The reason for using this as opposed to toolbox_create_object (SWI Toolbox_CreateObject) is that the Window module first checks for plotting rather than creation mode. If the Gadget provides a plot handler, then it may safely use Toolbox_CreateObject SWI in its add handler.

CreateGadget (3)

Entry:	R1 =	3 (CreateGadget)
	R2 =	ObjectId (must be current ObjectId)

	R3 ->	gadget definition
	R4 =	tag (0x800 to 0xfff)
Exit:	R0 =	ComponentId or 0 if failed to create

This call, used for implementing composite gadgets, creates a gadget from the given template definition. The Window module generates a unique ComponentId for the gadget derived from the tag - It will be of the form 0x?????ttt. The use of a tag allows improved reaction to events on composite gadgets in that if a Composite Gadget creates all its Gadgets with tag 999, then its event routines can return immediately if the ComponentId's lower bits do not equal 999. Tags do not require registration, but it is worth advising Acorn so that where possible, clashes can be avoided. In order to receive Toolbox events, a module must register an interest in them - this is described in application note 280. A C equivalent is available:

```
int create_gadget(int o,int *i,int f);
```

MemAllocate (4)

Entry:	R1 =	4 (MemAllocate)
	R2 =	amount
Exit:	R0 =	pointer to memory or NULL

This call allocates memory from the Window modules allocator. A C equivalent is available:

```
void *mem_allocate(int amount);
```

MemFree (5)

Entry:	R1 =	5 (MemFree)
	R2 =	pointer
Exit:	All registers preserved	

This call frees memory that was previously allocated by the Window modules allocator. A C equivalent is available:

```
void mem_free(void *tag);
```

MemExtend (6)

Entry:	R1 =	6 (MemExtend)
	R2 =	pointer
	R3 =	change in size
Exit:	R0 =	pointer to memory of NULL if extend failed

This call reallocates memory that was previously allocated by the Window modules allocator. The change may be positive or negative to grow or shrink the block accordingly. Note that the pointer to the block may have to change and this is returned in R0.

C functions

GLib provides a number of functions that are useful for writing Gadgets, which are not provided by the Support SWI.

```
void graphics_window (BBox *area)
```

This call sets the current graphics (ie clipping window) to the given bounding box by calling VDU 23. It is of use by Redrawable Gadget handlers and possibly plot handlers.

```
BBox *intersection(BBox *one, BBox *two)
```

This call returns the bounding box that is the intersection between the two given boxes. Note that as this call returns the pointer to a static area, its contents will not persist over consecutive calls. If there is no intersection, then the returned pointer will be zero.

C interface to filters

For filters an event list is a pair array of event and class terminated by a -1. For no class, eg. wimp messages or wimp events a class of zero should be used. For example an event list for a toolbox event on a Window object might be {1 /* client event code */, Window_ObjectClass, -1} or for redraw events { Wimp_ERedrawWindow, 0 /* wimp window */, -1}.

```
_kernel_oserror *add_task_interest(FilterTypes type, int *events, int SWI)
```

This function calls (if necessary) Toolbox_RegisterPostFilter (described in detail in application note 280) with the given list of events for the current task. The type can be one of GLib_ToolboxEvents (typically used for composite Gadgets), GLib_WimpEvents or GLib_WimpMessages. The SWI to call when matching this event pattern should also be given. The call should be made in the Gadget add handler and then GLib decides whether or not to make the call to the Toolbox according to a reference count - This ensures an efficient use of filters as registering multiple times can affect performance.

```
_kernel_oserror *remove_task_interest(FilterTypes type, int *events)
```

This function decrements its reference count and then deregisters the Toolbox filter if necessary. It should be called un the Gadget remove handler - this obviously means that a default handler cannot be used for Gadgets that have registered an interest in events.

Note that only one list may be used per module per type of filter. This means that where more than one Gadget is implemented by a module and each Gadget requires toolbox events the relevant lists require combination. This is however a restriction of GLib and not the Toolbox.

