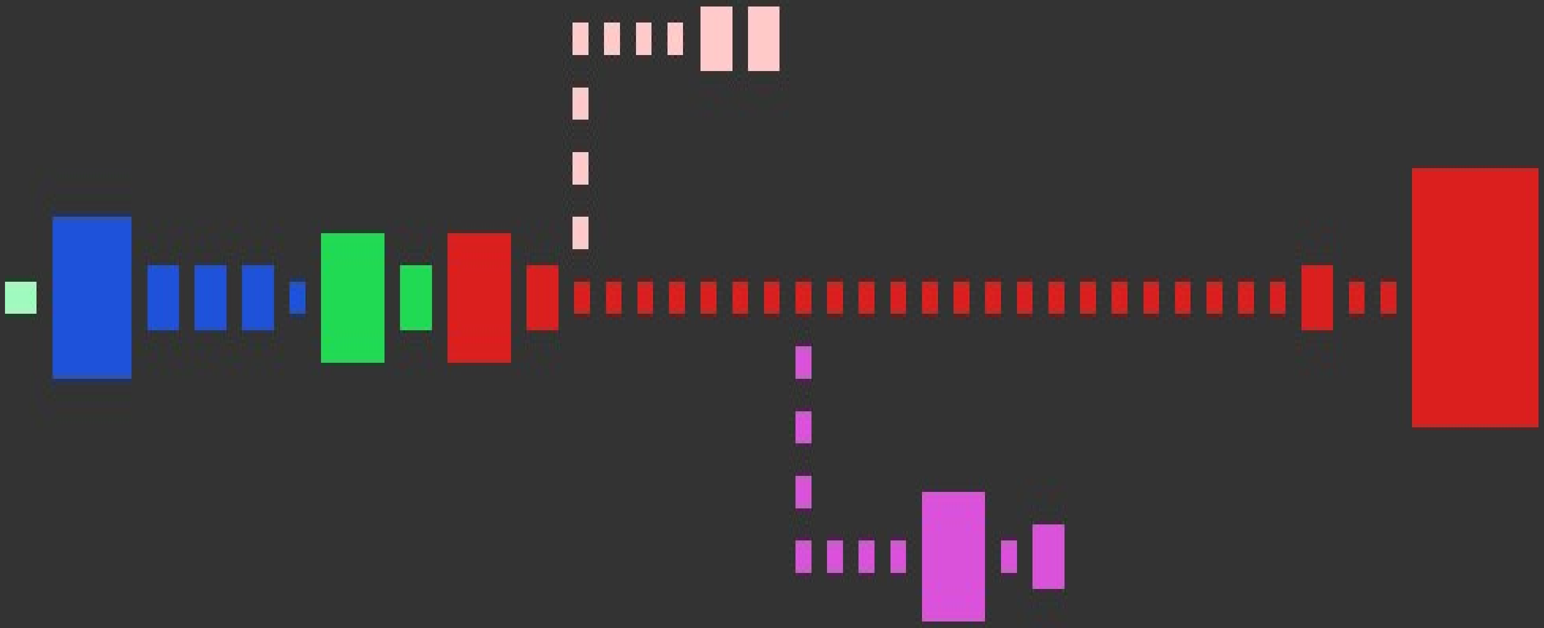
ROM Kernel Reference Manual

CHANGES & ADDITIONS



THE COVETED TECHNICAL REFERENCE ADDENDUM

Nothing that's really good ever gets really lost!

COVERS *AMIGA*OS3.2

ROM Kernel Reference Manual

CHANGES & ADDITIONS

Work in progress   Work in progress   Work in progress   Work in progress
1st January 2024

Covers AmigaOS 3.2

Dedicated to Robert A. Peck
The author of the original ROM Kernel Reference Manuals, who set a very high bar for how computer documentation should be written.

Preface

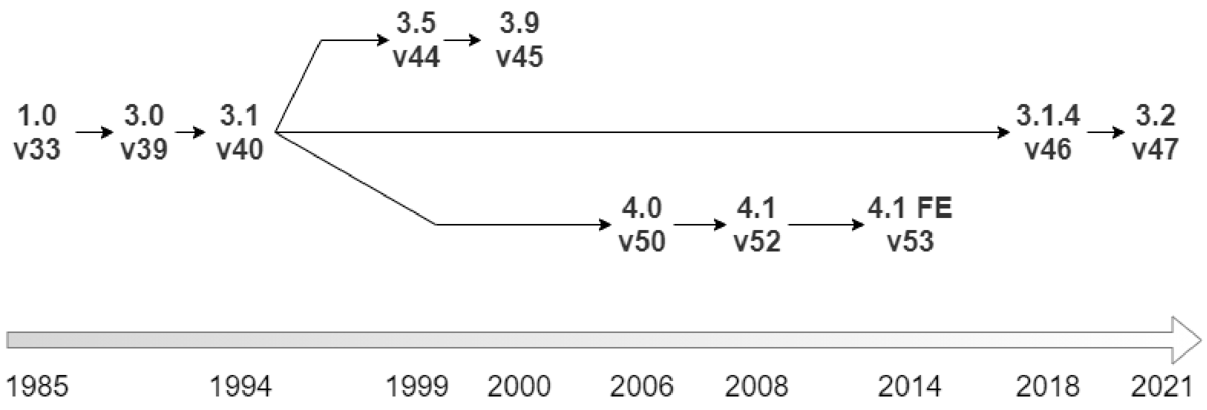
This book *ROM Kernel Reference Manual: Changes & Additions* aims to cover all programming topics of the Amiga Operating System since Release 2 all the way to 3.2. Some of the features are 30 years old, and some things were added within the last few years. That is a huge span, and a huge undertaking to write about all of it.

This book is initially published as a PDF, as it is more important to get the information out quickly than for it to be complete. Over time it will be improved, and more information will be added. Thus initially you may find several topics lacking, but please keep an eye out for updates.

This book is meant to help you expand your knowledge of programming the Amiga. It assumes you have previous experience with programming the Amiga, and it all but assumes you have all the books in the Amiga ROM Kernel Reference Manual series. Like the books in that series a knowledge of the C programming language is not required, but will make it much easier to understand the material in this book.

The history of AmigaOS 3.2

Since release 3.1 the AmigaOS has seen many different directions of developments.



AmigaOS 3.2 builds upon 3.1.4 but has inherited numerous features and improvements from the other development forks. Therefore, understanding the lineage of AmigaOS 3.2 will enable developers to take full advantage of the numerous features that were incorporated into it.

After the demise of Commodore in 1994, Haage & Partner was the first company granted a licence to update AmigaOS. They released AmigaOS 3.5 in 1999, and 3.9 in 2000. The main improvements were the inclusion of the ReAction toolkit and many updated or new utility programs.

In 2006, Hyperion Entertainment released AmigaOS 4.0 which was directly evolved from the original AmigaOS but redesigned to become fully PowerPC compatible. The developers faced numerous challenges since it was difficult to make the AmigaOS independent of the original chipsets. In general, new tags and attributes to functions, numerous bug fixes, and enhancements exist throughout AmigaOS 4.0/4.1.

In 2018, Hyperion Entertainment released AmigaOS 3.1.4, which was a new branch from 3.1. The new development team took a different path now targeting the “retro” computing crowd, and incorporated the wisdom and knowledge gained from the 3.9 and 4.1 branches. The version number tries to reflect that it is an enhancement over 3.1 with a focus on bug fixing while also providing several enhancements. The low level bug fixes, enhancements, and the fact it is still usable on a Motorola 68000 CPU made this update a commercial success.

In 2021, Hyperion Entertainment released AmigaOS 3.2. It is a continuation of the 3.1.4 project, but this time with more emphasis on improvements and new features. Versions 3.9 and 4.1 were influential in what to backport or reimplement, but also in what not to do. One of the guiding principles is respect for the Commodore past. Out of the box it can be hard to see the enhancements, but they are right there to make it easier for users and developers alike.

History of the BOOPSI GUI Classes

After Commodore released the 3.1 version of AmigaOS the idea was to provide many new gadgets and a layout system. The GadTools library was a great system for its time, but for more demanding applications there really is a need for windows that can adapt their layout of gadgets to different font sizes and when the user changes the size of the window.

So during the 1993 Developer conference Commodore’s plans for many new gadget classes were presented. Commodore soon after went bankrupt, but the ideas were there and a private initiative to implement those gadgets was born and known as ClassAct and later ReAction.

In AmigaOS 3.2 those BOOPSI classes have been incorporated into the system and are no longer seen as a package named ClassAct nor ReAction. They are simply BOOPSI classes that are part of AmigaOS. And they have seen a lot of improvements over what has been previously released.

Brief Overview of the Contents

Chapter 1, *BOOPSI windows and layouts*. These are the building blocks to create new and dynamic graphical user interfaces

Chapter 2, *Common gadgets*. A walkthrough of the gadgets that are needed in almost every application: button.gadget, checkbox.gadget, radiobutton.gadget, chooser.gadget, scroller.gadget, slider.gadget, string.gadget, and integer.gadget.

Chapter 3, *Listbrowser.gadget*. A detailed look at this very versatile gadget. It supports 3 modes: a tree, a multicolumn list, or a simple list.

Chapter 4, *Clicktab.gadget*. Another important gadget for many modern applications. It allows you to organise other gadgets inside tabs.

Chapter 5, *Texteditor.gadget*. This is almost an entire text editor in a single gadget

Chapter 6, *Speedbar.gadget*. Bla bla

Chapter 7, *Sketchboard.gadget*. Bla. bla

Chapter 8, *Space.gadget and virtual.gadget*. Bla. bla

Chapter 9, *Creating a gadget subclass with modern features*. Since Release 2 many new features have been introduced: Hierarchical layout, contextual help text, contextual mouse pointers, mouse wheel scrolling and visual theme support. This chapter revisits the topic of creating your own BOOPSI gadget, and supporting those features in your own gadget.

Chapter 10, *Screens under Release 3*. Many things have been improved since Release 2.

Chapter 11, *Menus under Release 3*. As implemented under Intuition, GadTools and Window.class.

Chapter 12, *Bitmaps and RTG*. In a world where RTG has become common we take a closer look at what implications this has for working with bitmaps.

Chapter 13, *Datatypes*. Bla bla

Chapter 14, will look at the features found in the Workbench API.

Chapter 1
BOOPSI WINDOWS AND LAYOUTS

The BOOPSI **window.class** is a new class which together with the new **layout.gadget** provides the framework for font sensitive and self adjusting automatic layout of gadgets. It is a huge boost to how the UI is made on the Amiga.

The Window and Layout Classes

The **window.class** object doesn't pretend to be a **Window** struct, unlike gadget class and image class which pose as the good old structs **Gadget** and **Image** Instead the **window.class** represents a window even when the window is not yet opened. The window object can then be asked to actually open the window on screen and this produces an instance of the **Window** structure.

But a **window.class** object is not enough on its own. You will need a main layout in the window. The **layout.gadget** class is as its name suggests a gadget. Layout gadgets are like invisible containers, and they can hold other gadgets within themselves including other layout gadgets.

For now, just understand that the **layout.gadget** can work in either of two directions: Horizontal or vertical. By inserting such layout gadgets into other layout gadgets quite complex layouts can be created.

OPENING A WINDOW THE BOOPSI WAY

The process goes like this: Create a **window.class** object and ask it to open your window giving you a pointer to the **Window** struct. So please be careful when reading and coding that you don’t get the window object and the **Window** struct mixed up.

A window.class object is created like any other BOOPSI object. You call **NewObject**() with a pointer to its class or by giving the public name “window.class”. In either case it’s important that the class exists beforehand with a call to **OpenLibrary**() This is because the class is not loaded into the system until needed. So you also need to free the class resources when you are done by calling **CloseLibrary**(). You get a pointer to the class by calling a function in the library. For the window.class the library function is **WINDOW_GetClass()**.

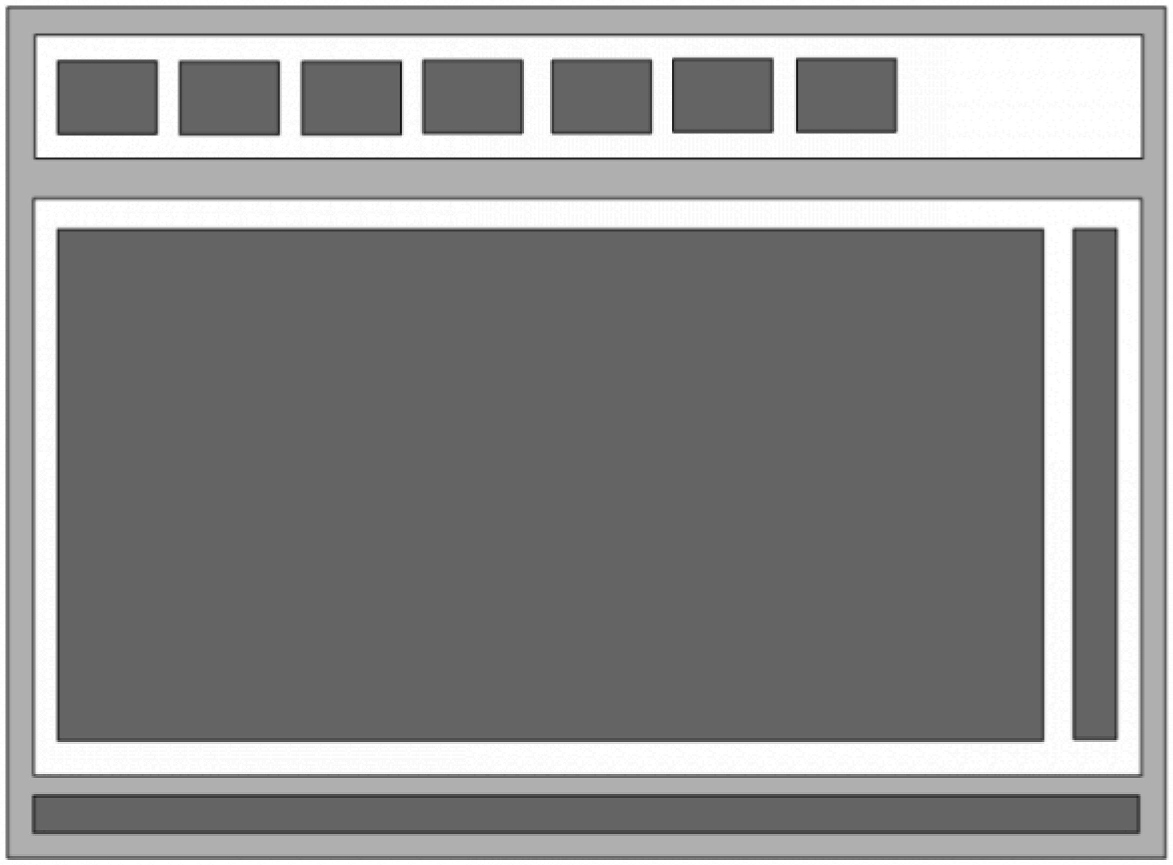
Once you have the window.class object you can ask it to open the window by calling its WM_OPEN method using **DoMethod**() or **DoMethodA**()

Open BOOPSI Window Example

```
#include <stdlib.h>

#include <intuition/classusr.h>
#include <gadgets/layout.h>
#include <classes/window.h>

#define __CLIB_PRAGMA_LIBCALL
#include <proto/alib.h>
#include <proto/exec.h>
#include <proto/layout.h>
#include <proto/window.h>
#include <proto/intuition.h>

char *vers="$VER: BOOPSIdemo 1 (09.12.2019)";

struct IntuitionBase *IntuitionBase;
struct Library *WindowBase;
struct Library *LayoutBase;

void cleanexit(Object *windowObject);
void processEvents(Object *windowObject);

int main(void)
{
   struct Window *intuiwin = NULL;
   Object *windowObject = NULL;
   Object *mainLayout = NULL;

   if (! (IntuitionBase = (struct IntuitionBase*)OpenLibrary("intuition.library",47)))
      cleanexit(NULL);

   if (! (WindowBase = OpenLibrary("window.class", 47)))
      cleanexit(NULL);

   if (! (LayoutBase = OpenLibrary("gadgets/layout.gadget", 47)))
      cleanexit(NULL);

   mainLayout = NewObject(LAYOUT_GetClass(), NULL,
         LAYOUT_Orientation, LAYOUT_ORIENT_VERT,
         LAYOUT_DeferLayout, TRUE,
         LAYOUT_SpaceInner, TRUE,
         LAYOUT_SpaceOuter, TRUE,
         TAG_DONE);

   windowObject = NewObject(WINDOW_GetClass(), NULL,
         WINDOW_Position, WPOS_CENTERSCREEN,
         WA_Activate, TRUE,
         WA_Title, "BOOPSI window demo",
         WA_DragBar, TRUE,
         WA_CloseGadget, TRUE,
         WA_DepthGadget, TRUE,
         WA_SizeGadget, TRUE,
         WA_InnerWidth, 300,
         WA_InnerHeight, 150,
         WA_IDCMP, IDCMP_CLOSEWINDOW,
         WINDOW_Layout, mainLayout,
         TAG_DONE);
   if (! windowObject)
      cleanexit(NULL);

   if (! (intuiwin = (struct Window *) DoMethod(windowObject,WM_OPEN, NULL)))
      cleanexit(windowObject);

   processEvents(windowObject);

   DoMethod(windowObject,WM_CLOSE);
   cleanexit(windowObject);
}

void processEvents(Object *windowObject)
{
   ULONG windowsignal;
   ULONG receivedsignal;
   ULONG result;
   ULONG code;
   BOOL end = FALSE;

   GetAttr(WINDOW_SigMask, windowObject, &windowsignal);

   while (!end)
   {
      receivedsignal = Wait(windowsignal);
      while ((result = DoMethod(windowObject, WM_HANDLEINPUT, &code)) != WMHI_LASTMSG)
      {
         switch (result & WMHI_CLASSMASK)
         {
            case WMHI_CLOSEWINDOW:
               end=TRUE;
               break;
         }
      }
   }
}

void cleanexit(Object *windowObject)
{
   if (windowObject)
      DisposeObject(windowObject);

   CloseLibrary((struct Library*)IntuitionBase);
   CloseLibrary(WindowBase);
   CloseLibrary(LayoutBase);
   exit(0);
}
```

Making Layouts With Many Gadgets

In the above example code the main layout was already created and added to the window. A window only holds a single object of **layout.gadget** class. To have more gadgets in your window you add them to the main layout. Some or all of these children can be layout gadgets themselves. In the following figure an example application is sketched.



The light grey represents the main **layout.gadget** of the window that was already created in the example. It is of vertical orientation, and contains two layout gadgets objects (white) and a **scroller.gadget** object at the bottom depicted in dark grey.

The top white **layout.gadget** can be imagined to hold several buttons all shown as dark grey. The middle white layout.gadget can be imagined to hold some kind of **sketchboard.gadget** and a vertical **scroller.gadget**.

To create the sub layouts you basically do the same as when creating the main layout.

Direction of the Layout, Margins and Spacing

The layout can be either horizontal or vertical, and it must be decided at the time of creation with the LAYOUT_Orientation tag set to either LAYOUT_ORIENT_VERT or LAYOUT_ORIENT_HORIZ.

All around the layout you can have a margin (outer spacing) which you can turn on or off with LAYOUT_SpaceOuter. It allows you to create exactly the layout you want. How many pixels the margin consists of is under user control in preferences. You only give it a value of TRUE or FALSE.

However, you can also control the margin of the top, left, right, bottom individually via tags. The values are given as a number of virtual pixels, which behind the scenes is scaled according to the active theme. It is currently suggested to only use these tags to turn off margins on an edge by setting it to 0. And also be aware that to set them after any of the general tags LAYOUT_SpaceOuter and LAYOUT_SpaceInner, as these will reset the more specific values.

The layout can also have spacing between its children which you control with LAYOUT_SpaceInner in a similar manner. And there is also a similar tag to control the number of virtual pixels but it is not recommended to use it.

General Distribution and Alignment of Children

You can align the children of the layout left, center or right horizontally, and top, center or bottom vertically. This only makes sense if the children don't take up all the space of the layout already.

The LAYOUT_ShrinkWrap tag makes sure there is no surplus space between children. You can think of it as the "justify" alignment you know from word processors. You can also think of it as growing all gadgets so there is no empty space. This is generally a bad user experience, so only use it in special cases.

The LAYOUT_EvenSize tag makes the layout.gadget use some of the surplus space to make all children of equal size. It is only increased up to the largest of the children's minimum sizes. This comes in handy when you have a row of buttons. The style guide suggests that they have the same size, yet they shouldn't take up all of the space.

Adding, Removing or Modifying Children

You can add children with the LAYOUT_AddChild tag and images with LAYOUT_AddImage tag with the value being a pointer to the object you add. They are added at the end. In a similar way you can remove children.

When the layout is deleted it will automatically delete its children. The removed child is also deleted automatically. If you don't want this to happen use the CHILD_NoDispose tag.

After you have added a child you can modify how it behaves inside the layout with one or more CHILD_xxxx tags. These tags must follow after the LAYOUT_AddChild or LAYOUT_AddImage tag.

So you set the CHILD_xxxx tags on the layout.gadget but they modify how one of the children behaves. The subject of these tags is the last child you added, or you can use LAYOUT_ModifyChild or LAYOUT_ModifyImage to set the subject.

Example Snippet

The first example created the mainLayout. In this small piece of code the middleWhiteLayout is created and also shows how to add that into the mainLayout. You should imagine that a lot of the other gadgets have already been created. For the full picture see the complete example at the end of the chapter

```
   middleWhiteLayout = NewObject(LAYOUT_GetClass(), NULL,
         LAYOUT_Orientation, LAYOUT_ORIENT_HORIZ,
         LAYOUT_SpaceInner, TRUE,
         LAYOUT_SpaceOuter, FALSE,
         LAYOUT_AddChild, sketchboardGadget,
         CHILD_WeightedWidth, 100,
         LAYOUT_AddChild, verticalScroller,
         CHILD_WeightedWidth, 0,
         TAG_DONE);

   mainLayout = NewObject(LAYOUT_GetClass(), NULL,
         LAYOUT_Orientation, LAYOUT_ORIENT_VERT,
         LAYOUT_DeferLayout, TRUE,
         LAYOUT_SpaceInner, TRUE,
         LAYOUT_SpaceOuter, TRUE,
         LAYOUT_AddChild, topWhiteLayout,
         CHILD_WeightedHeight, 0,
         LAYOUT_AddChild, middleWhiteLayout,
         CHILD_WeightedHeight, 100,
         LAYOUT_AddChild, horizontalScroller,
         CHILD_WeightedHeight, 0,
         TAG_DONE);
```

FINE CONTROL OF THE DISTRIBUTION OF CHILDREN

Most gadgets in 3.2 are able to tell what their minimum, nominal and maximum sizes are, but you can override with CHILD_MinWidth, CHILD_MinHeight, CHILD_MaxWidth, and CHILD_MaxHeight. If you later want to get back to using the gadget's own reported value, you can use the special value of ~0 with these tags.

As can be seen in the example snippet each child can be assigned a weight. The weight is a number between 0 and 100 (default) and it determines how much space, in relation to the other children, that this child is allowed. When the width/height for the layout is calculated, the allowable space is divided up between the childrens based on this, and the min/max values. For instance, if you had two children, one with a weight of 100 and another with a weight of 50, and the layout was 150 pixels wide, 100 would be offered to the first, and 50 to the second; how much space it actually takes depends on whether that particular value exceeds the maximum, the minimum of the children.

A value of 0 effectively locks the child at the minimum of the child. This is used in the example to make sure the scrollers don't grow at all, but you could also use weights to make children grow proportionally.

WEIGHT BAR FOR USER CONTROL

The layout gadget is also capable of giving the user control over the weights. To enable this use the CHILD_WeightBar tag with a value of TRUE. This will cause a weight bar to appear after the child which the user can click and drag to adjust the relative size of the children. Effectively the weights are being manipulated.

Be aware that if you have set the weight to 0 no movement is possible.

If you programmatically want to set the size of the gadget to the left of the weight bar, then you have to do this:

```
        SetGadgetAttrs(layout, window, NULL,
                       LAYOUT_ModifyChild, gadget,
                       CHILD_MinWidth, value,
                       CHILD_MaxWidth, value,
                       CHILD_WeightMinimum, TRUE, // recalc the weight
                       TAG_DONE);
        RethinkLayout(layout, window, NULL, FALSE); // make it take hold
        SetGadgetAttrs(layout, window, NULL,
                       LAYOUT_ModifyChild, gadget,
                       CHILD_WeightMinimum, FALSE, // don't recalc the weight
                       CHILD_MinWidth, ~0,
                       CHILD_MaxWidth, ~0,
                       TAG_DONE);
        RethinkLayout(layout, window, NULL, TRUE); // set it free
```

Also be aware that there is no way to remove a weight bar again.

EFFICIENCY OF COMPLEX LAYOUTS

Please note that many layouts within layout slows down the layout process. It can become quite noticeable on an un-accelerated Amiga. So keep the number of gadgets to a minimum. Both regarding depth and total number of gadgets. 4 levels deep and a total of 50 gadgets can easily be too much for a pleasant experience.

ACTIVATING A GADGET WITHIN A LAYOUT

In plain intuition where all gadgets are top gadgets the intuition function ActivateGadget will activate a gadget. However with hierarchical layouts a new function is needed:

```
BOOL ActivateLayoutGadget(struct Gadget *topLayout, struct Window *win, struct Requester
*req, Object *gadgetToBeActive)
```

topLayout is the layout.gadget you give to the window.class via WINDOW_Layout. It is NOT the layout that immediately contains the gadgetToBeActive.

INSTIGATING A RELAYOUT

With a tree of layouts you sometimes want to relayout a sublayout. You most likely don't want to call WM_RETHINK on the window as that can be quite slow. Instead call this function:

```
BOOL RethinkLayout(struct Gadget *gadget, struct Window *window, struct Requester
*requester, LONG refresh);
```

The first argument is a pointer to the layout you want to rethink

The Event Loop

Just like GadTools builds its own event handling on top of the traditional intuition event handling, so does the **window.class**. For that reason it is NOT possible to combine GadTools and window.class. You choose one or the other.

As you might have noticed, when looking at the first example in this chapter the window.class event handling is different from what you are used to. The idea is the same though. You wait for messages. Then you handle each message in turn. The difference is how you get the message and how you handle it.

```
   GetAttr(WINDOW_SigMask, windowObject, &windowsignal);

   while (!end)
   {
      receivedsignal = Wait(windowsignal);
      while ((result = DoMethod(windowObject, WM_HANDLEINPUT, &code)) != WMHI_LASTMSG)
      {
         switch (result & WMHI_CLASSMASK)
         {
            case WMHI_CLOSEWINDOW:
               end=TRUE;
               break;
         }
      }
   }
```

Another thing to notice is that you only get a single ULONG return value. You don't get an entire **IntuiMessage** struct. Instead you split the return value in an event class value with WMHI_CLASSMASK and for say menu items you can get the traditional code with WMHI_MENUMASK.

Where you used to get IDCMP_CLOSEWINDOW you now get WMHI_CLOSEWINDOW and so on. Some IDCMP events are not delivered as WMHI events, but you can install a hook as described next.

HANDLE IDCMP HOOK

The IDCMP_IDCMPUPDATE, IDCMP_GADGETDOWN and IDCMP_RAWKEY don't have WMHI equivalents, so if for some reason you want to hear about those you need to create a hook and attach it to the window object. First you create a standard **Hook** structure which points to your function, and then use the WINDOW_IDCMPHook tag to set the hook on the window object.

Your function will look like this:

```
processIDCMP(struct Hook *hook, Object *windowObject, struct IntuiMessage *message)
```

You should NOT reply to the message as it has already been replied to and it is a copy you are looking at. The exception is the IDCMP_IDCMPUPDATE which as you know contains a taglist and you thus have to process it before the message is replied. You should still not reply to it yourself. The window.class will take care of that when your function returns.

Also note that you have to make sure WINDOW_IDCMPHookBits and WA_IDCMP have the bits set you want to listen to.

Menus

Please read the chapter dedicated to menus, which also covers how to have menus with the window.class.

Iconification of Windows

Also new in 3.2 is OS level support for an iconification gadget next to the depth and zoom gadgets. Window.class takes advantage of this with the WINDOW_IconifyGadget tag. But you need to do a bit more yourself. You also need to create and supply a messageport with the WINDOW_AppPort tag.

The application needs to react to WMHI_ICONIFY by calling **DoMethod()**

```
   DoMethod(windowObject, WM_ICONIFY, NULL)
```

This will actually close the intuition **Window**, so any pointer to that will no longer be valid.

The application also needs to react to WMHI_UNICONIFY by calling **DoMethod()**

```
   DoMethod(windowObject, WM_OPEN, NULL)
```

This will create the intuition **Window** again, and you must remember to set any menuitem check marks and disables again, as they are not remembered if the menu is built with WINDOW_NewMenu. If on the other hand the menu is added with WINDOW_Menu the program is in control and check marks and disables will survive.

Workbench uses iconification as a means of changing screen mode while programs are running. But for this to work your program should also give up allocated pens, pubscreen lock and possible other similar things. It is encouraged that your program tries to allow screen changes, but please test that your program doesn't assume the screen is the same after un-iconify.

A Complete Example

In this complete example we use the **button.gadget** as a stand-in for all gadgets. This is so the important parts are easier to see. The example creates a window with a more complex layout.. The window can be iconified and restored. It implements an IDCMP hook, but the hook doesn't do much.

Complete Window and Layout Example

```
#include <stdlib.h>

#include <intuition/classusr.h>
#include <gadgets/layout.h>
#include <gadgets/button.h>
#include <classes/window.h>

#define __CLIB_PRAGMA_LIBCALL
#include <proto/alib.h>
#include <proto/exec.h>
#include <proto/layout.h>
#include <proto/window.h>
#include <proto/intuition.h>
#include <proto/button.h>
#include <clib/compiler-specific.h>

char *vers="$VER: BOOPSIdemo 2 (20.12.2020)";

struct IntuitionBase *IntuitionBase;
struct Library *WindowBase;
struct Library *LayoutBase;
struct Library *ButtonBase;

struct Hook IDCMPHook;

void cleanexit(Object *windowObject);
void processEvents(Object *windowObject);

void __SAVE_DS__ __ASM__ processIDCMP(__REG__ (a0, struct Hook *hook),
    __REG__ (a2, Object *obj),
    __REG__ (a1, struct IntuiMessage *msg))
{
   switch (msg->Class)
   {
      case IDCMP_GADGETDOWN:
         /* Do something about it */
         break;
   }
}

int main(void)
{
   struct Window *intuiwin = NULL;
   Object *windowObject = NULL;
   Object *topWhiteLayout = NULL;
   Object *sketchboardGadget = NULL;
   Object *verticalScroller = NULL;
   Object *middleWhiteLayout = NULL;
   Object *horizontalScroller = NULL;
   Object *mainLayout = NULL;
   struct MsgPort *appPort;

   if (! (IntuitionBase = (struct IntuitionBase*)OpenLibrary("intuition.library",47)))
      cleanexit(NULL);

   if (! (WindowBase = OpenLibrary("window.class", 47)))
      cleanexit(NULL);

   if (! (LayoutBase = OpenLibrary("gadgets/layout.gadget", 47)))
      cleanexit(NULL);

   if (! (ButtonBase = OpenLibrary("gadgets/button.gadget", 47)))
      cleanexit(NULL);

   topWhiteLayout = NewObject(LAYOUT_GetClass(), NULL,
         LAYOUT_Orientation, LAYOUT_ORIENT_HORIZ,
         LAYOUT_SpaceInner, TRUE,
         LAYOUT_SpaceOuter, FALSE,
         LAYOUT_AddChild, NewObject(BUTTON_GetClass(), NULL, TAG_END),
         CHILD_WeightedWidth, 10,
         CHILD_WeightBar, TRUE,
         LAYOUT_AddChild, NewObject(BUTTON_GetClass(), NULL, TAG_END),
         CHILD_WeightedWidth, 20,
         LAYOUT_AddChild, NewObject(BUTTON_GetClass(), NULL, TAG_END),
         CHILD_WeightedWidth, 0,
         TAG_DONE);

   sketchboardGadget = NewObject(BUTTON_GetClass(), NULL, TAG_END);

   verticalScroller = NewObject(BUTTON_GetClass(), NULL, TAG_END);

   middleWhiteLayout = NewObject(LAYOUT_GetClass(), NULL,
         LAYOUT_Orientation, LAYOUT_ORIENT_HORIZ,
         LAYOUT_SpaceInner, TRUE,
         LAYOUT_SpaceOuter, FALSE,
         LAYOUT_AddChild, sketchboardGadget,
         CHILD_WeightedWidth, 100,
         LAYOUT_AddChild, verticalScroller,
         CHILD_WeightedWidth, 0,
         TAG_DONE);

   horizontalScroller = NewObject(BUTTON_GetClass(), NULL, TAG_END);

   mainLayout = NewObject(LAYOUT_GetClass(), NULL,
         LAYOUT_Orientation, LAYOUT_ORIENT_VERT,
         LAYOUT_DeferLayout, TRUE,
         LAYOUT_SpaceInner, TRUE,
         LAYOUT_SpaceOuter, TRUE,
         LAYOUT_AddChild, topWhiteLayout,
         CHILD_WeightedHeight, 0,
         LAYOUT_AddChild, middleWhiteLayout,
         CHILD_WeightedHeight, 100,
         LAYOUT_AddChild, horizontalScroller,
         CHILD_WeightedHeight, 0,
         TAG_DONE);

   /* initialize IDCMPHook for listening to IDCMP Messages */
   IDCMPHook.h_Entry    = (ULONG (*)()) processIDCMP;
   IDCMPHook.h_SubEntry = NULL;
```

```
   IDCMPHook.h_Data = NULL;

   appPort = CreateMsgPort();

   windowObject = NewObject(WINDOW_GetClass(), NULL,
         WINDOW_Position, WPOS_CENTERSCREEN,
         WA_Activate, TRUE,
         WA_Title, "BOOPSI window demo",
         WA_DragBar, TRUE,
         WA_CloseGadget, TRUE,
         WA_DepthGadget, TRUE,
         WA_SizeGadget, TRUE,
         WA_InnerWidth, 300,
         WA_InnerHeight, 150,
         WA_IDCMP, IDCMP_CLOSEWINDOW,
         WINDOW_Layout, mainLayout,
         WINDOW_IconifyGadget, TRUE,
         WINDOW_IDCMPHook,     &IDCMPHook,
         WINDOW_IDCMPHookBits, IDCMP_GADGETDOWN,
         WINDOW_AppPort, appPort,
         TAG_DONE);

   if (! windowObject)
      cleanexit(NULL);

   if (! (intuiwin = (struct Window *) DoMethod(windowObject,WM_OPEN, NULL)))
      cleanexit(windowObject);

   processEvents(windowObject);

   DoMethod(windowObject,WM_CLOSE);
   cleanexit(windowObject);
}

void processEvents(Object *windowObject)
{
   ULONG windowsignal;
   ULONG receivedsignal;
   ULONG result;
   ULONG code;
   BOOL end = FALSE;

   GetAttr(WINDOW_SigMask, windowObject, &windowsignal);

   while (!end)
   {
      receivedsignal = Wait(windowsignal);
      while ((result = DoMethod(windowObject, WM_HANDLEINPUT, &code)) != WMHI_LASTMSG)
      {
         switch (result & WMHI_CLASSMASK)
         {
            case WMHI_CLOSEWINDOW:
               end=TRUE;
               break;

            case WMHI_ICONIFY:
               DoMethod(windowObject, WM_ICONIFY, NULL);
               break;

            case WMHI_UNICONIFY:
               DoMethod(windowObject, WM_OPEN, NULL);
               break;
         }
      }
   }
}

void cleanexit(Object *windowObject)
{
   if (windowObject)
      DisposeObject(windowObject);

   CloseLibrary((struct Library*)IntuitionBase);
   CloseLibrary(ButtonBase);
   CloseLibrary(WindowBase);
   CloseLibrary(LayoutBase);
   exit(0);
}
```

Chapter 2
COMMON GADGETS

This chapter introduces many of the modern BOOPSI gadgets, except the **layout.gadget** that was already described in the previous chapter. Also the more complex gadgets like the **texteditor.gadget** will be covered in subsequent chapters.

A Collection of Standard Gadgets

GadTools library started the trend of providing a collection of standard user interface gadgets. The new BOOSPI gadgets presented here follow that trend, but contrary to GadTools the new gadgets are layout aware, subclassable, runtime modifiable, and connectable.

A collection of standard gadgets is important, so that you can concentrate on producing great new applications. It furthermore has the benefit that all applications will have a consistent user interface.

These gadgets are all library based so don't forget to call **OpenLibrary(**"name.gadget", 47**)** with "name" replaced with the name of the gadget. You need a pointer to the class in order to call **NewObject()**. You get that class pointer for say **button.gadget** by calling a library function **BUTTON_GetClass()**. The function for the other classes follow the same pattern. The prototype for **BUTTON_GetClass()** is attained with:

```
#include <proto/button.h>
```

Button.gadget

We already met the **button.gadget** in the last chapter where it was used as a stand-in for more complex gadgets. As its name suggests it acts as a button, but it can do a bit more than the buttons of GadTools or classic intuition. It can show a glyph by specifying either BUTTON_AutoButton or BUTTON_Glyph, and it can act as a toggle by setting BUTTON_PushButton to TRUE.

In 3.2 it has a new interesting attribute BUTTON_TextPadding which when set to TRUE makes sure that the minimum width of the gadget is enlarged a bit, so the text doesn't touch the side borders. This gives the recommended look, so please use it.

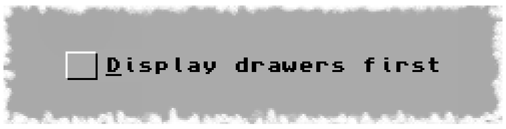
An older attribute BUTTON_DomainString is related in that it can supply a different string for minimum sizing than the string actually shown on screen. The new BUTTON_TextPadding is much more powerful if the aim is simply to try and add some side padding as it alleviates the need to build a special domain string (remember with localization it would have to be built at runtime). The BUTTON_DomainString is probably better used in conjunction with BUTTON_VarArgs.

Other than that it can have images instead of text, and it can even show a bitmap though that is not really recommended. It can also have an array of strings or images via BUTTON_Array and the currently shown is selected with BUTTON_Current.

There are plenty of other attributes, so take a look at the auto docs

Checkbox.gadget

There is even less to tell about **checkbox.gadget**. It just works. The CHECKBOX_TextPlace can be set to either PLACETEXT_LEFT or PLACETEXT_RIGHT and contrary to the GadTools variant the entire area can be clicked with the mouse.

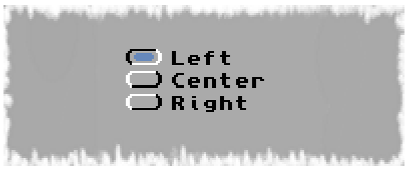


You can set up an IC_TARGET or you can query the state directly with:

```
GetAttr(CHECKBOX_Checked, ptrToGadget, &someLong);
```

Radiobutton.gadget

The **radiobutton.gadget** is another well known kind of gadget. And like the **checkbox.gadget** described above the entire area including the text can be clicked. The gadget makes it much easier to work with than the counterpart in GadTools because layouting is no longer a concern.



You can use GA_TEXT pointing to a NULL terminated array of strings. Another attribute named RADIOBUTTTON_Strings does nothing and should not be used.

Alternatively you can create a linked list of nodes and use RADIOBUTTON_Labels, where the nodes have to be allocated and freed using these two functions:

```
struct Node *AllocRadioButtonNode(UWORD pri, Tag firstTag, ...);
```

```
VOID FreeRadioButtonNode(struct Node *node);
```

The nodes can be manipulated with:

```
VOID SetRadioButtonNodeAttrs(struct Node *node, Tag firstTag, ...);
VOID GetRadioButtonNodeAttrs(struct Node *node, Tag firstTag, ...);
```

Currently the only reason for using the nodes is that you can specify RBNA_UserData for each node.

You can set up an IC_TARGET or you can query which item is selected directly with:

```
GetAttr(RADIOBUTTON_Selected, ptrToGadget, &someLong);
```

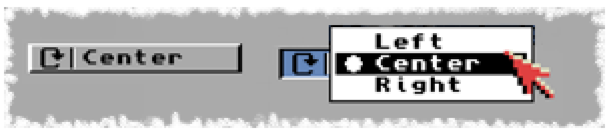
And you can set the selected item too. Just remember to use `SetGadgetAttrs` so the gadget can update it's presence on screen.

Chooser.gadget

The **chooser.gadget** has two very distinct modes of operation, pop-up and drop-down. It is important to understand the differences between these two modes and to use the proper mode in the proper context. The API docs have the following to say about the two modes:

Pop-up Choosers

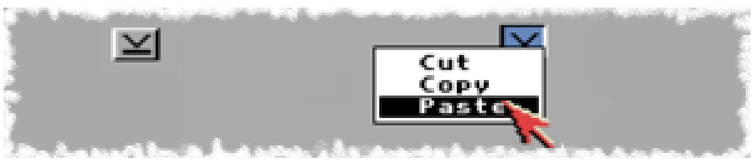
A pop-up is generally used for setting an application mode or state, and in many cases it can replace a GadTools CYCLE_KIND gadget or MX_KIND (radio button) gadget. In this mode, there is a currently active item in the list of selections, which will be displayed in the gadget select button. This has the same advantage of a cycle menu in that it is compact, with the further advantage that all values can be displayed simultaneously and therefore the list of values can be much larger, though generally no more than a dozen items should be displayed.



Drop-down Choosers

A drop-down chooser is for performing an action from a list of available actions. In this mode, the gadget select box contains a title indicating what the actions are for. These actions should all be closely related, and specific to a certain context within the application. This can be used to replace a group of buttons, or a cycle gadget and a button where the cycle gadget modifies the behaviour of the button. Since using the chooser in this mode makes functions effectively hidden, it should generally only be used where compactness is a significant issue.

An alternate use for a drop-down is to use it as a means of accessing a "hot list" for a string or integer gadget. For example, in a word processor you might have a string gadget at the top of the screen for entering the text point size. Beside that you could have a drop-down chooser that lists some common point sizes that would then be copied into your integer gadget and change the text size when you make a selection. When a drop-down is used this way, it is generally desirable not to have the drop-down display a title within the gadget box (pass NULL for CHOOSER_Title) and to make the gadget thin enough so that just the arrow is displayed (use about 20 for GA_Width if not using the chooser within a layout group).



ADDING LABELS

You can supply an array of strings with CHOOSER_LabelArray or a linked list of nodes with CHOOSER_Labels. The nodes are created, freed and manipulated with the functions:

```
struct Node *AllocChooserNode(Tag firstTag, ...);
VOID FreeChooserNode(struct Node *node);
VOID SetChooserNodeAttrs(struct Node *node, Tag firstTag, ...);
VOID GetChooserNodeAttrs(struct Node *node, Tag firstTag, ...);
```

The Node attributes of particular interest are CNA_Text, CNA_Image and CNA_SelImage to specify text and images.

The node attribute CNA_UserData can come in handy, but the boolean attributes CNA_Separator, CNA_Disabled, CNA_ReadOnly really allow you to make advanced lists of choices.

DROP-DOWN FROM A HIDDEN CHOOSER

You don't have to actually have the gadget visible in order to drop down its list of actions. This usage of the **chooser.gadget** is geared towards being used from within another gadget implementation. For example **texteditor.gadget** uses it to show a list of choices dropped down from the cursor position.

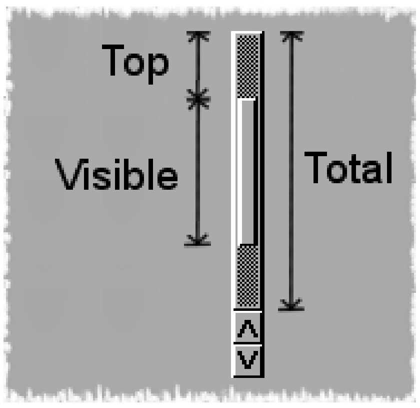
The way for a user to manipulate such a drop-down is via the keyboard. Under 3.2 and earlier the drop down will not react well if the user clicks it with the mouse. This shortcoming is planned to be fixed in 3.3 and already is in OS4 although the OS4 version has other shortcomings from a user perspective.

The steps needed in your gadget to accomplish a similar effect is as follows:

- In OM_NEW create an instance of chooser.gadget with CHOOSER_DropDown and CHOOSER_Hidden both set to TRUE. Don't put it inside a layout or do anything with it except as described below.
- When needed, set the labels and call **ShowChooser**(). This could for example happen as a result of a key event. It is important that you then remain active by returning GMR_MEACTIVE.
- In GM_HANDLEINPUT and GM_HANDLESCROLL you should check if the chooser is active (it may become inactive on it's own initiative) and if active you should pass the gpi message on to the chooser and not react on the message on your own. If it returns GMR_NOREUSE you should call **HideChooser**() but your own gadget could still choose to remain active.
- In GM_RENDER you need to forward the render message to the chooser (if active). You probably should render your own gadget as well, as the render message might be intended for your gadget in the first place.
- In the future other functions might need to be forwarded, so right now doing this is a rather brittle construction. GM_HELPTEST and GM_QUERYHOVERED come to mind as methods that should forward to the chooser.

Scroller.gadget

The **scroller.gadget** is another well known type of gadget. The values it can hold are full ULONG (ie 32 bit) SCROLLER_Top, SCROLLER_Visible and SCROLLER_Total.



The top value is ensured to never be greater than total minus visible. If total is less than visible then top will always be 0.

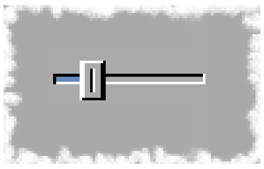
The arrows can be hidden by setting SCROLLER_Arrows to FALSE, and the amount the arrows change the top can be controlled with SCROLLER_ArrowDelta.

It can signal a task when active if you specify SCROLLER_SignalTask and SCROLLER_SignalTaskBit.

SCROLLER_Stretch despite being defined in header files doesn't do anything and should not be used

Slider.gadget

In the past a slider was not a separate gadget class. Instead scrollers were abused to provide comparable functionality. Now with **slider.gadget** there is also a distinct look, so the user can tell them apart.



The slider value is limited between a min and max value that can be controlled with SLIDER_Min and SLIDER_Max respectively. The min and max can be visually flipped by setting SLIDER_Invert to TRUE. The current slider value can be set and read with SLIDER_Level. The delta amount when clicking on either side of the knob can be controlled with SLIDER_KnobDelta.

It is possible to show tick marks by setting SLIDER_Ticks to TRUE, and the size of the ticks can alternate in size if SLIDER_ShortTicks is set to TRUE. You can also control the size of the ticks with SLIDER_TickSize.

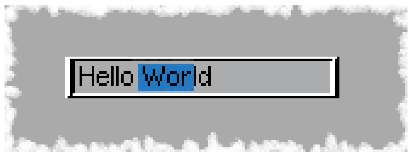
In 3.2 the slider can also display the value if you specify a classic printf-style format string using SLIDER_LevelFormat. The SLIDER_LevelMaxLen should probably be specified at the same time and SLIDER_LevelDomain specifies a string that is measured to ensure there is enough space to show the text. The SLIDER_LevelDomain string is however never rendered. Alternative to a domain string you can specify the width in pixels that should be reserved.

SLIDER_LevelPlace and SLIDER_LevelJustify can be used for more control over where the value is displayed.