When such a filter is called it will have the following parameters passed to it:-

Entry:	R0 =	event code
	R1 ->	event block
	R2	
	R3 ->	Id block for event
Exit:	R0 =	Claim state

The event code will be one of the Wimp event codes or 512 (Toolbox Event). The filter should process the event and set R0 as follows. If the filter was uninterested in the event or it has just used the information in the event (as a Composite Gadget might do and then raise a 'higher level' event) then R0 should be set to zero. If the Filter has updated the Id block (eg. set the Component Id) then +1 should be returned in R0, but note that for Gadgets this is a very unlikely use as it would generally break the philosophy of the toolbox. Finally setting R0 to -1 will claim the event (in wimp terms) so that it will not be seen by the client. This could be acceptable in the case of a keyboard shortcut for instance, but again generally events should be allowed through.

Other C functions

Other functions within wimplib and toolboxlib are also useful, in particular wimp_delete_icon, wimp_create_icon (when glib_create_icon not used), wimp_plot_icon (for plot handlers - see below), toolbox_delete_object, toolbox_raise_toolbox_event etc. Note the Window_AddGadget method **must not be used** when writing composite Gadgets as the parent Window object may not have finished being created.

Handlers

When the Gadget Module's SWI handler is called, R0 holds the flags (which are reason code specific), R1 the Gadget type and the reason code (in R2). The reason codes and register usage are described below. Note that if a handler needs to call another handler (eg. the mouse click handler calls the method handler), then this should be done directly inside the extension module and not through the Toolbox SWIs. It is however acceptable to make Toolbox SWIs for Gadgets of a different type.

GADGET_ADD (1)

Entry:	R2 =	GADGET_ADD (1)
	R3 ->	gadget template
	R4 =	ObjectId of parent window
	R5 =	Wimp window handle
Exit:	R0 =	Handle to use in future calls
	R1 ->	icon list

On receiving this reason code an extension module should create the new gadget from the given data and return its icon list and private handle. The template will already have been checked for valid flags. Any attached objects (like the Click show objects attached to action buttons) should be created using the GLib calls.

GADGET_REMOVE (2)

Entry:	R0 =	flags (bit 0 is recurse bit)
	R2 =	GADGET_REMOVE (2)
	R3 ->	private data
Exit:	All registers preserved	

On receiving this reason code an extension module should remove the gadget. If the recurse bit is set then any attached objects such as Windows or Menus should not be removed. This is in common with the flag bit in the SWI Toolbox_DeleteObject.

GADGET_FADE (3)

Entry:	R2 =	GADGET_FADE (3)
	R3 ->	private data
	R4 =	state (non zero = fade)
	R5 =	ObjectId of window to which this Gadget belongs
Exit:	All registers preserved	

On receiving this reason code an extension module should fade or unfade the gadget, according to the value in R4. A Gadget will have been faded by the client making a call to the Gadget_SetFlags method.

GADGET_METHOD (4)

Entry:	R2 =	GADGET_METHOD (4)
	R3 ->	private data
	R4 ->	registers passed to Toolbox_ObjectMiscOp SWI
Exit:	Depends on Method	

The window module raises this reason code when a task uses the Toolbox_ObjectMiscOp SWI on a Gadget which is provided by an extension module. R4 points to a register bank (in C terms this is the _kernel_swi_regs type) which contains the values that the client passed to the SWI. If any information is to be returned to the client then this bank should be updated. Ie. to return a value in R0, do not set R0 to the value but update the contents of the location pointed to by R4+0. Note that the generic Gadget methods such as Gadget_SetFlags do not result in a call to GADGET_METHOD.

GADGET_MCLICK (6)

Entry:	R2 =	GADGET_MCLICK (6)
	R3 ->	private data
	R4 ->	wimp event block
Exit:	R1 = claim	

The window module raises this reason code when an icon belonging to the Gadget is clicked on. The event block in R4 is just like the one returned by Wimp_Poll when an icon is clicked on. An extension module would typically use this to update the state of the Gadget or return a 'high-level' Toolbox Event. By setting R1 to be non-zero the Window module will update the tasks ID block so that when the wimp mouse click event is delivered it is marked as belonging to this Gadget. Although it is unlikely that the task will be interested in the underlying event it should still be claimed. Possible exceptions to this rule are when the menu mouse button is used - in this case, not claiming the click (R1 = 0) passes the mouse click on which would allow the Window's menu to be viewed. This is not true for a popup menu type Gadget though.