There is even more control over the display of the level however, as you can specify a level hook that is called before the value is rendered. The hook returns an ULONG that is then passed through the format string. In theory if you have enough control over your buffer the value returned by the hook could be a string pointer and the format could be %s

String.gadget

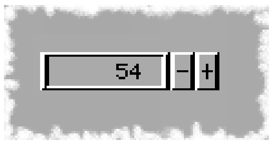
The **string.gadget** basically does the same as the intuition and GadTools gadgets, but it does more. It supports cut, copy and paste to the clipboard using the Amiga+X, Amiga+C and Amiga+V shortcuts. To accomplish this it has another feature not in GadTools: It supports making a text selection using the mouse, or with shift + cursor keys. Likewise you can also move the cursor to beginning and end with ctrl + cursor keys, or on modern keyboards with Home and End. These features come built in without any further work needed by the application programmer.



Exactly like its ancestors you can add filtering. For details you would have to refer to the old Amiga ROM Kernel Reference Manual: Libraries.

Integer.gadget

This gadget allows the user to type in a number, or use arrow buttons to increase or decrease the number. The **integer.gadget** can have a minimum and maximum value which are signed LONG. The current value is accessed with the INTEGER_Number tag.



The arrow buttons change the value by the amount specified by INTEGER_SkipVal, or the arrows can be hidden completely with INTEGER_Arrows set to FALSE.

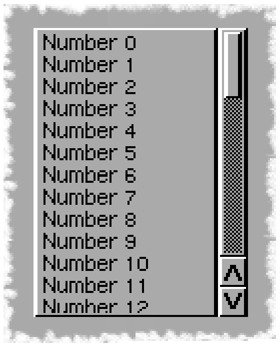
Users generally prefer a slider over numerically entering numbers in most cases, so this is worth considering when deciding which gadget to use.

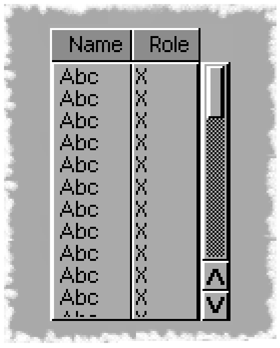
The size of the gadget can be influenced through INTEGER_MinVisible and INTEGER_MaxChars in different ways. Additionally INTEGER_MaxChars also affects the number that can be entered, and don't forget to allow for a sign character.

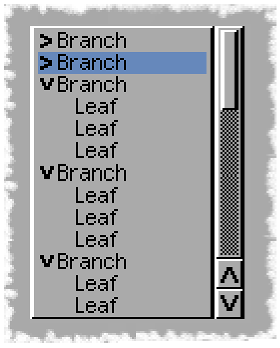
Chapter 3
LISTBROWSER GADGET

Many programs need a way to show data in either a list, a multicolumn list, or a tree. The **listbrowser.gadget** answers all of these needs. It is a very versatile gadget that can present a lot of data in a very intuitive way.

This chapter will explain how to create such a gadget, populate it with data, listen to events and manipulate the gadget, as well as the more subtler ways to make it do what you want.







The user can scroll up and down the list of nodes, and depending on what features are enabled or disabled, she can select, multi-select, rename, or doubleclick the nodes. You can make the gadget keep focus so she can interact with it using the keyboard. In tree mode she can expand or collapse the branches revealing deeper generations of nodes.

When used to show a list or multicolumn list it can even sort the nodes, and you can specify the comparison using a hook. The column title can show an arrow so the user can change the sorting interactively.

It is possible for the nodes to show an image and the text can be word-wrapped automatically where '\n' additionally causes a hard wrap.

Creating a ListBrowser Gadget

Like creating any other of the new gadgets, the first job is to make sure the class is available by opening its library. And don't forget to close the library again when quitting.

```
ListBrowserBase = OpenLibrary("gadgets/listbrowser.gadget", 47))
```

Then it is simply a case of calling **NewObject** like this:

```
listBrowser = NewObject(LISTBROWSER_GetClass(), NULL,
        TAG_DONE);
```

And then put it into a layout as described in chapter 1.

CREATING NODES

A list is not much use without some data to show and this is organized in an Exec **struct List**. The nodes you add to the list *have* to be allocated and freed with the following functions.

```
struct Node *AllocListBrowserNode(UWORD numcolumns, Tag, ...)
void FreeListBrowserNode(struct Node *node)
```

The code would look something like this:

```
struct List content;
struct Node *node;

NewList(&content);

node = AllocListBrowserNode(1,
      LBNCA_Text, "some text",
      TAG_DONE))

if (node)
   AddTail(&content, node);

SetGadgetAttrs(listBrowser, window, NULL, LISTBROWSER_Labels, &content, TAG_DONE);
```

The creation of the list and nodes can be done ahead of time, and LISTBROWSER_Labels set while creating the gadget,

ADDING, REMOVING, OR MODIFYING NODES

You can add nodes and remove nodes by modifying the list. And you can manipulate the nodes.

> *Warning:* Adding or removing nodes, or changing attributes of any node in the list, requires that the list is *detached* from the gadget. This is done by setting LISTBROWSER_Labels to ~0,

```
SetGadgetAttrs(listBrowser, window, NULL, LISTBROWSER_Labels, ~0, TAG_DONE);
```

With that done it is safe to reorder the nodes in the list, remove or add more nodes, and to manipulate the nodes. Any changing of attributes is done with:

```
void SetListBrowserNodeAttrs(struct Node *node, Tag, ...)
```

Later in this chapter, we will cover most attributes you can set, as the various topics are discussed. You are also encouraged to look at the *gadgets/listbrowser.h* header file.

When done manipulating the list,, the list should be attached to the listbrowser again.

```
SetGadgetAttrs(listBrowser, window, NULL, LISTBROWSER_Labels, &content, TAG_DONE);
```

CLEANUP AND FREEING NODES

The gadget doesn't automatically dispose of the nodes you have allocated, so it is important that you free them. Remember they are nodes in a list so do it safely, where you don't use a pointer from freed memory to access the next node. You could do it like this:

```
while (node = GetTail(&contents)
   FreeListBrowserNode(node);
```

There is also a function that does this for you, when your program doesn't need to do any additional processing of each node:

```
FreeListBrowserList(&contents);
```

In both cases, unless the gadget itself has been disposed of first, it is important to detach the list from the gadget first. For example when just clearing the list before putting in new nodes.

A Simple ListBrowser Example

```
#include <stdlib.h>

#include <intuition/classusr.h>
#include <gadgets/layout.h>
#include <gadgets/listbrowser.h>
#include <classes/window.h>

#define __CLIB_PRAGMA_LIBCALL
#include <proto/alib.h>
#include <proto/exec.h>
#include <proto/layout.h>
#include <proto/listbrowser.h>
#include <proto/window.h>
#include <proto/intuition.h>
#include <proto/utility.h>

char *vers="$VER: BOOPSIdemo 1 (09.12.2019)";

struct IntuitionBase *IntuitionBase;
struct Library *WindowBase;
struct Library *LayoutBase;
struct Library *ListBrowserBase;

void cleanexit(Object *windowObject);
void processEvents(Object *windowObject);

int main(void)
{
   struct Window *intuiwin = NULL;
   Object *windowObject = NULL;
   Object *mainLayout = NULL;
   Object *listBrowser = NULL;
```

```
struct List contents;
WORD i;

if (! (IntuitionBase = (struct IntuitionBase*)OpenLibrary("intuition.library",47)))
      cleanexit(NULL);

if (! (UtilityBase = OpenLibrary("utility.library",47)))
      cleanexit(NULL);


if (! (WindowBase = OpenLibrary("window.class", 47)))
      cleanexit(NULL);

if (! (LayoutBase = OpenLibrary("gadgets/layout.gadget", 47)))
      cleanexit(NULL);

if (! (ListBrowserBase = OpenLibrary("gadgets/listbrowser.gadget", 47)))
      cleanexit(NULL);

NewList(&contents);

for (i = 0; i < 42; i++)
{
      UBYTE buffer[64];
      struct Node *node;

      SNPrintf(buffer, 64, "Number %ld", i);

      node = AllocListBrowserNode(1,
            LBNCA_CopyText, TRUE,
            LBNCA_Text, buffer,
            TAG_DONE);

      if (node)
            AddTail(&contents, node);
}

listBrowser = NewObject(LISTBROWSER_GetClass(), NULL,
            LISTBROWSER_Labels, &contents,
            TAG_DONE);

mainLayout = NewObject(LAYOUT_GetClass(), NULL,
            LAYOUT_Orientation, LAYOUT_ORIENT_VERT,
            LAYOUT_DeferLayout, TRUE,
            LAYOUT_SpaceInner, TRUE,
            LAYOUT_SpaceOuter, TRUE,
            LAYOUT_AddChild, listBrowser,
            TAG_DONE);

windowObject = NewObject(WINDOW_GetClass(), NULL,
            WINDOW_Position, WPOS_CENTERSCREEN,
            WA_Activate, TRUE,
            WA_Title, "Simple ListBrowser demo",
            WA_DragBar, TRUE,
            WA_CloseGadget, TRUE,
            WA_DepthGadget, TRUE,
            WA_SizeGadget, TRUE,
            WA_InnerWidth, 300,
            WA_InnerHeight, 150,
            WA_IDCMP, IDCMP_CLOSEWINDOW,
            WINDOW_Layout, mainLayout,
            TAG_DONE);
```

```
   if (! windowObject)
         cleanexit(NULL);

   if (! (intuiwin = (struct Window *) DoMethod(windowObject,WM_OPEN, NULL)))
         cleanexit(windowObject);

   processEvents(windowObject);

   DoMethod(windowObject,WM_CLOSE);
   cleanexit(windowObject);
}

void processEvents(Object *windowObject)
{
   ULONG windowsignal;
   ULONG receivedsignal;
   ULONG result;
   ULONG code;
   BOOL end = FALSE;

   GetAttr(WINDOW_SigMask, windowObject, &windowsignal);

   while (!end)
   {
         receivedsignal = Wait(windowsignal);
         while ((result = DoMethod(windowObject, WM_HANDLEINPUT, &code)) != WMHI_LASTMSG)
         {
               switch (result & WMHI_CLASSMASK)
               {
                     case WMHI_CLOSEWINDOW:
                        end=TRUE;
                        break;
               }
         }
   }
}

void cleanexit(Object *windowObject)
{
   if (windowObject)
         DisposeObject(windowObject);

   CloseLibrary((struct Library*)IntuitionBase);
   CloseLibrary(UtilityBase);
   CloseLibrary(WindowBase);
   CloseLibrary(LayoutBase);
   CloseLibrary(ListBrowserBase);
   exit(0);
}
```

Nodes, More Than Just Text

Nodes can be so much more than a simple piece of text:

TEXT OR INTEGER

Specify either **LBNCA_Text** or **LBNCA_Integer** to show a string or integer. Note that integer is a pointer to a ULONG, so you can actually update the value without setting the attribute again.

If you want to fire and forget, then you can specify **LBNCA_CopyText** or **LBNCA_CopyInteger** respectively set to TRUE, to tell the node to make an internal copy. It must precede the text tag!

CHECKBOX

A checkmark can appear if you set **LBNA_CheckBox** to TRUE and the state of the checkbox is get and set with **LBNA_Checked**.

IMAGE AND SELECTED IMAGE

An image can be shown by setting **LBNCA_Image** to a pointer to an image object. Any subclass of image will do. An alternative image when the node is selected is also possible with **LBNCA_SelImage**.

JUSTIFICATION, COLORS AND READ-ONLY

The text, integer and images can be horizontally justified, by setting **LBNCA_Justification** to LCJ_LEFT, LCJ_CENTER or LCJ_RIGHT.

The foreground and background colors can be set with **LBNCA_FGPen**, **LBNCA_BGPen**, and **LBNCA_FillPen**. It however requires that **LBNA_Flags** has the LBFLG_CUSTOMPENS flag or'ed in.

Another flag for **LBNA_Flags** is LBFLG_READONLY. With this and some of the other ways to change the look you could implement separators, or section headlines in a list.

RENDER HOOK

If the above options are still not enough, an alternative is to implement a render hook that can render the node exactly as required. The hook is specified with **LBNCA_RenderHook** and then the height of the node needs to be specified with **LBNCA_HookHeight**. The hook effectively works the same way as in GadTools.

WORD WRAP

Dynamic word wrapping can be enabled by setting **LBNCA_WordWrap** to TRUE and also **LISTBROWSER_WordWrap** to TRUE on the gadget. From 3.2 onward, but not OS4, the wrapping is dynamic and also wraps if a newline character is encountered.

EDITABLE TEXT

The text of the node can be made editable with **LBNCA_Editable** set to TRUE and **LBNCA_MaxChars** set to the max length of the buffer you supplied. The node will enter into editing mode when the text is double clicked. The lisbrowser also needs to have **LISTBROWSER_Editable** set to TRUE.

By setting **LISTBROWSER_EditTrigger** to LBET_DELAYEDSECOND on the gadget, you can make it so edit mode is entered on a second click that isn't a double click. This means the user will have to wait the doubleclick period after selecting the node before clicking a second time. This allows the program to react to a double click while at the same time still allow edit mode.

USERDATA

There is also a tag for setting an ULONG, the meaning of which is for the program to decide. It could for example be a pointer to the underlying data this node represents.

Selections and User Interaction

There are many variations on how the user can select nodes. From not being able to select any nodes, to select multiple. And as mentioned in the previous section you can even make specific nodes read-only.

GA_ReadOnly
: To have a listbrowser.gadget that doesn't allow any selections, but still allows the user to scroll through and look at, you set this tag to TRUE.

LISTBROWSER_MultiSelect
: Single selections are the default. So to allow multiple selected nodes just set this to TRUE. A related tag to read is **LISTBROWSER_NumSelected** to see how many nodes are selected and to find out which ones you have to walk through your list and query the **LBNA_Selected** of each node. The **LISTBROWSER_PersistSelect** tag controls that the user doesn't have to hold down the shift key to select more than one node.

Mainly of relevance for the default single selection model, it can make sense to not show the selection. Say the purpose of the list is to do some action when clicking. In that case having it read only will not work. Instead set **LISTBROWSER_ShowSelected** to FALSE.

The current selection can be set or queried with **LISTBROWSER_Selected** and **LISTBROWSER_SelectedNode** being an index or the actual node respectively. You only have to set one of them and the other will be updated automatically. If your selection model is multi select then it doesn't make much sense to work with these two tags.

KEYBOARD NAVIGATION

From 3.2 you can enable keyboard navigation. On the Amiga this means that the gadget needs to have focus. Keyboard navigation works if focus is given via code or TAB navigation. However if you really want the user to do keyboard navigation you should set the **LISTBROWSER_StayActive** tag to TRUE, as it ensures that the focus is kept when the user clicks the gadget.

When navigating with the cursor keys a new cursor concept comes into play. It is shown visually as a slightly different colored "selection". Two tags can be queried: **LISTBROWSER_CursorSelect** and **LISTBROWSER_CursorNode**.

The user can stop the gadget focus by pressing the Escape key or most other special keys, which will also un-render the cursor. Pressing the spacebar emulates a single mouse click and pressing the enter key emulates a double click.

If the user types normal letters a search string will be constructed and the first node starting with this search string will be selected. If the user takes more than doubleclick time pressing the next letter the search string will be cleared.

LISTENING TO USER INTERACTION

When the user either with the mouse or keyboard makes selections or clicks on column titles etc the program would like to be told. The way to be told is to set up an IC_TARGET and wait to be notified about **LISTBROWSER_RelEvent**. If instead you query this tag when you get a WMHI_GADGETUP you will not be told about keyboard actions. The possible events are:

LBRE_NORMAL
LBRE_HIDECHILDREN
LBRE_SHOWCHILDREN
LBRE_EDIT
LBRE_DOUBLECLICK
LBRE_CHECKED
LBRE_UNCHECKED
LBRE_TITLECLICK
LBRE_COLUMNADJUST

LBRE_EDITTABNEXT
LBRE_EDITTABPREV

Hopefully they are self explanatory and the **LISTBROWSER_RelColumn** provides further info for a multi column list. The affected node is the **LISTBROWSER_CursorNode** and not the **LISTBROWSER_SelectedNode**. Although unless keyboard navigation is activated there is no practical problem in thinking it is the selected node, which is what makes legacy code still work.

Multicolumn List and Column Titles

First of all, multicolumn lists are not different. A single column list is simply a multicolumn list with only 1 column. It follows that you can also have headers for a single column list.

In order to have more than 1 column you must define the columns and tell the gadget. In the past you could do this with a static array, but it is much safer and better to use the special functions for doing this.

```
struct ColumnInfo *AllocLBColumnInfo( ULONG columns, ... );
VOID FreeLBColumnInfo( struct ColumnInfo *columninfo );
```

Column titles can be turned visible with LISTBROWSER_Columns set to true. But remember to give a title for each column, even if simply "".

The list can be sorted programmatically, or the user can activate the sorting. The user activation of sorting requires the individual columns marked as auto sort. Likewise the column titles need to be visible, so the user has a place to click. We will go into more details about sorting later in this chapter.

DEFINING EACH COLUMN

The way to define an arbitrary number of columns is by using LBCIA_Column to specify which column the *following* tags refer to. Then follow LBCIA_xxxxx tags, before another LBCIA_Column is specified to move on to the next column.

It could look like this, and it is important that you don't try to specify more columns than you declare in the first argument:

```
ci = AllocLBColumnInfo(3,
         LBCIA_Column, 0,
         LBCIA_Title, "First Column",
         LBCIA_Width, 400,
         LBCIA_Column, 1,
         LBCIA_Title, "Second Column",
         LBCIA_Width, 80,
         LBCIA_Column, 2,
         LBCIA_Title, "Third Column",
         LBCIA_Width, 80,
         TAG_DONE);
```

Then you just give this to the listBrowser via the LISTBROWSER_ColumnInfo tag.

GENERAL COLUMN INFO TAGS

In this section we go through some of the tags used to define the columns. There are further column info tags described in the section about sorting. If you afterwards want to modify it or get some values you have:

```
LONG SetLBColumnInfoAttrs( struct ColumnInfo *columninfo, ... );
LONG GetLBColumnInfoAttrs( struct ColumnInfo *columninfo, ... );
```

But remember to do so with a detached list.

LBCIA_Column
Specifies which column the following LBCIA_ tags apply to. The first column is 0,

LBCIA_Title
The title of the column. It must be set if LISTBROWSER_ColumnTitles is TRUE.

LBCIA_Weight
The relative weight to apply to this column, expressed as a percentage value. For example, a value of 40 means the column should take up 40% of the width. Note that columns are weighted by default.

LBCIA_Width
The width of the column in pixels. This and the LBCIA_Weight tag overrule each other so the last one specified is used.

LBCIA_UserData (V47)
Arbitrary user data for this column.

See also the section about sorting below for more tags related to sorting.

NODES FOR A MULTI COLUMN LIST

Even for multi column lists, there is still only *one node per row*. The nodes are allocated like before. However you change the first argument to number of columns and like when defining the columns, you do it in sequences with a leading tag of LBNA_Columnn specifying the column, and then any number of LBNCA_xxxx tags. Note the C in LBNCA meaning it is specific for the column whereas LBNA applies to the entire node.

```
node = AllocListBrowserNode(2,
               LBNA_Column, 0,
               LBNCA_Text, "Abc",
               LBNA_Column, 1,
               LBNCA_Text, "X",
               TAG_DONE);
```

Warning: Do not reference more columns than stated in the first argument. It will crash sooner or later.

Sorting

Sorting was mentioned earlier in this chapter, and now we go into the details of it. Single column lists can be sorted, and multi column lists can be sorted by one column at a time. But tree lists can not be sorted, as that doesn't make much sense.

Several things need to be set up before you can sort programmatically. And additional attributes need to be set to enable the user to sort by just clicking on the column titles.

To sort there must be order. The sorting is done by:

- ln_pri of the node. This is always checked.
- If the priority was equal the following is how the comparison is made:
    1. If there is a comparison hook the hook is used
    2. If both have strings **Strnicmp()** from utility.library is used.
    3. If both have integers a numerical comparison is made.
    4. They are considered equal

THE COMPARE HOOK

The comparison hook is a normal utility.library hook structure. In our example, only the primary function is implemented using directives to tell the compiler on how to pass arguments. The return value should be a signed LONG value. Positive means greater than, zero means equal, and negative means less than.

Since hook functions return ULONG some casting is needed to make sure it compiles nicely while it still works.

Even though the direction is part of the message, the hook is not supposed to work differently. The direction is for information only, and the outer sort will make sure it is respected.

```
ULONG __SAVE_DS__ __ASM__ compare(__REG__(a0, struct Hook *h), __REG__(a2, VOID *o),
__REG__(a1, struct LBSortMsg *msg))
{
   // This compare only compares the first character of the strings;
   // And it assumes the strings are valid, which production code should never assume.
   return (ULONG)(*msg->lbsm_DataA.Text - *msg->lbsm_DataB.Text);
}
```

The hook structure is initialized like this:

```
hook.h_Entry    = (ULONG (*)()) compare;
hook.h_SubEntry = NULL;
hook.h_Data = some_userdata;
```

INVOKING THE SORTING METHOD

If the program wants to sort the nodes it is just a matter of invoking the **LBM_SORT** method on the gadget. The only difficult point is filling in the **struct lbSort** message, but this can be done through varargs. Eg to sort by the 2nd column in forward direction using a hook the following should be used.

```
DoGadgetMethod(listbrowser, win, NULL, LBM_SORT, NULL, 2, LBMSORT_FORWARD, &hook);
```

As mentioned above, a hook is not strictly needed, just pass NULL instead if the default ordering is sufficient.

Note: If the column is not enabled for sorting no sorting will happen. Read more about the **LBCIA_Sortable** tag below.

USER INITIATED SORTING

It is also possible to let the user be in control of the sorting. All it takes is setting some columninfo tags. Namely the **LBCIA_AutoSort** tag and most likely the **LBCIA_SortArrow** tag too.

This is much simpler than waiting for **LISTBROWSER_RelEvent** with a value of LBRE_TITLECLICK and then invoking the sorting method. However auto sort doesn't work until 3.2.1

COLUMN INFO TAGS RELATED TO SORTING

These are the column info tags related to sorting:

LBCIA_Sortable (V47)

Marks the column as sortable.

Together with **LISTBROWSER_TitleClickable** it makes the gadget send out a LBRE_TITLECLICK. It would then be up to the program to react and actually invoke the sorting. But see **LBCIA_AutoSort** below for how to automate it.

LBCIA_SortArrow (V47)

Makes the column show a sort arrow. However it is only visible when the list is sorted by this column, either when LBM_SORT is invoked on this column, or when **LBCIA_AutoSort** was specified and the user activated the sort on the column.

The direction of the arrow depends on how the column was sorted. If the LBM_SORT method is invoked then the **lbs_Direction** field determines which arrow to display. If the column is auto-sorted then the **LBCIA_SortDirection** field is used.

LBCIA_AutoSort (V47)

When set to TRUE the gadget automatically invokes LBM_SORT when the user clicks the title, using the **LBCIA_SortDirection** and **LBCIA_CompareHook** tags.

When the gadget is initially created the **LISTBROWSER_SortColumn** defines which column is sorted . When the column is selected again the sort direction will be toggled.

If TRUE then **LBCIA_Sortable** is also TRUE implicitly.

LBCIA_SortDirection (V47)

The current sort direction for the column.

The direction is one of the following values:

LBMSORT_FORWARD -- Forward sort (down sort arrow)
LBMSORT_REVERSE -- Reverse sort (up sort arrow)

The value changes when **LBCIA_AutoSort** is enabled, and the user changes the direction.

It is possible to use the LBCIA_SortDirection tag and the LISTBROWSER_SortColumn tag together to save and restore sorting user preferences.

LBCIA_CompareHook (V47)

The sort comparison hook to use for this column.

Tree list

As mentioned at the beginning of the chapter, it is also possible to show a tree. Once again it is not that different from a regular list, But it does mean that you shouldn't do any kind of sorting of the nodes.

Initialize the **LISTBROWSER_Hierarchical** tag to TRUE on the listbrowser.gadget to enable the tree mode.

Nodes are then inserted into the list in the order they should appear if all branches are open. Each node should additionally specify **LBNA_Generation** with values starting from 0. It helps to think of the generation number as the indentation level of the node.

Any branch node should have the LBFLG_HASCHILDREN bit set in its **LBNA_Flags**.

If a branch should initially be collapsed the child nodes need to have the to LBFLG_HIDDEN bit set in **LBNA_Flags**. Alternatively the branch node should ensure the branch image is correct by setting the LBFLG_SHOWCHILDREN bit.

When the user clicks to show or hide a branch, the gadget will update these bits automatically. So the above is simply to get the initial state correct.

The branch indicator image can be overwritten with **LISTBROWSER_ShowImage** and **LISTBROWSER_HideImage** although it is strongly encouraged to keep the default as it will allow the user to choose the images in a future GUI preference editor. Similarly it is possible to set a **LISTBROWSER_LeafImage**.

FUNCTIONS TO SHOW AND HIDE CHILDREN

The following functions exist to show and hide children. The can be called without detaching and reattaching the list.

```
VOID ShowListBrowserNodeChildren( struct Node *node, LONG depth );
VOID HideListBrowserNodeChildren( struct Node *node );
VOID ShowAllListBrowserChildren( struct List *list );
VOID HideAllListBrowserChildren( struct List *list );
```

A Complete Example

```
#include <stdlib.h>

#include <intuition/classusr.h>
#include <intuition/icclass.h>
#include <gadgets/layout.h>
#include <gadgets/listbrowser.h>
#include <classes/window.h>

#include <clib/compiler-specific.h>

#define __CLIB_PRAGMA_LIBCALL
#include <proto/alib.h>
#include <proto/exec.h>
#include <proto/layout.h>
#include <proto/listbrowser.h>
#include <proto/window.h>
#include <proto/intuition.h>
#include <proto/utility.h>

char *vers="$VER: BOOPSIdemo 1 (09.12.2019)";

struct IntuitionBase *IntuitionBase;
struct Library *WindowBase;
struct Library *LayoutBase;
struct Library *ListBrowserBase;

void cleanexit(Object *windowObject);
void processEvents(Object *windowObject);

struct List treeNodes;
struct List detailNodes;
Object *treeBrowser = NULL;
Object *detailBrowser = NULL;
struct ColumnInfo *ci;
struct Window *intuiwin = NULL;
struct Hook CompareHook;
struct Hook IDCMPHook;

struct {STRPTR text; ULONG cmpLen; STRPTR RAM;} data[] =
{
 {"Amiga",            5, ""},
 {"Amiga 1000",       0, "256 KB"},
 {"Amiga 500",        0, "512 KB"},
 {"Amiga 2000",       0, "1 MB"},
 {"Amiga 600",        0, "1 MB"},
 {"Amiga 1200",       0, "2 MB"},
 {"Amiga 3000",       0, "2 MB"},
 {"Amiga 4000",       0, "2 MB"},
 {"Amiga CDTV",       0, "1 MB"},
 {"Commodore",        9, ""},
 {"Commodore PET",    0, "4 KB"},
 {"Commodore VIC-20", 0, "5 KB"},
 {"Commodore 64",     0, "64 KB"},
 {"Commodore 16",     0, "16 KB"},
 {"Commodore Plus/4", 0, "64 KB"},
 {"Commodore 128",    0, "128 KB"},
 {NULL,               0, NULL}
};

void installDetails()
{
  WORD i, treeIndex = 0;
  struct Node *treeNode;
  struct Node *node;
  ULONG sel;

  SetGadgetAttrs(detailBrowser, intuiwin, NULL, LISTBROWSER_Labels, ~0, TAG_DONE);

  FreeListBrowserList(&detailNodes);

  /* we do this loop outmost so we can only ever add models once */
  for (i = 0; data[i].text != NULL; i++)
  {
     if (data[i].cmpLen) /* don't show "Amiga" or "Commodore" in the details */
        continue;

     /* Now figure out if it, or it category was selected in the tree */
     for (treeIndex = 0, treeNode = treeNodes.lh_Head; treeNode->ln_Succ; treeIndex++, treeNode =
treeNode->ln_Succ)
     {
        GetListBrowserNodeAttrs(treeNode, LBNA_Selected, &sel, NULL);

        if (sel)
        {
           ULONG cmplen = data[treeIndex].cmpLen;
           if (!cmplen)    /* if not headline we only want exact matches */
              cmplen = 50;

           if (Strnicmp(data[treeIndex].text, data[i].text, cmplen) != 0)
              continue;

           node = AllocListBrowserNode(3,
              LBNA_Column, 0,
              LBNCA_CopyText, TRUE,
              LBNCA_Text, data[i].text,
              LBNA_Column, 1,
              LBNCA_Editable, TRUE,
              LBNCA_HorizJustify, LCJ_RIGHT,
              LBNCA_Text, data[i].RAM,
              LBNA_Column, 2,
              LBNCA_CopyText, TRUE,
              LBNCA_Text, "edit me",
              LBNCA_MaxChars, 100,
              LBNCA_Editable, TRUE,
              LBNCA_Justification, LCJ_LEFT,
              TAG_DONE);

                          if (node)
                          {
                             AddTail(&detailNodes, node);
```

```
                            break; /* Added, so stop looking for more selections */
                        }
                }
        }
  }

  SetGadgetAttrs(detailBrowser, intuiwin, NULL, LISTBROWSER_Labels, &detailNodes, TAG_DONE);
}

static ULONG asValue(STRPTR s)
{
  ULONG v = atoi(s);

  switch(s[strlen(s)-2])
  {
  case 'K':
     return v*1024;
  case 'M':
     return v*1024*1024;
  default:
     return v;
  }
}

static ULONG __SAVE_DS__ __ASM__ myCompare(__REG__(a0, struct Hook *hook), __REG__(a2, Object *obj),
                   __REG__(a1, struct LBSortMsg *msg))
{
         return asValue(msg->lbsm_DataA.Text) - asValue(msg->lbsm_DataB.Text);
}


void __SAVE_DS__ __ASM__ processIDCMP(__REG__(a0, struct Hook *hook), __REG__(a2, Object *obj),
        __REG__(a1, struct IntuiMessage *msg))
{
  struct TagItem *taglist = (struct TagItem *) msg->IAddress;
  struct TagItem                    *tag;

  switch (msg->Class)
  {
     case IDCMP_IDCMPUPDATE:
        while (tag = NextTagItem(&taglist))
        {
           if (tag->ti_Tag == LISTBROWSER_CursorNode)
              installDetails();
        }
        break;
  }
}

int main(void)
{
  Object *windowObject = NULL;
  Object *mainLayout = NULL;
  WORD i;

  if (! (IntuitionBase = (struct IntuitionBase*)OpenLibrary("intuition.library",47)))
           cleanexit(NULL);

  if (! (UtilityBase = OpenLibrary("utility.library",47)))
           cleanexit(NULL);


  if (! (WindowBase = OpenLibrary("window.class", 47)))
           cleanexit(NULL);

  if (! (LayoutBase = OpenLibrary("gadgets/layout.gadget", 47)))
           cleanexit(NULL);

  if (! (ListBrowserBase = OpenLibrary("gadgets/listbrowser.gadget", 47)))
           cleanexit(NULL);
```

```
/* initialize IDCMPHook for listening to IDCMP Messages */
IDCMPHook.h_Entry    = (ULONG (*)()) processIDCMP;
IDCMPHook.h_SubEntry = NULL;
IDCMPHook.h_Data = NULL;

NewList(&treeNodes);
NewList(&detailNodes);

for (i = 0; data[i].text != NULL; i++)
{
         struct Node *node;

         node = AllocListBrowserNode(1,
                 LBNA_Generation, data[i].cmpLen ? 1 : 2,
                 LBNCA_CopyText, TRUE,
                 LBNCA_Text, data[i].text,
                 LBNA_Flags, data[i].cmpLen ? LBFLG_HASCHILDREN : LBFLG_HIDDEN,
                 TAG_DONE);

         if (node)
                 AddTail(&treeNodes, node);
}

treeBrowser = NewObject(LISTBROWSER_GetClass(), NULL,
      LISTBROWSER_Labels, &treeNodes,
      LISTBROWSER_ShowSelected, TRUE,
      LISTBROWSER_MultiSelect, TRUE,
      LISTBROWSER_Hierarchical, TRUE,
      LISTBROWSER_StayActive, TRUE,
      ICA_TARGET, ICTARGET_IDCMP,
      GA_RelVerify, TRUE,
                 TAG_DONE);

/* initialize CompareHook for sorting the ram column */
CompareHook.h_Entry    = (ULONG (*)()) myCompare;
CompareHook.h_SubEntry = NULL;
CompareHook.h_Data = NULL;

ci = AllocLBColumnInfo(3,
      LBCIA_Column, 0,
      LBCIA_Flags, CIF_RIGHT,
      LBCIA_AutoSort, TRUE,
      LBCIA_Title, "Model",
      LBCIA_Weight, 40,
      LBCIA_Column, 1,
      LBCIA_Flags, CIF_RIGHT,
      LBCIA_AutoSort, TRUE,
      LBCIA_SortArrow, TRUE,
      LBCIA_Title, "RAM",
      LBCIA_Weight, 30,
      LBCIA_CompareHook, &CompareHook,
      LBCIA_Column, 2,
      LBCIA_AutoSort, TRUE,
      LBCIA_Title, "Comment",
      LBCIA_Weight, 30,
      TAG_DONE);

detailBrowser = NewObject(LISTBROWSER_GetClass(), NULL,
      LISTBROWSER_Labels, &detailNodes,
      LISTBROWSER_ShowSelected, TRUE,
      LISTBROWSER_ColumnInfo, ci,
      LISTBROWSER_ColumnTitles, TRUE,
      LISTBROWSER_TitleClickable, TRUE,
      LISTBROWSER_StayActive, TRUE,
      LISTBROWSER_Editable, TRUE,
      LISTBROWSER_EditTrigger, LBET_DELAYEDSECOND,
      TAG_DONE);

installDetails();

mainLayout = NewObject(LAYOUT_GetClass(), NULL,
        LAYOUT_Orientation, LAYOUT_ORIENT_HORIZ,
                   LAYOUT_DeferLayout, TRUE,
                   LAYOUT_SpaceInner, TRUE,
                   LAYOUT_SpaceOuter, TRUE,
                   LAYOUT_AddChild, treeBrowser,
                   CHILD_WeightedWidth, 40,
                   LAYOUT_AddChild, detailBrowser,
                   TAG_DONE);

  windowObject = NewObject(WINDOW_GetClass(), NULL,
                   WINDOW_Position, WPOS_CENTERSCREEN,
                   WA_Activate, TRUE,
                   WA_Title, "ListBrowser demo",
                   WA_DragBar, TRUE,
                   WA_CloseGadget, TRUE,
                   WA_DepthGadget, TRUE,
                   WA_SizeGadget, TRUE,
                   WA_InnerWidth, 500,
                   WA_InnerHeight, 150,
                   WINDOW_IDCMPHook,     &IDCMPHook,
                   WINDOW_IDCMPHookBits, IDCMP_IDCMPUPDATE,
                   WA_IDCMP, IDCMP_CLOSEWINDOW,
                   WINDOW_Layout, mainLayout,
                   TAG_DONE);

  if (! windowObject)
          cleanexit(NULL);

  if (! (intuiwin = (struct Window *) DoMethod(windowObject,WM_OPEN, NULL)))
          cleanexit(windowObject);

  processEvents(windowObject);

  DoMethod(windowObject, WM_CLOSE);
  cleanexit(windowObject);
}

void processEvents(Object *windowObject)
{
  ULONG windowsignal;
  ULONG receivedsignal;
  ULONG result;
  ULONG code;
  BOOL end = FALSE;

  GetAttr(WINDOW_SigMask, windowObject, &windowsignal);

  while (!end)
  {
          receivedsignal = Wait(windowsignal);
          while ((result = DoMethod(windowObject, WM_HANDLEINPUT, &code)) != WMHI_LASTMSG)
          {
                  switch (result & WMHI_CLASSMASK)
                  {
                          case WMHI_CLOSEWINDOW:
                             end=TRUE;
                             break;
                  }
          }
  }
}

void cleanexit(Object *windowObject)
{
  if (windowObject)
  {
     DisposeObject(windowObject);

     FreeLBColumnInfo(ci);
     FreeListBrowserList(&treeNodes);
     FreeListBrowserList(&detailNodes);
  }

  CloseLibrary((struct Library*)IntuitionBase);
  CloseLibrary(UtilityBase);
  CloseLibrary(WindowBase);
  CloseLibrary(LayoutBase);
  CloseLibrary(ListBrowserBase);
  exit(0);
}
```

Chapter 4
CLICKTAB GADGET

Chapter 5

TEXTEDITOR GADGET

The texteditor gadget is almost a full featured text editor. Just place it in a window and attach some scrollbars.

The gadget supports keyboard input like PageUp, PageDown, End and Home on keyboards that have these keys. It also supports scroll mouse in both vertical and horizontal directions. It supports cut and paste using the clipboard in IFF FTXT format. It supports an unlimited history of changes, with shortcuts Amiga+X for undo and Amiga+Y (or Amiga+Shift+X) for Redo automatically caught, and the change history can also be controlled programmatically.

The gadget can also show different text styles and colors, if you implement a syntax highlighter.

The gadget also supports search (showing all matches) and incremental search where the cursor can jump from one search hit to the next.

Chapter 9

CREATING A GADGET SUBCLASS WITH MODERN FEATURES

Sometimes you want to make your own gadget subclass. This was already described in the Amiga ROM Kernel Reference Manual: Libraries Third Edition. While the information from back then is still valid, many new topics need to be addressed. This chapter will start with a small recap, and will then move on to cover the following topics about your gadget subclass:

- Do its part in hierarchical layout.
- Support contextual help text.
- Support contextual mouse pointer.
- Support scroll wheel.
- Adapt to the visual theme chosen by the user.

Our initial recap on how to make a subclass will serve us as the base for introducing the new topics. At the end of the chapter we will present a full example.

Small Recap on Making a Gadget Class

A BOOPSI gadget is object oriented, hence the OOP in BOOPSI. This means that intuition and programs execute methods on the object. A method is defined by the class or its superclass(es).

When making your own class you need to expose the methods so that intuition and programs can execute them. Methods are not functions but rather intuition calls a so-called dispatch function of the class. One of the arguments to the dispatch function is the message, which instructs which method to be executed and what the arguments are.

```
static ULONG __ASM__ dispatch_function(__REG__(a0, Class *cl ),
                                       __REG__(a2, Object *o ),
                                       __REG__(a1, Msg msg) )
{
        if ( msg->MethodID == OM_NEW )
                return new_method(cl, o, (struct opSet *)msg);
        else
        {
                ULONG retval;

                switch( msg->MethodID )
                {
                        case GM_HANDLEINPUT:
                                retval = handleinput_method(cl, o, (struct gpInput *)msg);
                                break;

                        case OM_UPDATE:
                        case OM_SET:
                                retval = DoSuperMethodA(cl, o, (Msg)msg);
                                retval += set_method(cl, o, (struct opSet *)msg);
```

```
                              break;

                        case GM_RENDER:
                              retval = render_method(cl, o, (struct gpRender *)msg);
                              break;

                        case GM_LAYOUT:
                              retval = layout_method(cl, o, (struct gpLayout *)msg);
                              break;

                        case GM_GOACTIVE:
                              retval = goactive_method(cl, o, (struct gpInput *)msg);
                              break;

                        default:
                              retval = DoSuperMethodA(cl, o, msg);
                              break;
                  }

                  return retval;
            }

            return 0;
      }
```

You then create the class like this somewhere else:

```
      Class *cl = NULL;

      if (cl = MakeClass("demo.gadget", "gadgetclass", NULL, sizeof(struct InstData), 0))
      {
            cl->cl_Dispatcher.h_Entry = (HOOKFUNC)dispatch_function;
            cl->cl_Dispatcher.h_SubEntry = NULL;
            cl->cl_Dispatcher.h_Data = NULL;
      }
```

The InstData is a structure of your own design that contains all the variables that are private to each instance of your class. You can obviously call it anything you like.

THE METHODS REQUIRED IN RELEASE 2

So before we go into the new methods let us continue with our recap by going through the methods that were required in Release 2.

OM_NEW Method

The OM_NEW function is a bit special. The object argument is NULL, as the purpose is to create the object. Allocation is done by calling the superclass, which returns the object pointer, and then you can do your own setup.

```
static ULONG new_method(Class *cl, Object *o, struct opSet *ops)
{
        struct InstData *id;
        Object *object;

        if (object = (Object *)DoSuperMethodA(cl, o, (Msg)ops))
        {
                id = INST_DATA(cl, object);

                if (1) // in this simple example we are good now
                        return object;
                CoerceMethod(cl, object, OM_DISPOSE);
        }
        return NULL;
}
```

OM_DISPOSE Method

The OM_DISPOSE method should basically do the opposite of the OM_NEW. You should free and take down your own stuff and then call the superclass to free the object itself.

OM_SET and OM_GET Methods

The OM_SET and OM_GET methods are used to set and get attributes on your object. Once again it is important to also call the superclass, as GA_xxx attributes are generally handled by gadgetclass. The InstData struct discussed when creating the class is the obvious place to store the values being set, and where to fetch them, when giving them back in OM_GET.

GM_xxx Methods Related to Input

The GM_xxx methods pertain to the object being a gadget and not just a generic object. The main functions are GM_HITTEST, GM_GOACTIVE, GM_HANDLEINPUT and GM_GOINACTIVE. It basically goes like this. Intuition registers a mouse click, and calls GM_HITTEST to see which gadget was clicked. Then GM_GOACTIVE is sort of the first event handling, and depending on the return code many more GM_HANDLEINPUT are called with events until the gadget gives away its activation. A final call to GM_GOINACTIVE allows the gadget to clean up any state info.

GM_RENDER Method

This method actually draws the gadget.

The Window.class Layout Process

With the window.class and layout.gadget comes a whole new layout process. In the past the developer specified the exact pixel geometry of each gadget. But now the developer only specifies abstract spatial relationships. The layout process, translating those relationships into pixel geometry, is initiated by the window.class and carried out by the layout.gadget. The layout gadget recursively delegates the layout process to child gadgets.

If the user later changes the font, or resizes the window then the window.class will initiate a new layout process translating the abstract spatial relations into new and likely different pixel geometries.

Intuition is not aware of the child gadgets in a window. The window.class only maintains a single top level gadget that takes up the entire window (inside the window frame). This top level gadget has to be a layout.gadget, and the entire hierarchy of child gadgets are hidden from intuition.

> *Caution:* Even though child gadgets are in a hierarchy their pixel geometry are still described in window coordinates, and not relative to their parent gadget.

So Intuition is only ever aware of the top level layout gadget. Any method initiated by intuition goes to the top level layout, which in turn calls the child gadgets.

The layout process consists of 2 passes: Collecting size requirements, and then assigning pixel geometry.

First Pass: Collecting Size Requirement

The first pass is a depth first recursion. The minimum, nominal and maximum requirements of the deepest child gadget is queried. Those requirements then form the basis for the requirements of the gadget above the deepest. This continues all the way back up the recursion to the topmost layout.

The main purpose is to figure out how big the window should be. The layout.gadget will cache the information from its child gadgets as this information will be needed again in the next pass. For this collection a new method was introduced:

GM_DOMAIN Method

The GM_DOMAIN method is used to ask a gadget for its minimum, nominal and maximum sizes. Note that it is "domain" (meaning territory) and not "do main". It will be called several times on every gadget, for nominal, minimum and maximum respectively.

Often the child gadget will need to calculate these sizes based on its own child gadgets, or maybe the length of a string etc.

The message is of type **struct gmDomain**. The gpd_Which member tells you which of nominal, minimum, maximum the caller wants to hear about. The gpd_Domain is a struct IBox you should fill with values. Left and Top should just be zeros, and are not used for now.

For GDOMAIN_NOMINAL simply return the size that will fit all strings and padding around. This should be the minimum that looks good.

For GDOMAIN_MINIMUM you should return the minimum size where the gadget is still usable.

For GDOMAIN_MAXIMUM most of the gadgets in 3.2 will reply with 4000 as a sufficiently high number signalling that maximum can be any size you want, but with RTG and bigger and bigger screens that no longer sounds large enough. So if you don't care, then better return 0x7FFF (it is a signed WORD after all). If your gadget queries child gadgets take care not simply summing without checking if the sum is larger than 0x7FFF.

So first we need to add a new case ino the dispatch_function:

```
        case GM_DOMAIN:
                retval = domain_method(cl, o, (struct gpDomain *)msg);
                break;
```

And the implementation could look something like this:

```
static ULONG domain_method(Class *cl, Object *o, struct gpDomain *gpd)
{
        struct InstData *id = INST_DATA(cl, o);

        if (gpd->gpd_Which == GDOMAIN_MAXIMUM)
                gpd->gpd_Domain.Width = gpd->gpd_Domain.Height = 0x7FF; // as large as you want
        else
                gpd->gpd_Domain.Width = gpd->gpd_Domain.Height = 50;

        return 1;
}
```

Second Pass: Assigning Pixel Geometry

The second pass is top-down. The size and position is told to the topmost layout which in turn tells its child gadgets their size and position.

GM_LAYOUT Method

Placing gadgets using a method has the benefit that all are set in one go, where each gadget subclass can layout its children exactly as it wants. And the gadget can and should recursively set the position of any child gadgets.

The message is of type **struct gmLayout**.

> *Warning*: Your implementation of GM_LAYOUT should be prepared that GM_DOMAIN might not have been called. Intuition also calls this method for top level gadgets when windows are resized (if the gadget is GREL). To support that the gadget may be used like that, you should see how the example does it.

So we need to add a new case in the dispatch_function:

```
        case GM_LAYOUT:
                retval = layout_method(cl, o, (struct gpLayout *)msg);
                break;
```

And the layout function in our simple class is really simple, but if we had child gadgets we would need to call those as well. One thing to note is that our geometry is set before the layout method is called. You should do similar if you have child gadgets

```
static ULONG layout_method(Class *cl, Object *o, struct gpLayout *gpl)
{
        struct InstData *id = INST_DATA(cl, o);

        // Calc IBox just in case we are GREL_
        SetupGadgetIBox( o, &gpl->gpl_GInfo->gi_Domain, &id->Container);

        if (id->Container.Width == 0)
                return 0;

        id->Layouted = TRUE;

        return 1;
}
```

The function SetupGadgetIBox is a helper function (not shown here) that takes the gadget (possibly relative) geometry and turns it into absolute geometry. This way the gadget can also be used in a classic intuition setup with the GREL_ flags.

Contextual HelpText and Mouse pointer

Intuition introduced a GM_HELPTEST method way back in version 3.1 which is sort of similar to GM_HITTEST, but GM_HELPTEST is asked by intuition much more often. In fact every time the mouse is moved, but also in several other cases. Thus you will need to answer as quickly as possible.

GM_HELPTEST Method

You will only be called if the mouse is over your gadget, so unless you have child gadgets or an irregular outline you should just return GMR_HELPHIT.

Returning GMR_HELPHIT is exactly what the gadget superclass does already, so there is really no need to implement this method if that is all you want to do.

If your gadget has child gadgets you need to figure out which child gadget is hovered and call GM_HELPTEST on that child. You will also need to record which child was hit.

```
        case GM_HELPTEST:
                retval = helptest_method(cl, o, (struct gpHitTest *)msg);
                break;
```

In the following snippet, instead of looking at child gadgets, which our simple example doesn't have, we will simply see if the mouse is on the left or right half of the gadget.

```
static ULONG hittest_method(Class *cl, Object *o, struct gpHitTest *gpht)
{
        struct InstData *ourData = INST_DATA(cl, o);
        ourData->mouseOverRightPart = gpht->gpht_Mouse.X > G(o)->Width / 2;

        return GMH_HELPHIT;
}
```

> *Important*: The mouse coordinates are relative to the gadget, and if you are going to ask your child gadgets you would have to modify the mouse coordinates before asking the child (and restore them afterwards).