It is also possible to claim the event entirely so that the task will not see the mouse click. This is achieved by setting R1 to -1 on exit, but this should be avoided as this does not fit the Toolbox model and so should only be done in specialist cases.

GADGET_PLOT (9)

Entry:	R2 =	GADGET_PLOT (9)
	R3 ->	gadget template definition

Exit:	All registers preserved

The window module raises this reason code when !ResEd is plotting Gadgets. The extension module should use Wimp_PlotIcon or (Window_PlotGadget for composite Gadgets) to display the Gadget which is defined by the template.

GADGET_SETFOCUS (10)

Entry:	R0 =	flags, bit 0 = direction
	R2 =	GADGET_SETFOCUS (10)
	R3 ->	private data

Exit:	All registers preserved

When the Window module wishes to set the focus to a Gadget, this call will be made. The direction signals whether to set the focus to the before of after Gadget in the event of this Gadget being faded. Care should be taken when 'passing on' the focus as an infinite loop will be entered if all members of the list are faded. Gadgets which are composite and consist of a writable Gadget do not need to worry about this and should just set focus to their writable without checking their faded status, assuming that the before and after fields of the underlying writable have been set.

GADGET_MOVE (11)

Entry:	R2 =	GADGET_MOVE (11)
	R3 ->	private data
	R4 =	wimp window handle
	R5 ->	new bounding box
	R6 =	window ObjectId

Exit:	All registers preserved

This call is made when the Gadget_Move method is applied to a Gadget. A composite gadget must supply a handler if it is going to be moved after creation as the default handler can only move the icons which belong directly to a Gadget. It is also of use if the Gadget supports resizing - the default mover will only move by an offset as it generally does not know how to handle and change in the size of the bounding box.

GADGET_POSTADD (12)

This call is beyond the scope of this document.

GADGET_???? (?)

Over time more handlers may become available and so the SWI handler should not assume that only the handlers it has requested will result in a call out. For a handler in C this would result in nothing being done by the default case.

Miscellaneous Information

Hints & tips

Some thought should go into how different Gadgets are grouped together into modules. On the one hand adding more and more modules can affect system performance due to an increase in the number of SWI handlers and service call handlers. However, memory will be wasted by combining a popular Gadget with a set of Gadgets that are hardly used or are very specific to a particular application.

It is also worth considering whether a new Gadget is indeed a new Gadget or just a variant of an existing one - This is particularly important when it comes to maintainability of an interface as a Gadget cannot be converted into a different type, but it can have its properties modified. An example of this distinction could be the Labelled box Gadget. If this were two Gadget types (one for sprite labels and one for text labels) then it would be harder to maintain if the designer changed their mind as to whether they wanted a sprite or text.

Limitations

Early versions of the Window module have some faults which may affect the behaviour of the extension system:-

Fault	Comment	Fixed in
GADGET_POSTADD calls raised	Can be safely ignored	1.31
Default GADGET_MOVE will not work	Need to provide PH and call glib_move_gadget	1.31
Invalid flags can cause data abort	Either set valid flags to -1 and check in add handler or ignore - for a valid gadget this will not happen.	1.31

Examples

A number of examples are supplied with this application note. They are not intended to be used as complete Gadgets in the same way as those supplied with the Toolbox and Acorn reserves the right not to support them or even reuse the types. No guarantee is made as to how robust or complete they are and Acorn can not be held responsible if code within the examples is copied and breaks.

The Simple Gadget is a 'Tool Button' and this is developed into the Advanced Gadget which adds methods.

The Composite Gadget implements a 'Radio Group' - a labelled box with radio buttons inside.

The Combined example shows how both Gadget types are implemented as one module.