So now intuition and the hierarchy collectively knows which gadget was hit. Next you can expect intuition and window.class to ask you for help text, and mousepointer. However because intuition doesn't know the child of the child that was hit, intuition can't ask directly. Instead intuition will ask via a new method which was introduced in 3.2.

GM_QUERYHOVERED Method

The GM_QUERYHOVERED is sort of the same as OM_GET. It even reuses the opGet message struct. The gadget that doesn't care about which part was hit above can just ignore this method in the dispatcher. The gadget superclass will just transform this into an OM_GET and ask you.

In most cases you would simply let the superclass handle this method instead of returning specific hardcoded values. If you have child gadgets that are hovered you should recursively call the GM_QUERYHOVERED on the hovered child. For our small example, even though we don't have child gadgets, we do want to provide a different help tip and mouse pointer for the left and right side of our gadget.

So we need to add a new case in the dispatch_function:

```
        case GM_QUERYHOVERED:
                retval = queryhovered_method(cl, o, (struct opGet *)msg);
                break;
```

The function should be implemented with handling the specific attributes you want to handle and calling the superclass for any other attribute. The gadget class will call your OM_GET. So you should not call the set_method yourself directly. Let gadget class do that.

```
ULONG queryhovered_method(Class *cl, Object *o, struct opGet *opg)
{
   switch (opg->opg_AttrID)
   {
      case GA_GadgetHelpText:
         *opg->opg_Storage = ourData->mouseOverRightPart ? "right" : "left";
         return 1;
      case GA_CustomMousePointer:
         *opg->opg_Storage = ourData->mouseOverRightPart ? 0 : 1; // special for default and busy
         return 1;
      default:
         return DoSuperMethod(msg);
   }
}
```

Mouse Wheel Scrolling

If you want to support mouse wheel scrolling (and you should if you have child gadgets) then you must implement the GM_HANDLESCROLL method.

The layout gadget when called in turn calls the child under the mouse. You are expected to do the same if you have children.

In addition if your content can scroll don't forget to set the GMORE_SCROLLRASTER flag in the **MoreFlags** field in the **struct ExtGadget** (if you are a BOOPSI gadget then you are an extended gadget and can safely cast yourself).

GM_HANDLESCROLL Method

You should not assume the gadget is active when this method is called.

```
        case GM_HANDLESCROLL:
                retval = handlescroll_method(cl, o, (struct gpInput *)msg);
                break;
```

In our case we don't really do anything, but you should get an idea of how to do what you want in your own gadget.

```
ULONG handlescroll_method(Class *cl, Object *o, struct gpInput *gpi)
{
   struct InputEvent *ie = gpi->gpi_IEvent;
   struct instData *id = INST_DATA( cl, o );

   if (ie->ie_Class == IECLASS_RAWKEY)
   {
      switch (ie->ie_Code)
      {
      case NM_WHEEL_UP:
      case NM_WHEEL_LEFT:
         break;

      case NM_WHEEL_DOWN:
      case NM_WHEEL_RIGHT:
         break;

      default:
         return GMR_NOREUSE;
      }
   }
}
```

Adapting to Visual Theme

To adapt to different visual themes (Gadtools, thin, xen, etc) then you need to react to a few attributes being set: REACTION_SpecialPens and REACTION_ChangePrefs. Most often you simply want to set these attributes on any bevel.image that you are using to draw frames.

If you use bevel widths for anything, then it is a good time to ask for them again, after having received REACTION_ChangePrefs:

```
        SetAttrs(ourDatat->bevel, REACTION_ChangePrefs, tag->ti_Data, TAG_END);
        GetAttr(BEVEL_VertSize, id->bevel, &vert);
        GetAttr(BEVEL_HorizSize, id->bevel, &horiz);
```

You should not redraw or relayout yourself, as you will be asked to do so next.

If you simply draw using a bevel then you are almost good to go, but if you draw your own diagonal or rounded edges then you need to know the colour to draw with as well as which theme you are supposed to draw in.

The special pens needed to draw in the xen theme are defined in reaction/reaction_class.h

The preference is defined in reaction_reaction_prefs.h

A Complete Example

In this complete example we create the class and open a window with the gadget in it. We do not turn the gadget into a library that can be loaded. To learn how to make a library see the chapter dedicated to that subject.

Our gadget draws two bevels. One on its left side and one on its right side. So while it looks like two buttons it is in fact a single gadget. All for education.

```
#include <intuition/intuition.h>
#include <intuition/classes.h>
#include <intuition/classusr.h>
#include <intuition/gadgetclass.h>
#include <intuition/icclass.h>
#include <intuition/pointerclass.h>

#include <devices/inputevent.h>

#include <utility/tagitem.h>

#include <clib/alib_protos.h>

#include <proto/exec.h>
#include <proto/graphics.h>
#include <proto/intuition.h>
#include <proto/window.h>
#include <proto/layout.h>
#include <proto/bevel.h>
#include <proto/utility.h>


struct InstData
{
   ULONG mouseOverRightPart;
   ULONG midWidth;
   Object *bevel;
};

#define G(o)((struct Gadget *)(o))
#define IM(o) ((struct Image *)(o))
```

```
struct Library *WindowBase;
struct Library *LayoutBase;
struct Library *BevelBase;

static ULONG set_method(Class *cl, Object *o, struct opSet *ops);

static ULONG dorender(BOOL redraw, Object *o, struct InstData *id, struct GadgetInfo *gi)
{
   struct RastPort *rp = ObtainGIRPort(gi);

   if (rp)
   {
      DoMethod(o, GM_RENDER, (ULONG) gi, (ULONG) rp, redraw );

      ReleaseGIRPort(rp);
      return 1;
   }
   return 0;
}

/***********************************************************************/


static ULONG new_method(Class *cl, Object *o, struct opSet *ops)
{
   struct InstData *ourData;
   Object *object;

   if (object = (Object *)DoSuperMethodA(cl, o, (Msg)ops))
   {
      ourData = INST_DATA(cl, object);

      ourData->bevel = NewObject(BEVEL_GetClass(), NULL,
            BEVEL_Style, BVS_BUTTON,
            BEVEL_Flags, BFLG_XENFILL,
            BEVEL_Transparent, FALSE,
            TAG_DONE);

      if (ourData->bevel) // in this simple example we are good now
         return (ULONG) object;

      DisposeObject(ourData->bevel);

      CoerceMethod(cl, object, OM_DISPOSE);
   }
   return NULL;
}

static ULONG dispose_method(Class *cl, Object *o, Msg msg)
{
   struct InstData *ourData = INST_DATA(cl, o);

   DisposeObject(ourData->bevel);

   return DoSuperMethodA(cl, o, msg);
}

static ULONG set_method(Class *cl, Object *o, struct opSet *ops)
{
   struct InstData *ourData = INST_DATA(cl, o);
   struct TagItem *tstate = ops->ops_AttrList;
    struct TagItem *tag;
    ULONG  tidata;
   BOOL refresh = FALSE;

   while (tag = NextTagItem(&tstate))
   {
      switch (tag->ti_Tag)
      {
         case REACTION_SpecialPens:
            SetAttrs(ourData->bevel, REACTION_SpecialPens, tag->ti_Data, TAG_END);
            break;

         case REACTION_ChangePrefs:
            SetAttrs(ourData->bevel, REACTION_ChangePrefs, tag->ti_Data, TAG_END);
            break;

         case REACTION_TextAttr:
            break;
      }
   }

   if (ops->MethodID != OM_NEW)
   {
      if (ops->ops_GInfo && OCLASS(o) == cl)
      {
         dorender(refresh, o, ourData, ops->ops_GInfo);
         return 0;
      }
   }

   return 1;
}

static ULONG domain_method(Class *cl, Object *o, struct gpDomain *gpd)
{
   if (gpd->gpd_Which == GDOMAIN_MAXIMUM)
      gpd->gpd_Domain.Width = gpd->gpd_Domain.Height = 4096; // nice high number
   else
      gpd->gpd_Domain.Width = gpd->gpd_Domain.Height = 150;

   return 1;
}


static ULONG render_method(Class *cl, Object *o, struct gpRender *gpr)
{
   struct InstData *ourData = INST_DATA(cl, o);

   DrawImageState(gpr->gpr_RPort, IM(ourData->bevel),
         G(o)->LeftEdge, G(o)->TopEdge, IDS_NORMAL, gpr->gpr_GInfo->gi_DrInfo);
   DrawImageState(gpr->gpr_RPort, IM(ourData->bevel),
         G(o)->LeftEdge + ourData->midWidth, G(o)->TopEdge, IDS_NORMAL, gpr->gpr_GInfo->gi_DrInfo);

   return 1;
}

static ULONG goactive_method(Class *cl, Object *o, struct gpInput *gpi)
{
   struct InstData *ourData = INST_DATA(cl, o);
   ULONG retval = GMR_NOREUSE;

   if (!(G(o)->Flags & GFLG_DISABLED))
   {
      DoSuperMethodA(cl, o, (Msg)gpi);

      dorender(1, o, ourData, gpi->gpi_GInfo);
      retval = GMR_MEACTIVE;
   }

   return retval;
}

static ULONG handleinput_method(Class *cl, Object *o, struct gpInput *gpi)
{
   struct InputEvent *ievent = gpi->gpi_IEvent;
   ULONG retval = GMR_MEACTIVE;

   DoSuperMethodA(cl, o, (Msg)gpi);

   if (ievent->ie_Class == IECLASS_RAWMOUSE && ievent->ie_Code == SELECTUP)
```

```
      retval = GMR_NOREUSE;

   return retval;
}

ULONG helptest_method(Class *cl, Object *o, struct gpHitTest *gpht)
{
   struct InstData *ourData = INST_DATA(cl, o);
   ourData->mouseOverRightPart = gpht->gpht_Mouse.X > ourData->midWidth;;

   return GMR_HELPHIT;
}

ULONG queryhovered_method(Class *cl, Object *o, struct opGet *opg)
{
   struct InstData *ourData = INST_DATA(cl, o);

   switch (opg->opg_AttrID)
   {
      case GA_GadgetHelpText:
         *opg->opg_Storage = (ULONG)(ourData->mouseOverRightPart ? "right" : "left");
         return 1;
      case GA_CustomMousePointer:
         *opg->opg_Storage = ourData->mouseOverRightPart ? 0 : 1; // special for default and busy
         return 1;
      default:
         return DoSuperMethodA(cl, o, (Msg)opg);
   }
}

static ULONG layout_method(Class *cl, Object *o, struct gpLayout *gpl)
{
   struct InstData *ourData = INST_DATA(cl, o);

   ourData->midWidth = G(o)->Width / 2;
   IM(ourData->bevel)->Width = ourData->midWidth;
   IM(ourData->bevel)->Height = G(o)->Height;

   return 1;
}

static ULONG __ASM__ __SAVE_DS__ dispatch_function(__REG__(a0, Class *cl ),
               __REG__(a2, Object *o ),
               __REG__(a1, Msg msg) )
{
   if ( msg->MethodID == OM_NEW )
      return new_method(cl, o, (struct opSet *)msg);
   else
   {
      ULONG retval;

      switch( msg->MethodID )
      {
         case GM_HELPTEST:
            retval = helptest_method(cl, o, (struct gpHitTest *)msg);
            break;

         case GM_QUERYHOVERED:
            retval = queryhovered_method(cl, o, (struct opGet *)msg);
            break;

         case GM_HANDLEINPUT:
            retval = handleinput_method(cl, o, (struct gpInput *)msg);
            break;

         case OM_UPDATE:
         case OM_SET:
            retval = DoSuperMethodA(cl, o, (Msg)msg);
            retval += set_method(cl, o, (struct opSet *)msg);
            break;
         case GM_RENDER:
            retval = render_method(cl, o, (struct gpRender *)msg);
            break;

         case GM_LAYOUT:
            retval = layout_method(cl, o, (struct gpLayout *)msg);
            break;

         case GM_GOACTIVE:
            retval = goactive_method(cl, o, (struct gpInput *)msg);
            break;

         case GM_DOMAIN:
            retval = domain_method(cl, o, (struct gpDomain *)msg);
            break;

         case OM_DISPOSE:
            retval = dispose_method(cl, o, msg);
            break;

         default:
            retval = DoSuperMethodA(cl, o, msg);
            break;
      }

      return retval;
   }

   return 0;
}

struct HintInfo gb_HintInfo[] = {{ -1, -1, NULL, 0 }}; /* needed for tooltips even if empty */

int main(void)
{
   struct Window *intuiwin = NULL;
   Object *windowObject = NULL;
   Class *cl = NULL;

   // To make the code easier to understand we skip many checks that
   // should be there in production code

   IntuitionBase = (struct IntuitionBase*)OpenLibrary("intuition.library", 47);
   UtilityBase = OpenLibrary("utility.library",47);
   WindowBase = OpenLibrary("window.class", 47);
   LayoutBase = OpenLibrary("gadgets/layout.gadget", 47);
   BevelBase = OpenLibrary("images/bevel.image", 47);

   if (cl = MakeClass("demo.gadget", "gadgetclass", NULL, sizeof(struct InstData), 0))
   {
      cl->cl_Dispatcher.h_Entry = (HOOKFUNC)dispatch_function;
      cl->cl_Dispatcher.h_SubEntry = NULL;
      cl->cl_Dispatcher.h_Data = NULL;

      windowObject = NewObject(WINDOW_GetClass(), NULL,
         WA_Activate, TRUE,
         WA_Title, "Custom Gadget Demo",
         WA_DragBar, TRUE,
         WA_CloseGadget, TRUE,
         WA_SizeGadget, TRUE,
         WINDOW_HintInfo, gb_HintInfo,
         WINDOW_GadgetHelp, TRUE,
         WA_IDCMP, IDCMP_CLOSEWINDOW,
         WINDOW_Layout, NewObject(LAYOUT_GetClass(), NULL,
            LAYOUT_Orientation, LAYOUT_ORIENT_VERT,
            LAYOUT_DeferLayout, TRUE,
            LAYOUT_AddChild, NewObject(cl, NULL, TAG_DONE),
            TAG_DONE),
          TAG_DONE);
   }
   if (windowObject)
   {
      if (intuiwin = (struct Window *) DoMethod(windowObject,WM_OPEN, NULL))
      {
         ULONG windowsignal;
         ULONG receivedsignal;
         ULONG result;
         ULONG code;
         BOOL end = FALSE;

         GetAttr(WINDOW_SigMask, windowObject, &windowsignal);

         while (!end)
         {
            receivedsignal = Wait(windowsignal);
            while ((result = DoMethod(windowObject, WM_HANDLEINPUT, &code)) != WMHI_LASTMSG)
            {
               switch (result & WMHI_CLASSMASK)
               {
                  case WMHI_CLOSEWINDOW:
                  end=TRUE;
                  break;
               }
            }
         }

         DoMethod(windowObject, WM_CLOSE);
      }
      DisposeObject(windowObject);
   }

   if (cl)
      FreeClass(cl);

   CloseLibrary((struct Library*)IntuitionBase);
   CloseLibrary(BevelBase);
   CloseLibrary(UtilityBase);
   CloseLibrary(WindowBase);
   CloseLibrary(LayoutBase);
}
```

Chapter 10
SCREENS

The Amiga Screen concept has been extended under Release 3 to accommodate new chipsets and screen modes as well as new functionality such as pen sharing, backfill hooks, double buffering and attached screens.

There have also been some changes under the hood. In particular, AmigaOS 3.1.4 introduced an optional version of Intuition that allows windows to be dragged off-screen. AmigaOS 3.2 consolidates this by having the new version of Intuition installed by default. Intuition Screens can automatically take advantage of this feature.

Screens are still opened and closed using the same methods that were in use under Release 2. That is, they are opened using the **OpenScreenTagList()** or the stack-based **OpenScreenTags()** functions and closed with the **CloseScreen()** function. The extensible screen opening functions allow for new functionality to be added primarily using tags that are ignored in earlier versions of the operating system which allows for easier backward compatibility.

For a complete description of Intuition Screens, please refer to Amiga ROM Kernel Reference Manual: Libraries Third Edition. The remainder of the Screens chapter will focus on new features that were implemented since that book was published back in 1991.

New Screen Features

NEW AND MODIFIED SCREEN ATTRIBUTE TAGS

SA_DisplayID (V36)

There is no change to the tag, but the display mode ID names are found in <graphics/modeid.h> from V39.

SA_ErrorCode (V36)

From **V39**, there are two more error codes that can be returned.

- **OSERR_TOODEEP** - Screen is too deep to be displayed on this hardware.
- **OSERR_ATTACHFAIL** - An illegal attachment of screens was requested.

From **V47**, one more error code can be returned.

- **OSERR_NORTGBITMAP** - The RTG software in use was unable to allocate a bitmap for the screen.

SA_Parent, SA_FrontChild, SA_BackChild (V39)

Release 3 supports attached screens, where one or more child screens can be associated with a parent screen.

Attached screens depth-arrange as a group, and always remain adjacent depth-wise. Independent depth-arrangement of child screens is possible through the **ScreenDepth()** call.

If a child screen is made non-draggable by setting **SA_Draggable** to **FALSE**, then it will drag exclusively with the parent. Normal child screens drag independently of the parent, but are pulled down when the parent is.

Use **SA_Parent** if you wish to attach this screen to a parent screen that has already been opened by setting **SA_Parent** to point to the parent screen.

Use **SA_FrontChild** to attach a child screen, that has already been opened, to this screen by setting **SA_FrontChild** to point to the child screen. The child screen will come to the front of the family defined by the parent screen you are opening.

Use **SA_BackChild** to attach a child screen, that has already been opened, to this screen by setting **SA_BackChild** to point to the child screen. The child screen will go to the back of the family defined by the parent screen you are opening.

SA_BackFill (V39)

A pointer to a backfill hook for the screen's Layer_Info. See ***Screen Backfill Hooks*** above for further details.

SA_Draggable (V39)

Boolean which defaults to **TRUE**.

Set to **FALSE** if you want your screen to be non-draggable. This tag should be used very sparingly!

For child screens this tag has a slightly different meaning: non-draggable child screens are non-draggable with respect to their parent, meaning they always drag exactly with the parent, as opposed to having relative freedom.

SA_Exclusive (V39)

Boolean which defaults to **FALSE**.

Set to **TRUE** if you never want your screen to share the display with another screen. See ***Exclusive Screens*** above. This means that your screen can't be pulled down, and will not appear behind other screens that are pulled down. Your screen may still be depth-arranged, though. Use this tag sparingly!

Starting with **V40**, attached screens may be **SA_Exclusive**. Setting **SA_Exclusive** for each screen will produce an exclusive family.

SA_SharePens (V39)

Boolean which defaults to **FALSE**.

For those pens in the screen's DrawInfo->dri_Pens, Intuition obtains them in shared mode. See ***Pen Sharing*** above. For compatibility, Intuition obtains the other pens of a public screen as **PENF_EXCLUSIVE**. Screens that wish to manage the pens themselves should generally set this tag to **TRUE**. This instructs Intuition to leave the other pens unallocated.

SA_Colors32 (V39)

Used to set the screen's initial palette colours to 32 bits-per-gun by setting **SA_Colors32** to point to a series of records, each with the following format:

1 Word with the number of colours to load.
1 Word with the first colour to be loaded.
3 Longwords representing a left justified 32-bit RGB triplet.

The list is terminated with the next count value being 0.

For example, the table below will load colour register 0 with 100% red:

```
ULONG table[]={1l<<16+0,0xffffffff,0,0,0};
```

This format supports both runs of colour registers and sparse registers. Any colour
set
here has precedence over the same register set by **SA_Colors**.

SA_Interleaved (V39)

Boolean which defaults to **FALSE**.

Set to **TRUE** to request an interleaved bitmap for your screen. See ***Interleaved Screens*** above. If the system cannot allocate an interleaved bitmap for you, it will attempt to allocate a non-interleaved one.

SA_VideoControl (V39)

Points to a taglist that will be passed to **VideoControl()** after your screen is open. You might use this to turn on border-sprites, for example.

SA_ColorMapEntries (V39)

Defaults to 1<<depth, or 32. Whichever is greater.

The number of entries that you wish Intuition to allocate for this screen's **ColorMap**. While Intuition allocates a suitable number for ordinary use, certain graphics.library features require a **ColorMap** which is larger than the default.

SA_LikeWorkbench (V39)

Boolean which defaults to **FALSE**.

Set to **TRUE** to get a screen just like the Workbench screen. This is the best way to inherit all the characteristics of the Workbench, including depth, colours, pen-array, screen mode, etc. Individual attributes can be overridden through the use of tags. (**SA_LikeWorkbench** itself overrides things specified in the **NewScreen** structure). Attention should be paid to hidden assumptions when doing this. For example, setting the depth to two makes assumptions about the pen values in the **DrawInfo** pens. Note that this tag requests that Intuition ATTEMPT to open the screen to match the Workbench. There are fallbacks in case that fails, so it is not correct to make enquiries about the Workbench screen then make strong assumptions about what you're going to get.

SA_MinimizeISG (V40)

Boolean which defaults to **FALSE**.

For compatibility, Intuition always ensures that the inter-screen gap is at least 3 non-interlaced lines. If your application would look best with the smallest possible inter-screen gap, set SA_MinimizeISG to TRUE.

If you use the new graphics **VideoControl() VC_NoColorPaletteLoad** tag for your screen's **ViewPort**, you should also set this tag.

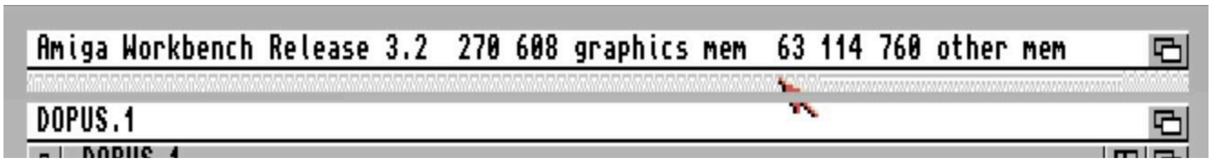
SA_OffscreenDragging (V45)

Boolean which defaults to FALSE for custom screens, and the value set in IControl Preferences for the Workbench screen.

If this tag is set, windows may be partially dragged out of the screen, with graphics being clipped at screen edges.

INTER SCREEN GAPS

In order to display more than one screen at once on a display the Amiga takes advantage of its hardware co-processor (the COPPER) to set up the colour and resolution differences between two screens as can be seen in the image below.


Amiga Workbench Release 3.2 270 608 graphics mem 63 114 760 other mem
DOPUS.1

There are some grey lines just before the front screen (DOPUS.1) is shown in front of the Workbench. This is called the Inter Screen Gap (ISG) and is an area that shows no image on the display in order to give the COPPER time to change resolution and prepare colours etc. Even the pointer won't display at that time.

With the AGA chipset there is also a potential for many different colours and resolutions which can take a few lines (6 or so) to calculate, leaving a larger ISG. You can ask Intuition to do its best to minimise the ISG in situations that might not need so much time to calculate. To do this, pass the **SA_MinimizeISG** tag to the **OpenScreenTagList()** call. If Intuition thinks it can perform the calculations without so much blank space it will try to do that, however this is not a guarantee. This tag is available from AmigaOS 3.0.

ATTACHED SCREENS

Screens can now have a parent-child relationship in that child screens can be attached to a parent screen in order to present a more cohesive interface with multiple potential resolutions and colour palettes. This is described in more detail in the Attached Screens chapter below.

DRAWINFO VERSIONS 2 AND 3

The screen DrawInfo structure has been extended in AmigaOS 3.0 (v39) to version 2, and again in AmigaOS 3.2 (v47) to version 3.

DrawInfo is used to assist applications and BOOPSI gadgets to understand the drawing environment they may wish to render into. It is obtained by calling the **GetScreenDrawInfo()** function, and released by calling the **FreeScreenDrawInfo()** function below. You should only access the DrawInfo pointer between the **GetScreenDrawInfo()** call and the **FreeScreenDrawInfo()** call.

Locking of the screen is not performed by those functions so to ensure the screen does not get closed while you are examining the DrawInfo structure the screen should already have been locked - either by having a window open on the screen or having performed a **LockPubScreen()** on it.

An excerpt showing how to retrieve the screen’s default font from a DrawInfo follows:

```
struct Screen *src;
struct DrawInfo *di;
struct TextFont *font;

scr = OpenScreenTags(NULL, TAG_END);
if(scr != NULL)
{
    di = GetScreenDrawInfo(scr);
    if(di != NULL)
    {
        font = di->dri_Font;

        FreeScreenDrawInfo(scr, di);
    }
    CloseScreen(scr);
}
```

The changes in AmigaOS 3.0 bump the DRI_VERSION to 2 and has been extended to include three new pens as well as a pointer to the Checkmark and Amiga Key images. The included fields are described in more detail below:

dri_CheckMark

The dri_CheckMark field points to the Checkmark image that is used in menus to indicate selected menu items.

dri_AmigaKey

The dri_AmigaKey field points to the Amiga Key image that is used in menus to indicate that a menu item can also be selected by pressing the right-Amiga key combined with a particular character.

dri_Pens

One of the most useful fields in the DrawInfo structure is the dri_Pens[] array of pens that are used for rendering the user interface elements of the screen.

Under v37 the following pens were defined:

DETAILPEN	1.x compatible text drawing pen.
BLOCKPEN	1.x compatible background drawing pen.
TEXTPEN	Pen for rendering text.
SHINEPEN	Pen for drawing the bright edges of objects.
SHADOWPEN	Pen for drawing the dark edges of objects.
FILLPEN	Pen used for filling an active window or selected gadget.
FILLTEXTPEN	Pen for drawing text over FILLPEN.
BACKGROUNDPEN	Background pen. Usually colour 0.
HIGHLIGHTTEXTPEN	Pen for drawing highlighted text.

Under Release 2 the screen title bars and menus were rendered in 1.x compatible pens. Where under 1.x this was a white background with blue text, this became a black background with grey text under Release 2 due to pen limitations.

Release 3 allows specific pens to be defined for drawing the title bar and menu imagery and now includes a trim colour at the bottom of the menu. The pens to support this were added in v39 and are described below:

BARDETAILPEN	Pen for drawing the text and details in the screen bar/menu.
BARBLOCKPEN	Pen for filling the screen bar/menu background.
BARTRIMPEN	Pen for drawing the trim under the screen bar/menus.
BARCONTOURPEN	Pen for drawing the contour above the screen title bar and menus (V47)

Finally, v39 adds some helpful definitions for finding the complements of the first four colours in a screen. They are:

PEN_C3	Complement of pen 3.
PEN_C2	Complement of pen 2.
PEN_C1	Complement of pen 1.
PEN_C0	Complement of pen 0.

Though these macros are defined in v39 and above of the includes, they can be used with earlier versions of the Operating System.

dri_Screen

If using V47 or later, you can also get a pointer to the associated screen from this position. This can come in handy when rendering BOOPSI/ReAction gadgets.

SCREEN CONTROL

A new function to allow more control over screen depth arrangement has been added. **ScreenDepth()** replaces the functionality of both **ScreenToBack()** and **ScreenToFront()** while adding some extensible new features, primarily to support the new Release 3 Attached Screens functionality.

Using **ScreenDepth()** as described below will work in the same way as calling **ScreenToBack()**:

```
ScreenDepth(screen, SDEPTH_TOBACK, NULL);
```

Similarly, to have **ScreenDepth()** behave the same way as **ScreenToFront()**, call the function as follows:

```
ScreenDepth(screen, SDEPTH_TOFRONT, NULL);
```

The two examples above allow the screen to be sent to the front or back of all screens. If you want to only send the screen to the back or front of a group of attached screens of which it is a member, you need to add the **SDEPTH_INFAMILY** flag to the flags parameter.
For this to work, the screen must be a member of a family of attached screens.

To send to the back of the family screen group:

```
ScreenDepth(screen, SDEPTH_TOBACK | SDEPTH_INFAMILY, NULL);
```

To send to the front of the family screen group:

```
ScreenDepth(screen, SDEPTH_TOFRONT | SDEPTH_INFAMILY, NULL);
```

The third parameter to the **ScreenDepth()** function call is always NULL for now.

The **MoveScreen()** function has been supplemented with a new function called **ScreenPosition()**.

Whilst **MoveScreen()** allows you to move the screen up and down (or left and right where possible) by a number of lines from the current position, **ScreenPosition()** provides greater control over the movement.

The extra control comes from what is placed in the flags field. Setting the flags field to **SPOS_RELATIVE** will cause **ScreenPosition()** to behave in the same manner as **MoveScreen()**. The flags and how they are used are described below:

SPOS_RELATIVE	Moves the screen by the number of pixels specified in x1 and y1. The x2 and y2 fields are ignored for this mode.
SPOS_ABSOLUTE	Moves the screen to the location specified by x1 and y2. The x2 and y2 fields are ignored for this mode.

SPOS_MAKEVISIBLE	Specifies a location on screen that needs to be visible. This is handy for displays that are wider than the display clip and allows for the screen to scroll, for example, where text is being typed. The area to be made visible is described using x1/y1/x2/y2 as the bounding box, where:
	x1 = left
	y1 = top
	x2 = right
	y2 = bottom

SPOS_FORCEDRAG	The SPOS_FORCEDRAG flag does not specify a mode but instead can be used with any of the three modes above to force a screen to be dragged which explicitly disabled dragging (i.e. **SA_Draggable** = FALSE) when opened. **This flag should only be used on screens you opened and not on any screen that you do not explicitly own.**

OFF-SCREEN DRAGGING

Intuition v45 and higher now support the ability to drag windows off-screen.

When creating your screen, you can enable this by setting **SA_OffscreenDragging** to TRUE. By default, this will be disabled so it must be explicitly enabled.

The Workbench screen is slightly different in that the IControl preference tool is configured by the user to specify whether **SA_OffscreenDragging** is enabled or not.

AGA DISPLAY MODES

The AGA chipset contains many more modes, can make use of up to 8 bit planes and can now also be interleaved. Asking for the specific screen modes is not the best way to obtain the AGA features. Instead, it is best to ask for the individual features you want and allow the OS to provide you the screen mode that best fits both your requirements and the user's display (which may be RTG).

You ask for the best match mode ID for your requirements by using the **BestModeID()** function from the Graphics library. By using this function to bid for features you will not, for example, lock yourself into a PAL screen when an equivalent RTG screen might be available.

For instance, say I would like a High Resolution screen of 640x480 with 256 colours. *This kind of request would require either an Amiga with the AGA chipset or an RTG graphics card that can support those parameters.*

To request this you would call **BestModeID()** as follows:

```
ULONG myScreenId = BestModeID(BIDTAG_NominalWidth, 640,
                              BIDTAG_NominalHeight, 480,
                             BIDTAG_Depth, 8,
                             TAG_END);
```

If the Amiga you run this on does not have a mode capable of displaying the screen you have specified, **BestModeID()** will return **INVALID_ID**. Otherwise the screen mode that best represents your request will be returned. The returned mode id can then be passed as the **SA_DisplayID** parameter to the **OpenScreenTags()** call when opening the screen.

If, for example, a user's machine has both AGA and RTG but you want to be sure that this screen opens in an AGA screen mode rather than one on the RTG card, then you can also pass **BIDTAG_MonitorID** in to the **BestModeID()** function with the monitor you want to restrict the screen to (in this example, you would pass **VGA_MONITOR_ID**). As an alternative, you could also pass in an existing **ViewPort** of a screen that is open on the type of monitor you would also like your screen to open on via the **BIDTAG_ViewPort** tag. The Workbench's **ViewPort** is often the best option for this.

See the graphics.library autodoc for more information on the **BestModeID()** tags.

Below is an example of obtaining a 640x480 display containing 256 colours using **BestModeID()** to obtain a compatible mode. The Workbench is used as a reference **ViewPort** so the newly created screen is on the same display type as the workbench.

```
#include <exec/types.h>
#include <exec/libraries.h>
#include <dos/dos.h>

#include <intuition/screens.h>

#include <proto/exec.h>
#include <proto/graphics.h>
#include <proto/intuition.h>

struct Screen *scr;
struct Window *win;
WORD pens[] = {-1}; /* Use default pens */
UWORD xPos, yPos;
UWORD boxWidth = 25;
UWORD boxHeight = 25;

int main(int argc, char *argv[])
{
    IntuitionBase = (struct IntuitionBase *)OpenLibrary("intuition.library", 39);
    if(IntuitionBase != NULL)
    {
        GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 39);
        if(GfxBase != NULL)
        {
            ULONG modeId;

            /* Request the best possible 640x480 sized display containing 16 colours */
            modeId = BestModeID(BIDTAG_NominalWidth, 640,  /* Request a screen mode that 
best
                                                              suits a width of 640 pixels, 
*/
                                BIDTAG_NominalHeight, 480, /* and a height of 480 pixels, 
*/
                                BIDTAG_Depth, 8,           /* and displaying 256 colours. 
*/
                                TAG_END);

            scr = OpenScreenTags(NULL,
                                 SA_DisplayID, modeId, /* Supply the mode obtained above */
                                 SA_Width, 640,
                                 SA_Height, 480,
                                 SA_Depth, 8, /* 256 colours. Needs AGA or RTG. */
                                 SA_Title, "BestModeID Example Screen",
                                 SA_Type, CUSTOMSCREEN,
                                 SA_Pens, pens,
                                 TAG_END);
            if(scr)
            {
                win = OpenWindowTags(NULL,
                                     WA_CustomScreen, scr,
                                     WA_Left, 50,
                                     WA_Top, 50,
                                     WA_IDCMP, IDCMP_CLOSEWINDOW,
                                     WA_Title, "BestModeID Example Window (256 colours)",
                                     WA_DragBar, TRUE,
                                     WA_CloseGadget, TRUE,
                                     WA_NoCareRefresh, TRUE,
                                     WA_Activate, TRUE,
                                     WA_SmartRefresh, TRUE,
                                     WA_InnerWidth, boxWidth*16,
                                     WA_InnerHeight, boxHeight*16,
                                 TAG_END);

                if(win)
                {
                    int i,j;
                    BOOL done = FALSE;
                    struct IntuiMessage *iMsg;
                    ULONG sigWait = (1L << win->UserPort->mp_SigBit);
                    ULONG sigRcvd;

                    /* Draw 256 coloured boxes. */
                    for(i=0,yPos=win->BorderTop; i<16; ++i,yPos+=boxHeight)
                    {
                        for(j=0,xPos=win->BorderLeft; j<16; ++j,xPos+=boxWidth)
                        {
                            SetAPen(win->RPort, (i*16)+j);
                            RectFill(win->RPort,
                                     xPos, yPos,
                                     xPos+boxWidth-1, yPos+boxHeight-1);
                        }
                    }

                    /* Message Loop. We only care if the window close gadget was pressed.
*/
                    do
                    {
                        sigRcvd = Wait(sigWait);
                        while((iMsg = (struct IntuiMessage *)GetMsg(win->UserPort)) !=
NULL)
                        {
                            if(sigRcvd & sigWait)
                            {
                                switch(iMsg->Class)
                                {
                                    case IDCMP_CLOSEWINDOW:
                                        done = TRUE;
                                        break;
                                }
                            }

                           ReplyMsg((struct Message *)iMsg); /* Reply all IntuiMessages */
                        }
                    } while(!done);

                    CloseWindow(win);
                }
                CloseScreen(scr);
            }
            CloseLibrary((struct Library *)GfxBase);
        }
        CloseLibrary((struct Library *)IntuitionBase);
    }
    return RETURN_OK;
}
```

SCREEN BACKFILL HOOKS

Screens now support their own backfill pattern hooks.

This is supported by creating a **Hook** and supplying it to the **SA_BackFill** tag when calling **OpenScreenTagList()**.

The **Hook** will be called with the same parameters as those used by **InstallLayerInfoHook()** in **layers.library**, so your hook function needs to have the following signature:

```
struct bfMsg {APTR priv, struct Rectangle *bounds};
ULONG HookFunc(struct Hook *hook, struct RastPort *rport, struct LayerBackfillMsg *bfmsg);
```

The hook is called with the hook parameter supplied in register **A0**, the rport parameter supplied in register **A1**, and the bfmsg parameter supplied in register **A2**.

When your hook is called, you will be supplied a **RastPort**, a private field that you should ignore, and a **Rectangle** describing the bounds for your rendering.

It is important to remember that your hook function should restrict itself to bitmap rendering (e.g. **BltBitMap()**) and **avoid any operation that uses layers**. So, if you need to call a rendering operation that uses a **RastPort**, make a copy of the **RastPort** passed to your hook and set the **Layer** pointer to **NULL**.

Below is an example of a screen backfill hook.

```
#include <exec/types.h>
#include <exec/libraries.h>
#include <dos/dos.h>
#include <graphics/clip.h> /* for LayerMsg */
#include <string.h> /* for memcpy */

#include <intuition/screens.h>

#include <clib/alib_protos.h> /* HookEntry is here */
#include <proto/exec.h>
#include <proto/graphics.h>
#include <proto/intuition.h>

struct Screen *scr;
struct Window *win;
WORD pens[] = {-1}; /* Use default pens */
struct Hook hook;
struct BitMap *bgBitMap;

/* 16x16x2 bitmap pattern for filling */
UWORD bgData[2][16] =
{
    /* Plane 0 */
    {0x0001, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7E7F, 0x7C3F, 0x7C3F, 0x7E7F,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF},
    /* Plane 1 */
    {0xFFFE, 0xFFFE, 0xFFFE, 0xFFFE, 0xFFFE, 0xFFFE, 0xFE7E, 0xFC3E, 0xFC3E, 0xFE7E,
0xFFFE, 0xFFFE, 0xFFFE, 0xFFFE, 0xFFFE, 0x8000}
};

/* Preferably get the structure from the includes, but it was added very recently. */
#ifndef LayerBackfillMsg
```

```
struct LayerBackfillMsg
{
    struct Layer    *lbfm_Layer;
    struct Rectangle lbfm_Bounds;
    LONG             lbfm_OffsetX;
    LONG             lbfm_OffsetY;
};
#endif

/* This simple function is used to initialize a Hook */
VOID InitHook(struct Hook *h, ULONG (*func)(), VOID *data)
{
    /* Make sure a pointer was passed */
    if (h)
    {
        /* Fill in the Hook fields */
        h->h_Entry = (ULONG (*)())HookEntry; /* See HookEntry autodoc for details */
        h->h_SubEntry = func;
        h->h_Data = data;
    }
}

/* Backfill hook that is called for refreshing the screen */
ULONG scrBackFillFunc(struct Hook *hook, struct RastPort *rp, struct LayerMsg *lm)
{
    struct BitMap *bm = rp->BitMap;

    LONG left, top, width, height;
    LONG xPos, yPos; /* Relative to screen */

    /* Note that this is compiled with 32-bit pointers (sc option ABSFUNCPOINTER), not the
       16-bit offsets of the SMALL link model. This is to make the demonstration program
       simpler. Remember that this code is not executing on your program's context so set
up
       your environment appropriately if you use the 16-bit offset (SMALL) model or else
you
       will likely see mysterious crashes. See the documentation of geta4() or __SAVEDS for
       your compiler. */

    /* Define the bounds where actual rendering will take place */
    left = lm->lbfm_Bounds.MinX;
    top = lm->lbfm_Bounds.MinY;
    width = lm->lbfm_Bounds.MaxX - lm->lbfm_Bounds.MinX + 1;
    height = lm->lbfm_Bounds.MaxY - lm->lbfm_Bounds.MinY + 1;
    xPos = yPos = 0;

    /* NOTE: Feel free to optimize! :-) */
    for(;;)
    {
        LONG leftClip, rightClip, topClip, bottomClip;

        /* Are we in the vertical bounds? */
        if((yPos < top+height) && (yPos+16 >= top))
        {
            /* Yes! Are we in the horizontal bounds? */
            if((xPos < left+width) && (xPos+16 >= left))
            {
                /* Yes! Let's clip then render. */
                leftClip = (xPos < left) ? left-xPos : 0;
                rightClip = (xPos+16 > left+width) ? (16 - (left+width-xPos)) : 0;

                topClip = (yPos < top) ? top-yPos : 0;
                bottomClip = (yPos+16 > top+height) ? (16 - (top+height-yPos)) : 0;

                BltBitMap(bgBitMap,leftClip,topClip,
                          bm,xPos+leftClip,yPos+topClip,
                          16-leftClip-rightClip,16-topClip-bottomClip,
                          0xC0,0xFF,NULL);
            }
        }


        /* Bump position and determine if more rendering is needed */
        if((yPos+16) < (top))
        {
            /* Don't muck around, bump the line. */
            xPos = 0;
            yPos += 16;
        }
        else
        {
            xPos += 16;
            if(xPos >= (left+width))
            {
                xPos = 0;
                yPos += 16;
                if(yPos >= top+height)
                    break;  /* No more to do */
            }
        }
    }
    return 0;
}

/* Main program */
int main(int argc, char *argv[])
{
    IntuitionBase = (struct IntuitionBase *)OpenLibrary("intuition.library", 39);
    if(IntuitionBase != NULL)
    {
        GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 39);
        if(GfxBase != NULL)
        {
            /* Prepare the hook and bitmap */
            InitHook(&hook, scrBackFillFunc, NULL);
            bgBitMap = AllocBitMap(16, 16, 4, BMF_CLEAR, NULL);

            /* Open the screen */
            if(bgBitMap != NULL)
            {
                int i;

                /* Set the pattern in the bitmap. Only 2 planes used. */
                UWORD *planeData = (UWORD *)bgBitMap->Planes[0];
                for(i=0;i<16;++i)
                    *planeData++ = bgData[0][i];

                planeData = (UWORD *)bgBitMap->Planes[1];
                for(i=0;i<16;++i)
                    *planeData++ = bgData[1][i];

                /* Open the screen */
                scr = OpenScreenTags(NULL,
                                     SA_Title, "Screen Backfill Hook Example Screen",
                                    SA_Type, CUSTOMSCREEN,
                                    SA_Pens, pens,
                                    SA_BackFill, &hook, /* The hook to call on refresh */
                                    SA_LikeWorkbench, TRUE,
                                    SA_Depth, 4, /* Like Workbench, but just 16 colours */
                                    TAG_END);
                if(scr)
                {
                    struct Window *bgWin = NULL;

                    /* Open and close a full screen backdrop window to force the screen to
                    fully refresh. This window is not supposed to be seen. */
                    bgWin = OpenWindowTags(NULL,
                                           WA_CustomScreen, scr,
                                           WA_Left, 0,
                                           WA_Top, 0,
                                           WA_Width, scr->Width,
                                           WA_Height, scr->Height,
                                           /* Make window backdrop and borderless so we
don't
                                              see it flash up */
                                           WA_Borderless, TRUE,
                                           WA_Backdrop, TRUE,
                                           WA_SimpleRefresh, TRUE,
                                           WA_NoCareRefresh, TRUE,
                                           TAG_END);
                    if(bgWin)
                    {
                        /* Close straight away. This will trigger the screen backfill hook.
*/
                        CloseWindow(bgWin);
                    }

                    /* Open a window to show that the screen has the background
                    and to allow termination of the program. */
                    win = OpenWindowTags(NULL,
                                        WA_Left, 50,
                                        WA_Top, 50,
                                        WA_Width, 300,
                                        WA_Height, 100,
                                        WA_IDCMP, IDCMP_CLOSEWINDOW,
                                        WA_Title, "<=== Click to close",
                                        WA_CustomScreen, scr,
                                        WA_MinWidth, 0,
                                        WA_MinHeight, 0,
                                        WA_MaxWidth, (ULONG)-1,
                                        WA_MaxHeight, (ULONG)-1,
                                        WA_SizeGadget, TRUE,
                                        WA_DragBar, TRUE,
                                        WA_CloseGadget, TRUE,
                                        WA_NoCareRefresh, TRUE,
                                        WA_Activate, TRUE,
                                        WA_SmartRefresh, TRUE,
                                        TAG_END);

                    if(win)
                    {
                        BOOL done = FALSE;
                        struct IntuiMessage *iMsg;
                        ULONG sigWait = (1L << win->UserPort->mp_SigBit);
                        ULONG sigRcvd;
                        /* Message Loop. We only care about window close gadget. */
                        do
                        {
                            sigRcvd = Wait(sigWait);
                            while((iMsg = (struct IntuiMessage *)GetMsg(win->UserPort)) !=
                                   NULL)
                            {
                                if(sigRcvd & sigWait)
                                {
                                    switch(iMsg->Class)
                                    {
                                        case IDCMP_CLOSEWINDOW:
                                            done = TRUE;
                                            break;
                                    }

                                    ReplyMsg((struct Message *)iMsg); /* Reply all
                                                                         IntuiMessages */
                                }
                            }
                        } while(!done);

                        CloseWindow(win);
                    }
                    CloseScreen(scr);
                }

                WaitBlit(); // Make sure all blitting is done first!
                FreeBitMap(bgBitMap);
            }
            CloseLibrary((struct Library *)GfxBase);
        }
        CloseLibrary((struct Library *)IntuitionBase);
    }
    return RETURN_OK;
}
```

CLONING THE WORKBENCH

It was more difficult to clone the Workbench screen in previous versions of the OS. Along with that, there is no way to know what features will be added to the Workbench screen in future that you may need to be capturing. Finally, the Workbench screen is likely to be configured to look and perform best for the user, so cloning as many of those characteristics as possible will maximise the chances of your software working on the majority of configurations.

To address these issues, the **SA_CloneWorkbench** tag was added to **OpenScreenTagList()** which will clone as much of the Workbench screen as it can if you supply **TRUE** as its parameter. The copy of Workbench is then used as a base and individual attributes can be overridden by supplying other tags.

It is important to note that there is no guarantee that you will get an exact copy of the Workbench screen. There is only a guarantee that **SA_CloneWorkbench** will make its best attempt to clone the Workbench so don't assume you have received exactly what was in the Workbench screen. *Check the Screen structure first!*

INTERLEAVED SCREENS

The Amiga hardware uses planar bitmaps in order to minimise the amount of CHIP memory in use. This is what allows Amiga computers to display 8, 32, 64 & 128 colour screens along with the 2, 4, 16 or 256 colour screens contemporary computers were usually restricted to.

For example, a 16 colour screen of 320x200 pixels in size would require 32,000 bytes of memory to display using the equation (width×height×depth)÷8. This would mean that 32,000 bytes of contiguous memory would be needed to store the screen

Since the Amiga is using bit planes, instead of allocating 32,000 bytes of contiguous data, it can allocate 4 separate blocks of 8,000 bytes of CHIP RAM increasing the chances that the screen will fit.

Whilst this works well, a side-effect of allocating separate bit planes scattered throughout memory is that when scrolling the bitmap (for example, when scrolling text up in a console window) the Amiga must move one bit plane at a time resulting in ghosting, especially when there are screens containing 16 colours or more.

This is visually unappealing so from Release 3 a screen can now be allocated using an Interleaved bitmap. Although this loses the advantage of scatter-loading the screen's bitplanes across CHIP RAM, the ghosting can be removed because the blitter can move images line by line rather than plane-by-plane.

Release 3 introduces a new tag to **OpenScreenTagList()** called **SA_Interleaved**. If it is set to **TRUE** then the screen will attempt to be opened as an interleaved bitmap. If the screen cannot obtain an interleaved bitmap (e.g. because there is not enough contiguous CHIP RAM) the screen will instead be opened with a normal planar bitmap. As such, **SA_Interleaved** is merely a request for an interleaved bitmap, not a guarantee that you will get one if the screen is opened. If **SA_Interleaved** is set to **FALSE**, then you will definitely receive a non-interleaved bitmap if the screen is successfully opened.

EXCLUSIVE SCREENS

There are times you need to ensure that your screen is the only one that is visible on the display. Applications that may do this include screen blankers, presentation software, and games.

To achieve this under Release 3, set the **SA_Exclusive** tag to **TRUE** when calling **OpenScreenTagList()**.

This tells Intuition that you do not want any other screen to be visible while your screen is showing. Screen dragging will not be enabled, so it can't be pulled down, and the screen will also not show if the screen in front is dragged down. Screen back and front gadgets will still behave in their usual manner.

Under v40, setting **SA_Exclusive** to **TRUE** for attached screens (see *Attached Screens* below) will allow those screens to behave as a family. This means that they will share the screen together but no other screen can be displayed at the same time.

PEN SHARING

By default, when a new screen is created, all the pens supplied by the **SA_Pens** tag when calling **OpenScreenTagList()** are allocated as shared pens. This means that the pens are available for use by any other application that is also sharing the screen. If there are more pens than the number defined in **SA_Pens** then those pens will be allocated as exclusive pens, meaning no other application sharing the screen will be able to request those pens for drawing.

Under v39 and higher of Intuition, **OpenScreenTagList()** now supports the **SA_SharePens** flag. If set to **TRUE**, then the screen will only allocate the pens supplied by the **SA_Pens** tag, again as shared pens. All other pens will NOT be allocated and are free to be allocated by any other program. This is ideal if you want to manage the pens yourself.

Double Buffered Screens

Intuition now supports double-buffering natively in Screens in a way that is compatible with gadgets and menus.

This is accomplished through the use of the new Intuition functions; **AllocScreenBuffer()**, **ChangeScreenBuffer()** and **FreeScreenBuffer()** in conjunction with additional support provided by graphics.library.

The key to this is the new **ScreenBuffer** structure found in screens.h. It looks like this:

```
struct ScreenBuffer
{
    struct BitMap *sb_BitMap;        /* BitMap of this buffer */
    struct DBufInfo *sb_DBufInfo;    /* DBufInfo for this buffer */
};
```

The **sb_BitMap** pointer points to the bitmap that is associated with a particular ScreenBuffer, and the **sb_DBufInfo** pointer points to the underlying graphics structure that enables the double buffering feature described here.

The only way to allocate a **ScreenBuffer** structure is to call **AllocScreenBuffer()**. You will need to create as many **ScreenBuffer** structures as buffers you would like to swap. The double buffering is not limited to two buffers so you can have as many bitmaps as memory will allow, though you are unlikely to need more than triple-buffering.

Once your screen has been opened and set up the way you want (including any menus and gadgets), you will need to allocate a **ScreenBuffer** for it. The **AllocScreenBuffer()** function takes parameters as described below:

```
struct ScreenBuffer *AllocScreenBuffer(struct Screen *, struct BitMap *, ULONG);
```

When allocating a **ScreenBuffer** for your original screen, the screen must be passed as the first parameter.

If the screen you created is of type **CUSTOMBITMAP**, you will need to supply a pointer to that **BitMap** as the second parameter. If your screen does not use a custom bitmap, set the second parameter to **NULL**.

The third parameter is for flags which tell **AllocScreenBuffer()** how to behave. If your screen is a **CUSTOMBITMAP** type, this parameter can be set to 0 and the associated **BitMap** will be the **BitMap** you passed as the second parameter. If your screen does not use a custom bitmap, set this parameter to **SB_SCREEN_BITMAP** which instructs **AllocScreenBuffer()** not to allocate any bitmap, but instead to use the **Screen**'s actual **BitMap**. You need to do this for the original **ScreenBuffer** that will be associated with your **Screen**.

Once you have allocated a **ScreenBuffer** for your **Screen**, you need to allocate one for each of the extra buffers you need. The method is the same as the original **ScreenBuffer** allocation, including passing the **Screen** as the first parameter, but you set the third parameter to **SB_COPY_BITMAP**. This will cause **AllocScreenBuffer()** to make a copy of the **Screen**'s **BitMap** which is required in order to retain Menu Bar and **Gadget** imagery in the screen when swapping **BitMap**s.

The example code below shows how to allocate and free **ScreenBuffer**s for a double buffered screen. Error checking is not shown for brevity.

```
struct Screen *scr;
struct ScreenBuffer *sb[2];
…
/* Set up and open the screen */
scr = OpenScreenTags(NULL, …);

/* Allocate the primary screen buffer. */
sb[0] = AllocScreenBuffer(scr, NULL, SB_SCREEN_BITMAP);

/* Now allocate the extra screen buffer with a copy of the screen's bitmap */
sb[1] = AllocScreenBuffer(scr, NULL, SB_COPY_BITMAP);
…
/* Use the double buffering setup. */
...
/* Wait for any unreplied messages (see below) */
…
/* Free the screen buffers BEFORE closing the screen */
```

```
FreeScreenBuffer(scr, sb[1]);
FreeScreenBuffer(scr, sb[0]);
CloseScreen(scr);
```

Once the **ScreenBuffer** structures have been allocated you are now ready to implement double buffering.

The key to the double buffering lies in the **DBufInfo,** the pointer to which can be found in your **ScreenBuffer**. A **DBufInfo** was also allocated when you called the **AllocScreenBuffer()** function.

```
struct DBufInfo {
       APTR   dbi_Link1;
       ULONG  dbi_Count1;
       struct Message dbi_SafeMessage; /* replied to when safe to write to old bitmap */
       APTR dbi_UserData1;            /* first user data */

       APTR   dbi_Link2;
       ULONG  dbi_Count2;
       struct Message dbi_DispMessage; /* replied to when new bitmap has been displayed at
                                          least once */
       APTR   dbi_UserData2;          /* second user data */
       ULONG  dbi_MatchLong;
       APTR   dbi_CopPtr1;
       APTR   dbi_CopPtr2;
       APTR   dbi_CopPtr3;
       UWORD  dbi_BeamPos1;
       UWORD  dbi_BeamPos2;
};
```

Most of the fields in **DBufInfo** are private, but there are a few that are of use to a user application:

dbi_SafeMessage
An embedded Exec Message structure that Intuition will reply to so you know when it is safe to write to the BitMap. It may not be safe to render because of menu or gadget rendering, for example, so you need to receive this message before rendering. ***Note that the message is replied TO you, so you do not need to ReplyMsg() this message when received.***

dbi_DispMessage
Similar to dbi_SafeMessage, this is an embedded Exec Message structure that Intuition will reply to so you know that the BitMap has been fully displayed at least once. This provides a mechanism to ensure you don't swap BitMaps too quickly. ***Note that the message is replied TO you, so you do not need to ReplyMsg() this message when received.***

dbi_UserData1, dbi_UserData2
Reserved for your use.

So that Intuition can reply to **dpi_SafeMessage** and **dpi_DispMessage**, you will need to set the **mn_ReplyPort** of each of those messages to the **MsgPort** you will use to receive the messages. You will not need to send the messages yourself because it is done when you call **ChangeScreenBuffer()**.

The message reply ports are set up as shown below. Again error checking is ignored for brevity:

```
struct Screen *scr;
struct MsgPort *safePort, *dispPort;
struct ScreenBuffer *sb;
…
safePort = CreateMsgPort();
dispPort = CreateMsgPort();
…
scr = OpenScreenTags(NULL, …);
sb = AllocScreenBuffer(scr, NULL, SB_SCREEN_BITMAP);
sb->sb_DBufInfo->dbi_SafeMessage.mn_ReplyPort = safePort;
sb->sb_DBufInfo->dbi_DispMessage.mn_ReplyPort = dispPort;
...
```

Once all the setup has been completed, you follow a process that waits until the backup buffer is safe to be written to, render to the backup buffer, wait until the current buffer has been displayed for at least one frame, call **WaitBlit()** to ensure the **BLITTER** has finished its work, call **ChangeScreenBuffer()**, and loop again using the other buffer that was just swapped out.

Finally, when all the work is completed, the **ScreenBuffer** and **MsgPort** structures need to be freed after ensuring all outstanding messages to the ports have been replied to.

> *Screen buffers do not need to be as originally assigned.* Although you allocate one **ScreenBuffer** using the **Screen**'s original **BitMap**, and a second **ScreenBuffer** using a copy of the **Screen**'s **BitMap**, it does not matter which **ScreenBuffer** is assigned when calling **FreeScreenBuffer()**.

To free up the resources used by a **ScreenBuffer**, call **FreeScreenBuffer()**. The **FreeScreenBuffer()** function takes parameters as described below:

```
VOID FreeScreenBuffer(struct Screen *, struct ScreenBuffer *);
```

The first parameter is a pointer to the screen the **ScreenBuffer** was allocated for, and the second parameter is a pointer to the **ScreenBuffer** itself.

The steps below summarise what needs to be done and assume double buffering with a screen type that is not of type **CUSTOMBITMAP**:

1. Create and open the **Screen**.
2. Create required **MsgPort**s for **dpi_SafeMessage** and **dpi_DispMessage**.
3. Open the window and set up any menus & gadgets required.

4. Allocate **ScreenBuffer**s.
    a. Call **AllocScreenBuffer()** using the screen BitMap (sb1).
    b. Call **AllocScreenBuffer()** making a copy of the screen BitMap (sb2).
5. Assign **MsgPort**s to **ScreenBuffer**s.
    a. Assign a **MsgPort** to sb1->sb_DBufInfo->dbi_SafeMessage.mn_ReplyPort.
    b. Assign a **MsgPort** to sb1->sb_DBufInfo->dbi_DispMessage.mn_ReplyPort.
    c. Assign a **MsgPort** to sb2->sb_DBufInfo->dbi_SafeMessage.mn_ReplyPort.
    d. Assign a **MsgPort** to sb2->sb_DBufInfo->dbi_DispMessage.mn_ReplyPort.
6. The buffer to display is now sb1 and the buffer to render into is now sb2.
7. When ready to render:
    a. Wait until the render buffer is safe to write (i.e **dbi_SafeMessage** is replied to your port).
       NOTE: if this is the first render, there is no message to wait for so you can skip step a and step c.
    b. Render into the render buffer.
    c. Wait until the current display buffer has been displayed at least once (i.e. **dbi_DispMessage** is replied to your port).
    d. Call **WaitBlit()**.
    e. Call **ChangeScreenBuffer()**.
    f. Switch the buffers (i.e. If the current buffer was sb1, it is now sb2; and vice-versa).
8. When the screen is ready to be cleaned up:
    a. Wait for final **dbi_SafeMessage** to be returned for RenderBuffer.
    b. Wait for final **dbi_DispMessage** to be returned for CurrentBuffer.
    c. Free the **ScreenBuffer**s
        i. Call **FreeScreenBuffer()** for sb2.
        ii. Call **FreeScreenBuffer()** for sb1.
    d. Free the allocated **MsgPort**s.
    e. Call **CloseScreen()**.

Finally, the sample program below shows double-buffering in action:

```
/* V39+ Double Buffering example */
#include <exec/types.h>
#include <exec/libraries.h>
#include <dos/dos.h>

#include <intuition/screens.h>

#include <proto/exec.h>
#include <proto/graphics.h>
#include <proto/intuition.h>
#include <proto/diskfont.h>
#include <proto/gadtools.h>

#include <stdio.h>
#include <string.h>

struct Screen *scr;
struct Window *win;
struct MsgPort *safePort, *dispPort;
struct Menu *menuStrip;
struct TextFont *f;

UWORD diameter = 50;
BOOL safeToChange = TRUE;
BOOL safeToWrite = TRUE;
BOOL useDBuf = TRUE;
int currentBuf = 0;
int singleBufNum;
int speed = 1;

WORD pens[] = {-1}; /* Use default pens */
struct TextAttr topaz80 = {"topaz.font", 8, 0, 0};
struct TextAttr clientTA = {"diamond.font", 100, FSF_BOLD, 0};
struct TextExtent te;
struct ScreenBuffer *sb[2];
struct RastPort rpCopy;
APTR vi; /* Visual Info */

struct ColorSpec cSpecs[] = {
    {0, 0xA, 0xA, 0xA},
    {1, 0x0, 0x0, 0x0},
    {2, 0xF, 0xF, 0xF},
    {3, 0x4, 0x5, 0xA},
    {-1},
};

char scrollText[20];

struct NewMenu nm[] = {
    {NM_TITLE, "Project", NULL, 0, 0, NULL},
        {NM_ITEM,  "Quit", "Q",  0, 0, NULL},
    /* -------------------------------------------- */
    {NM_TITLE, "Preferences", NULL, 0, 0, NULL},
        {NM_ITEM,  "Double-Buffer",  NULL, CHECKIT|MENUTOGGLE|CHECKED, 0, NULL},
        {NM_ITEM,  "Increase Speed", "+",  0, 0, NULL},
        {NM_ITEM,  "Decrease Speed", "-",  0, 0, NULL},
    /* -------------------------------------------- */
    {NM_END}
};

int main(int argc, char *argv[])
{
    IntuitionBase = (struct IntuitionBase *)OpenLibrary("intuition.library", 39);
    if(IntuitionBase != NULL)
    {
        GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 39);
        if(GfxBase != NULL)
        {
            GadToolsBase = OpenLibrary("gadtools.library", 39);
            if(GadToolsBase != NULL)
            {
                DiskfontBase = OpenLibrary("diskfont.library", 39);
                if(DiskfontBase != NULL)
                {
                    scr = OpenScreenTags(NULL,
                                         SA_DisplayID, 0,
                                         SA_Width, 320,
                                         SA_Height, ~0,
                                         SA_Depth, 2,   /* 4 colours. */
                                         SA_Title, "v39 Double Buffer Example Screen",
                                         SA_Type, CUSTOMSCREEN,
                                         SA_Pens, pens,
                                         SA_Colors, cSpecs,
                                         SA_Font, &topaz80,
                                         TAG_END);
                    if(scr)
                    {
                        ULONG safeSigMask = 0;
                        ULONG dispSigMask = 0;

                        /* Allocate ports for double buffer notifications */
                        safePort = CreateMsgPort();
                        dispPort = CreateMsgPort();

                        /* Need a Visual Info for gadtools stuff */
                        vi = GetVisualInfo(scr, TAG_END);

                        /* Get a nice large font for rendering */
                        f = OpenDiskFont(&clientTA);

                        if(safePort && dispPort && vi && f)
                        {
                            /* Open the window. */
                            win = OpenWindowTags(NULL,
                                                 WA_CustomScreen, scr,
                                                 WA_IDCMP, IDCMP_INTUITICKS | IDCMP_MENUPICK,
                                                 WA_SizeGadget, FALSE,
                                                 WA_DragBar, FALSE,
                                                 WA_DepthGadget, FALSE,
                                                 WA_CloseGadget, FALSE,
                                                 WA_Borderless, TRUE,
                                                 WA_Backdrop, TRUE,
                                                 WA_Activate, TRUE,
                                                 WA_SmartRefresh, TRUE,
                                                 WA_NoCareRefresh, TRUE,
                                                 WA_Width, scr->Width,
                                                 WA_Height, scr->Height,
                                                 WA_NewLookMenus, TRUE,
                                                 TAG_END);
```

```
if(win)
{
    BOOL done = FALSE;
    struct IntuiMessage *iMsg;
    struct MenuItem *item;
    ULONG winSigMask;
    ULONG sigWait;
    ULONG sigRcvd;
    WORD x,y;
    UWORD menuNumber, menuNum, itemNum;

    /* Set up the menus (non-fatal) */
    menuStrip = CreateMenus(nm, GTMN_FullMenu, FALSE, TAG_END);
    if(menuStrip)
    {
        if(LayoutMenus(menuStrip, vi, GTMN_NewLookMenus, TRUE, TAG_END))
        {
            /* Add the menu if it was successfully created */
            SetMenuStrip(win, menuStrip);
        }
    }

    /* Allocate the screen buffers */
    sb[0] = AllocScreenBuffer(scr, NULL, SB_SCREEN_BITMAP);
    sb[1] = AllocScreenBuffer(scr, NULL, SB_COPY_BITMAP);
    if(sb[0] && sb[1])
    {
        /* These message ports enable the OS to advise your application
           of when to write to the bitmap and also when a screen has
           been fully displayed at least once so flashing can be eliminated */
        sb[0]->sb_DBufInfo->dbi_SafeMessage.mn_ReplyPort = safePort;
        sb[1]->sb_DBufInfo->dbi_SafeMessage.mn_ReplyPort = safePort;

        sb[0]->sb_DBufInfo->dbi_DispMessage.mn_ReplyPort = dispPort;
        sb[1]->sb_DBufInfo->dbi_DispMessage.mn_ReplyPort = dispPort;

        /* Prepare text and RastPort for display. */
        strncpy(scrollText, "Do I flash?", 19);
        SetFont(win->RPort, f);
        SetABPenDrMd(win->RPort, 2, 3, JAM2);
```

```
TextExtent(win->RPort, scrollText, strlen(scrollText), &te);

/* Use a copy of the RastPort we can switch bitmaps for */
memcpy(&rpCopy, win->RPort, sizeof(struct RastPort));

x = scr->Width;
y = (scr->Height/2);

/* Prepare our signal masks. Only the window signal is waited on in the
main loop */
winSigMask = 1L << win->UserPort->mp_SigBit;
safeSigMask = 1L << safePort->mp_SigBit;
dispSigMask = 1L << dispPort->mp_SigBit;
sigWait = winSigMask;

/* Message Loop. We only care if the window close gadget was pressed. */
do
{
    sigRcvd = Wait(sigWait);

    if(sigRcvd & winSigMask)
    {
        while((iMsg = (struct IntuiMessage *)GetMsg(win->UserPort)) != NULL)
        {
            if(sigRcvd & sigWait)
            {
                switch(iMsg->Class)
                {
                    case IDCMP_MENUPICK:
                        menuNumber = iMsg->Code;
                        while((menuNumber != MENUNULL) && !done)
                        {
                            item = ItemAddress(menuStrip, menuNumber);

                            menuNum = MENUNUM(menuNumber);
                            itemNum = ITEMNUM(menuNumber);

                            switch(menuNum)
                            {
                                case 0:
```

```
        switch(itemNum)
        {
            case 0:
                done = TRUE;
                break;
        }
        break;

    case 1:
        switch(itemNum)
        {
            case 0:
                if(useDBuf == TRUE)
                {
                    /* Switch to single buffer. We
                    need to set the used buffer as
                    opposite to the current buffer
                    because that is the one currently
                    used by the screen. */
                    singleBufNum = currentBuf ^ 1;
                    useDBuf = FALSE;
                }
                else
                {
                    /* Switch to double buffer */
                    useDBuf = TRUE;
                }
                break;

            case 1:
                if(speed < 10)
                    ++speed;
                break;

            case 2:
                if(speed > 1)
                    --speed;
                break;
        }
    break;
```

```
        }
        menuNumber = item->NextSelect;
    }
    break;

case IDCMP_INTUITICKS:
    if(useDBuf)
    {
        /* Use Double-Buffering */
        /* Ensure the buffer is ready to be written into */
        if(!safeToWrite)
        {
            while(!GetMsg(safePort))
            {
                Wait(safeSigMask);
            }
            safeToWrite = TRUE;
        }

        /* Render into the current off-screen bitmap */
        rpCopy.BitMap = sb[currentBuf]->sb_BitMap;
        EraseRect(&rpCopy, x, y-f->tf_Baseline, scr->Width, y+te.te_Height);
        if((x + te.te_Width) < 0)
            x = scr->Width;
        else
            x -= speed;

        Move(&rpCopy, x, y);
        Text(&rpCopy, scrollText, strlen(scrollText));

        /* Wait until the previous bitmap has been shown at least once */
        if(!safeToChange)
        {
            while(!GetMsg(dispPort))
            {
                Wait(dispSigMask);
            }
        }
        safeToChange = TRUE;
```

```
                                WaitBlit(); /* Make sure rendering is done */

                                /* Switch buffers */
                                ChangeScreenBuffer(scr, sb[currentBuf]);

                                safeToChange = FALSE;
                                safeToWrite = FALSE;

                                /* Make sure the off-screen buffer is the one we write into next */
                                currentBuf ^= 1;
                            }
                            else
                            {
                                /* Do not use Double-Buffering */
                                /* Render into the current on-screen bitmap. */
                                rpCopy.BitMap = sb[singleBufNum]->sb_BitMap;
                                EraseRect(&rpCopy, x, y-f->tf_Baseline, scr->Width, y+te.te_Height);
                                if((x + te.te_Width) < 0)
                                    x = scr->Width;
                                else
                                    x -= speed;

                                Move(&rpCopy, x, y);
                                Text(&rpCopy, scrollText, strlen(scrollText));
                            }
                            break;

                        default:
                            break;
                    }

                    ReplyMsg((struct Message *)iMsg);
                }
            }
        }
    } while(!done);
}

if(menuStrip)
{
```

```
                ClearMenuStrip(win);
                FreeMenus(menuStrip);
            }
            CloseWindow(win);
        }
    }

    if(f)
        CloseFont(f);

    if(vi)
        FreeVisualInfo(vi);

    /* Clean up pending messages */
    if(!safeToChange)
    {
        while(!GetMsg(dispPort))
            Wait(dispSigMask);
    }

    if(!safeToWrite)
    {
        while(!GetMsg(safePort))
            Wait(safeSigMask);
    }

    if(dispPort)
        DeleteMsgPort(dispPort);

    if(safePort)
        DeleteMsgPort(safePort);

    if(sb[1])
        FreeScreenBuffer(scr, sb[1]);

    if(sb[0])
        FreeScreenBuffer(scr, sb[0]);

    CloseScreen(scr);
}
```

```
                    CloseLibrary(DiskfontBase);
                }
                CloseLibrary(GadToolsBase);
            }
            CloseLibrary((struct Library *)GfxBase);
        }
        CloseLibrary((struct Library *)IntuitionBase);
    }
    return RETURN_OK;
}
```

Attached Screens

Some applications present their interface using more than one screen at once. An example of such an application could be a picture editing program. It displays the picture being edited in the screen behind and the user interface buttons in another screen that is in front. This would allow full control of the resolution and display of the gadgets and other UI elements in the front screen while allowing the picture being edited to use all its native resolution and colours without being disturbed by the UI.

Although this could be done in all versions of AmigaOS, the position and depth of the screens is difficult to monitor and manage because the screens are all considered independent by Intuition.

Now one screen can be associated with another so that one or more screens can be addressed together. A child screen could be prevented from being scrolled independent of its parent and also prevented from going behind the parent so that they always appear to the user as one clear interface rather than two independent screens.

Full Example

Below is an example of opening a 16-colour screen with a window that will close once the Close gadget has been pressed. It requires AmigaOS 3.0 (V39) or better. This example is written for SAS/C 6.5x but should work with other compilers with little changes.

```
/*
** OpenScreen V39 example.
*/
#include <stdio.h>
#include <stdlib.h>

#include <exec/types.h>          /* Contains all the standard Amiga data types */
#include <exec/libraries.h>      /* Defines the library structure used to access
                                    IntuitionBase */
#include <intuition/intuition.h> /* Defines intuition structures. Vital for any application
                                    using Intuition */
#include <proto/exec.h>          /* Defines everything needed for Exec Library functions to
                                    be called. This includes the definition of SysBase.
                                    Without that global variable being defined you will not
                                    be able to use Exec. Unlike Intuition and Graphics
                                    libraries you do not need to open the Exec library
                                    because it is always opened for you by default. */
#include <proto/intuition.h>     /* Defines everything needed for Intuition library
                                    functions to be called. This includes the definition of
                                    IntuitionBase. Without that global variable being
                                    defined you will not be able to use Intuition. */
#include <proto/graphics.h>      /* Defines everything needed for Graphics Library
                                    functions to be called. This includes the definition of
                                    GfxBase. Without that global variable being defined you
                                    will not be able to use Graphics. */
```

```
/* For a screen to support the 3D look of Release 2 and higher a set of pens must be
   defined. This is the bare minimum required. It tells Intuition to allocated the default
   pen set. If this is not included when opening the screen you will not get the 3D look.
*/
LONG scrPens[] = {~0};

/* This is the main function for drawing the content of the window. */
VOID DrawBoxes(struct Window *win)
{
   int i,j;
   int numCols = 8;
   int numRows = 2;

   WORD left, top, width, height;
   WORD boxWidth, boxHeight;

   /* Here we determine the client area */
   left = win->BorderLeft;
   top = win->BorderTop;
   width = win->Width - win->BorderLeft - win->BorderRight;
   height = win->Height - win->BorderTop - win->BorderBottom;

   boxWidth = width / 8;
   boxHeight = height / 2;

   for(i=0; i<numRows; ++i)
   {
       WORD boxTop = top + (i * boxHeight);

       for(j=0; j<numCols; ++j)
       {
           WORD boxLeft = left + (j * boxWidth);

           /* The A Pen is the main pen for drawing. Setting this to a different number
              tells RectFill() below to draw in a different colour each time it is called.
           */
           SetAPen(win->RPort, (i*8)+j);

           /* RectFill() actually draws the colour blocks in the window. */
           RectFill(win->RPort, boxLeft, boxTop, boxLeft + boxWidth, boxTop + boxHeight);
       }
   }
   return;
}

void main(int argc, char *argv[])
{
   /* Before we do anything with Intuition the library must first be opened.
      Asking for version 39 means we don't want any version of Intuition
      older than AmigaOS 3.0. */
   IntuitionBase = (struct IntuitionBase *)OpenLibrary("intuition.library", 39L);
   if (IntuitionBase != NULL)
   {
       /* As with Intuition, the Graphics library needs to be opened because SetAPen() and
          RectFill()are part of the graphics library. */
       GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 39L);
       if(GfxBase != NULL)
       {
           LONG errVal;

           struct Screen *scr = OpenScreenTags(NULL,
                                  SA_DisplayID, HIRES_KEY, /* Tells Intuition we want a
```

```
                                        HIRES screen mode */
                        SA_Pens, scrPens, /* Without this tag, the screen will
                                             not be rendered with the 3D look.
                                             Experiment with removing this tag
                                             to observe the difference */
                        SA_Width, 640,
                        SA_Height, 200,
                        SA_Depth, 4, /* 16 colours */
                        SA_Title, "Release 3 Test Screen",
                        SA_Type, CUSTOMSCREEN, /* Your own screen to do with as
                                                  you please */
                        SA_FullPalette, TRUE,  /* Set all colours defined by
                                                  user preferences */
                        SA_ErrorCode, &errVal, /* If the screen open fails, this
                                                  variable will contain the
                                                  reason */
                        TAG_DONE);

    if (scr != NULL)
    {
        struct Window *win = OpenWindowTags(NULL,
                            WA_Top, (scr->Height / 2) - 50,
                            WA_Left, (scr->Width / 2) - 280,
                            WA_InnerWidth, 560, /* WA_InnerWidth defines the
                                                   width but does not include
                                                   the window borders. */
                            WA_InnerHeight, 100, /* WA_InnerHeight defines the
                                                    width but does not include
                                                    the window borders. */
                            WA_CustomScreen, scr, /* This is our defined screen.
                                                     This must be defined if the
                                                     window is to open on our
                                                     screen instead of the
                                                     default screen (usually the
                                                     Workbench) */
                            WA_CloseGadget, TRUE, /* Provide a Close gadget so
                                                     the window can be closed */
                            WA_SmartRefresh, TRUE,
                            WA_DragBar, TRUE,
                            WA_NoCareRefresh, TRUE, /* No need to care about
                                                       refreshing the window
                                                       here. */
                            WA_Activate, TRUE,
                            WA_IDCMP, IDCMP_CLOSEWINDOW, /* Allow Intuition to
                                                            tell us when the
                                                            Close gadget has
                                                            been pressed. */
                            WA_Title, "Release 3 Test Window",
                            TAG_DONE);

        if(win != NULL)
        {
            struct IntuiMessage *iMsg;
            ULONG iClass;
            ULONG winSig, rcvdSigs;
            BOOL done = FALSE;

            /* Draw the coloured boxes within the window */
            DrawBoxes(win);

            /* This is the signal that Intuition will use to tell us when something
               happens to the window. In this case the only thing it will tell us is
               when the Close gadget is pressed. */
            winSig = 1L << win->UserPort->mp_SigBit;

            while(!done)
            {
                /* Wait until Intuition tells us something has happened to the
                   window. */
                rcvdSigs = Wait(winSig);

                if(rcvdSigs & winSig)
                {
                    /* Something happened. Check our messages to find out what. */
                    while((iMsg = (struct IntuiMessage *)GetMsg(win->UserPort))
                          != NULL)
                    {
                        /* Take copies of the parts of the message that we need and
                           reply to the message as soon as possible so that
                           Intuition can free or re-use it. Be careful not to
                           reference the iMsg once replied because it no longer
                           belongs to you. */
                        iClass = iMsg->Class;
                        ReplyMsg((struct Message *)iMsg);

                        switch(iClass)
                        {
                            case IDCMP_CLOSEWINDOW:
                                /* The Close gadget was pressed. Quit the loop and
                                   the program. */
                                done = TRUE;
                                break;
                        }
                    }
                }
            }
            /* The window must be closed when done. Note there is NO automatic
               cleanup of resources. */
            CloseWindow(win);
        }
        else
        {
            printf("Could not open the window!\n");
        }

        /* The screen must also be closed when done. */
        CloseScreen(scr);
    }
    else
    {
        /* The screen failed to open. Why? Let’s check the error code we supplied
           to the OpenScreenTags() function. For the example we will just print
           out the error code.*/
        printf("Could not open the screen. Error code %ld!\n", errVal);
    }

    /* Close the Graphics library when you are done with it. */
    CloseLibrary((struct Library *)GfxBase);
}
else
{
    printf("Could not open Graphics library v39!\n");
}
```

```
        /* Close the Intuition library when you are done with it. */
        CloseLibrary((struct Library *)IntuitionBase);
    }
    else
    {
        printf("Could not open Intuition library v39!\n");
    }

    return;
}
```

New and Updated Screen Functions

Function	Min Version	Description
AllocScreenBuffer()	**39**	Get a ScreenBuffer structure for double-buffering.
ChangeScreenBuffer()	**39**	Swap the screen's bitmap.
FreeScreenBuffer()	**39**	Free a ScreenBuffer structure.
HideWindow()	**46**	Ask Intuition to hide a window.
ShowWindow	**46**	Ask Intuition to make a hidden window visible.
MakeScreen()	30	**V39** - Previously returned VOID. Now returns 0 for success and non-zero for error.
OpenScreenTagList()	36	**V39** - Accepts many new tags. See *New Screen Features* above. **V47** - Can return a new error type: OSERR_NORTGBITMAP
RemakeDisplay()	30	**V39** - Previously returned VOID. Now returns 0 for success and non-zero for error.
RethinkDisplay()	30	**V39** - Previously returned VOID. Now returns 0 for success and non-zero for error.
ScreenDepth()	**39**	Depth-arrange a screen with extra control.
ScreenPosition()	**39**	Move screens with greater control.

Chapter 11

MENUS

Menus have not changed much during the evolution of AmigaOS. There have been some small tweaks as well as support libraries and classes that make it easier to create and manage menu strips but the underlying architecture is pretty much the same.

This chapter is intended to identify the changes and enhancements that were made for Amiga's menu system under Release 3. This will be broken down into three subsections; the changes implemented under Intuition (the lowest level), changes implemented through the gadtools.library, and an overview of how menus are created and managed with the window.class. Even if you are using Reaction, the information contained in the original Rom Kernel Manual: Libraries 3rd Edition as well as the Intuition and GadTools updates here are still worth reading because the window.class takes those menu structures as an input parameter to create its menus.

Intuition

Firstly, if you do not yet understand how Intuition menus work, you should first read Chapter 6 of the Amiga ROM Kernel Manual: Libraries 3rd Edition. This will give you a complete picture of how the Amiga Menu system works under Intuition which will not be repeated here.

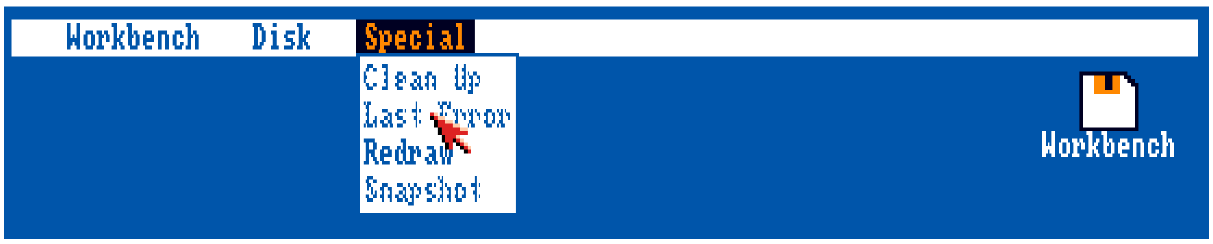
There are a couple of small changes that have been implemented for Intuition Menus for Release 3. These are New Look Menus and Menu Sharing.

New Look Menus

Although they could be created using any colour combinations the designer chooses, the original 1.x Workbench laid out the basic menu design as a white background with blue text, i.e. colour 1 is the background and colour 0 is the text.

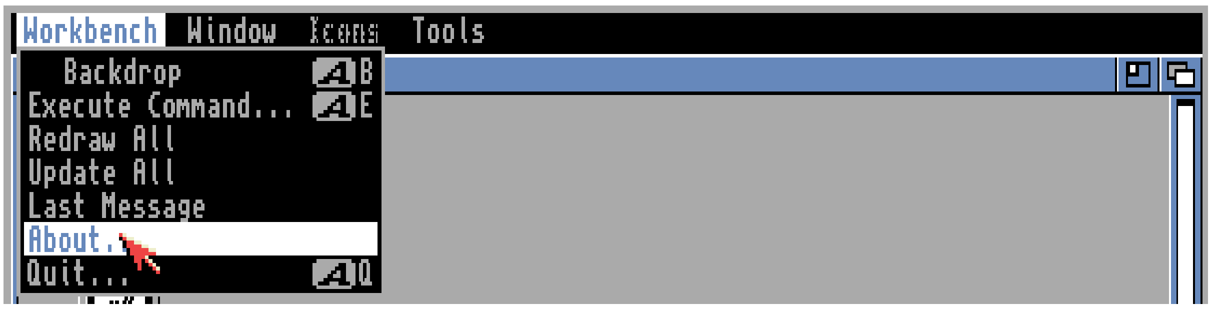
Highlighting is done by selecting the opposite colours. Since the Workbench was always 4 colours, this is colour 2 (black) for the background and colour 3 (orange) for the text,

This appears as shown below:



Release 2 brought about the introduction of a pseudo-3d design that is still based around a 4-colour Workbench, though other depths and sizes could now be selected.

This change introduced the base colour scheme as it still is under Release 3, with a grey background (colour 0) and black text. The Workbench uses the same pens to render the menu and title bar as it did under 1.x for compatibility. The basic menu design still retains colour 1 as the background (known as **BLOCKPEN**) and colour 0 as the menu text (known as **TEXTPEN**), which now meant that the menu bar was black and the menu text was grey. This looks the opposite of the original design but maintains compatibility for 1.x applications.



Release 3 added some extra pens to assist in the rendering of menus amongst other things, though menus will remain rendered in the backward-compatible Release 2 style unless specifically requested.

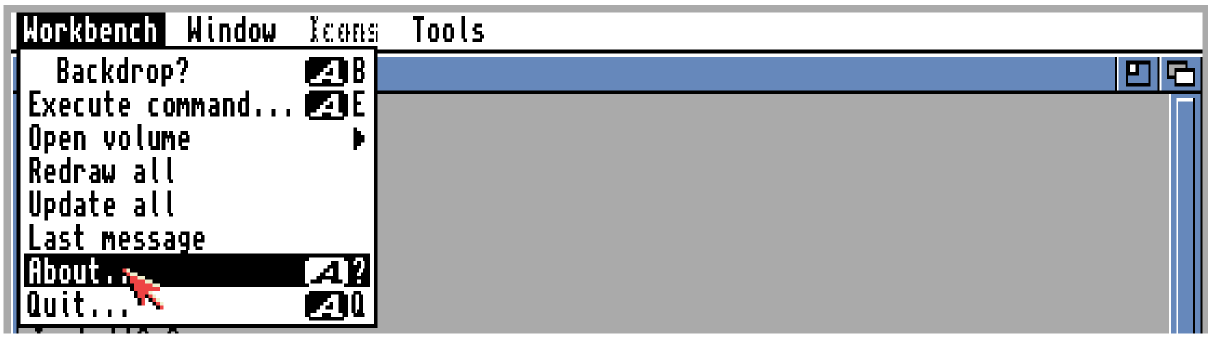
The way to request this is to set the **WFLG_NEWLOOKMENUS** flag when opening your window. This can be done by either:

- passing the tag **WA_NewLookMenus** set to **TRUE** when calling **OpenWindowTagList()**; or
- setting **WFLG_NEWLOOKMENUS** to the **NewWindow.Flags** field when calling **OpenWindow()**.

Activating NewLookMenus means that the menus being created for the associated window will make use of the newly created pens. These pens are:

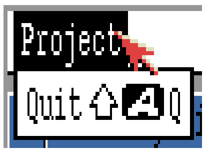
BARDETAILPEN	Pen for drawing the text and details in the screen bar/menu.
BARBLOCKPEN	Pen for filling the screen bar/menu background.
BARTRIMPEN	Pen for drawing the trim under the screen bar/menus.
BARCONTOURPEN	Pen for drawing the contour above the screen title bar and menus (V47)

Note that all the new pens exist from AmigaOS 3.0 onwards, but you need AmigaOS 3.2 or greater to make use of **BARCOUNTOURPEN**. This gives the look and feel that you can see below. Also, it allows for better customisation of menu colours.



Menu Shortcut Enhancements

Release 3.2 has added an extra option to the menu selection shortcut. Before 3.2, you had the option of Right-Amiga and a key (e.g. <RAmiga>-Q to quit). In addition, you can now specify the Shift key also. The user will see something similar to the image below when the menu item is shown:



The up-arrow represents any shift key. Used alongside the stylised 'A' as shown above indicates that the menu can be selected by pressing the 'Q', Right-Amiga and any Shift key simultaneously.

The flag to enable this feature is called **MIF_SHIFTCOMMSEQ**.

To use it in an Intuition menu, set the **MIF_SHIFTCOMMSEQ** flag in the MenuItem.Flags field of the item or sub-item you want to apply the shortcut to.

To use it in a GadTools menu, set the **MIF_SHIFTCOMMSEQ** flag in the NewMenu.nm_Flags field of the item or sub-item you want to apply the shortcut to.

Menu Lending

Menu lending was originally envisioned to be used alongside attached screens so that one screen could contain the primary window that handles all menu operations, whereas other windows which may or may not be on the same screen can simply be support screens.

AmigaOS has always allowed menus to be shared to an extent, in that the same menu structure can be shared between windows if desired. This works just fine for almost all situations but requires each window to handle and process the menu independently.

Using **LendMenus()**, the main window can "lend" its menu operations to other windows on other screens so that the main window/screen will always display and process the menu operations. The other windows do not need to handle the menu operations, but still enable them to be activated when they are the active window.

An example of this might be a program with a full-sized parent screen which has a short control panel screen attached in the front. Pressing the menu button even when the control panel window of the canvas screen is active can now cause the menus of the parent screen to appear.

See the Attached Screens chapter above for details about using parent/child screens.

GadTools Menus

There is no major change in the way that menus are created and handled under GadTools since Release 2. There are, however, some new tags that can be passed to enhance control over the menu being created.

New Tags

All the new tags listed below are available in **CreateMenus()**, **LayoutMenus()** & **LayoutMenuItems()**.

GTMN_FrontPen
GTMN_FrontPen specifies the pen that will be used to render menu text and has always been available for use in the **CreateMenus()** function. This has now also been made available to the **LayoutMenus()** and **LayoutMenuItems()** functions.

GTMN_NewLookMenus
GTMN_NewLookMenus has the same effect within GadTools menus as it does in standard Intuition menus. Activating NewLookMenus means that the menus being created for the associated window will make use of the newly created pens. These pens are:

BARDETAILPEN	Pen for drawing the text and details in the screen bar/menu.
BARBLOCKPEN	Pen for filling the screen bar/menu background.
BARTRIMPEN	Pen for drawing the trim under the screen bar/menus.
BARCONTOURPEN	Pen for drawing the contour above the screen title bar and menus (V47)

Note that all the new pens exist from AmigaOS 3.0 onwards, but you need AmigaOS 3.2 or greater to make use of **BARCOUNTOURPEN**. This gives the look and feel that you can see below. Also, it allows for better customisation of menu colours.

Under GadTools, **GTMN_NewLookMenus** also selects the appropriate Checkmark and Amiga-key images where needed.

GTMN_CheckMark
Use **GTMN_CheckMark** to pass a custom image that will be used to render a checkmark where required. If this tag is not specified, GadTools will supply a default image.

GTMN_AmigaKey
Use **GTMN_AmigaKey** to pass a custom image that will be used to render a checkmark where required. If this tag is not specified, GadTools will supply a default image.

BOOPSI Images In Menus

As of Release 3, BOOPSI images can be used for menu items and sub-items. In release 2, these images could only be standard Intuition Image structures. To use them, simply create the image you want via an Intuition **NewObject()** call and then add them to the menu where you previously supplied Intuition images.

Menus Under Reaction

There is no special type of menu in use by Reaction. It uses the same menu structures that were used under Intuition and GadTools. The two key differences are in how you supply the menus to the window and also how menu messages are processed.

Creating a Menu

You specify a menu when creating a Reaction window object by supplying the **WINDOW_NewMenu** tag with an array of **NewMenu** structures. This requires v47 or better of the window.class. If you want to supply a menu under earlier versions of the OS, you must supply the **WINDOW_MenuStrip** tag with already prepared and linked **Menu**/**MenuItem** structures.

Information on how to create menus using the traditional Intuition methods (**Menu**/**MenuItem** structures) can be found in the Amiga ROM Kernel Reference Manual 3rd Edition under Chapter 6 (Menus).

Information on how to create menus using GadTools (**NewMenu** structure array) can also be found in the Amiga ROM Kernel Reference Manual 3rd Edition, but under Chapter 15 (GadTools Library).

Handling Menu Messages Under Reaction

Processing of messages generated by menu selection is handled in the window's **WM_HANDLEINPUT** method. A menu message is identified by **WMHI_MENUPICK** or **WMHI_MENUHELP** being returned in the result. When retrieving the result of **WM_HANDLEINPUT**, use **WMHI_CLASSMASK** to isolate the class number.

The menu code is also supplied in the return value of **WMHI_HANDLEINPUT** and is identified by ANDing result with **WMHI_MENUMASK**. It represents the menu selection code which supports multiple menu selection in the same way as Intuition. The loop for handling menus in Reaction looks like this:

```
   GetAttr(WINDOW_Window, winobj, &win); // Retrieve Intuition Window from object.
   while(menuNumber != MENUNULL)
    {
        struct MenuItem *item;
        ULONG cmd;

        item = ItemAddress(win->MenuStrip, menuNumber);
       // Handle the selected menu item as required
       …
    }
```

Here is an example of setting up a menu for a Reaction Window using GadTools, and processing the “Quit” menu message to exit the program. This requires AmigaOS 3.2 because it uses the **WINDOW_NewMenu** tag when creating the Window object..

```
#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/classusr.h>
#include <classes/window.h>
#include <gadgets/layout.h>

#include <proto/exec.h>
#include <proto/intuition.h>
#include <proto/gadtools.h>
#include <proto/window.h>
#include <proto/layout.h>

#include <clib/alib_protos.h>

struct IntuitionBase *IntuitionBase;

struct Library *WindowBase;
struct Library *LayoutBase;

Class *WindowClass;
Class *LayoutClass;

/* Menu Structure as used by GadTools. There is no need to call any of the GadTools
** functions, such as CreateMenu(). This structure can be passed straight to the
** window on creation.
*/
struct NewMenu nm[] = {
    {NM_TITLE, "File", NULL, 0, 0, NULL},
    {NM_ITEM,  "Quit", "Q",  0, 0, NULL},
    {NM_END,   NULL,   NULL, 0, 0, NULL}
};

int main(int argc, char *argv[])
{
    /* Open libs, classes and gadgets we need */
    if((IntuitionBase = (struct IntuitionBase *)OpenLibrary("intuition.library", 40L)))
    {
        if((WindowBase = OpenLibrary("classes/window.class", 47L)))
        {
            if((LayoutBase = OpenLibrary("gadgets/layout.gadget", 44L)))
            {
                /* We cache the classes so we don't have to keep retrieving it from the
                   library. */
                WindowClass = WINDOW_GetClass();
                LayoutClass = LAYOUT_GetClass();

                if(WindowClass && LayoutClass)
                {
                    struct Gadget *layoutGad = NULL;
                    Object *winObj = NULL;
                    struct Window *win = NULL;
                    ULONG sigMask;
                    BOOL done = FALSE;

                    /* Create layout for window.
                    ** All Reaction windows need a layout.
                    ** This is a minimal layout that contains no real content.
                    */
                    layoutGad = (struct Gadget *)NewObject(LayoutClass, NULL, TAG_END);

                    /* Create window.
                    ** We do not need to check if the layout gadget above was created
                    ** because passing NULL as the WINDOW_Layout will cause the window not
to
                    ** open. We also pass through the NewWindow structure defined above so
       we
                    ** get our menus.
                    */
                    winObj = NewObject(WindowClass, NULL,
                                        WA_DepthGadget, TRUE,
                                        WA_DragBar, TRUE,
                                        WA_Activate, TRUE,
                                        WA_Title, "Reaction Menus",
                                        WA_InnerWidth, 600,
                                        WA_InnerHeight, 150,
                                        WA_NewLookMenus, TRUE,
                                        WINDOW_Position, WPOS_CENTERSCREEN,
                                        WINDOW_Layout, layoutGad,
                                        WINDOW_NewMenu, nm, /* This provides your menus. */
                                        TAG_END);

                    win = (struct Window *)DoMethod(winObj, WM_OPEN);

                    /* Get the sigmask to wait on.
                    ** If there is any chance the signal mask may change, move this code
                    ** inside the loop to ensure you have the latest mask.
                    */
                    GetAttr(WINDOW_SigMask, winObj, &sigMask);
                    while(!done)
                    {
                        ULONG sigsRcvd, result, selection;
                        UWORD code;

                        /* Standard wait call, but we just wait on the window signals */
                        sigsRcvd = Wait(sigMask);

                        /* WM_HANDLEINPUT is Reaction's message handler. It handles many
other
                        ** messages as well as the ones you are waiting for. When there is
```

```
    ** something for you or there is nothing left to do it will return
a
    ** ULONG result. If this is WMHI_LASTMSG, there is nothing further
to
    ** process, else we have some work to do.
    */
    while((result = DoMethod(winObj, WM_HANDLEINPUT, &code)) !=
           WMHI_LASTMSG)
    {
        UWORD menuNum, itemNum;
        struct MenuItem *item;

        /* WMHI_CLASSMASK allows us to determine the message class from
       ** the result.
        */
        switch(result & WMHI_CLASSMASK)
        {
            case WMHI_MENUPICK:
                /* One or more menu items have been selected. Masking
               ** result with WMHI_MENUMASK gives us the selected menu
               ** item which is then decoded in the same way as you do
               ** under Intuition.
                */
                selection = (result & WMHI_MENUMASK);

                /* Although there is only one item to select, the menu
                ** processing code below allows for multiple menu
               ** selection should other menu items be added.
                */
                while(selection != MENUNULL && !done)
                {
                    menuNum = MENUNUM(selection);
                    itemNum = ITEMNUM(selection);
                    item = ItemAddress(win->MenuStrip, selection);

                    switch(menuNum)
                    {
                        case 0: /* File menu */
                            switch(itemNum)
                            {
                                case 0: /* Quit */
                                    done = TRUE;
                                    break;
                            }
                            break;

                        default:
                            break;
                    }

                    /* Essential for processing more than one menu
                    ** selection
                    */
                    selection = item->NextSelect;
                }
                break;
        }
    }
}

/* Reaction method of closing the window. The object still needs to be
** destroyed, however.
```

```
                    */
                    DoMethod(winObj, WM_CLOSE);

                    /*
                    ** This cleans up the window object. It will also dispose of any
gadgets
                    ** still attached. In this case there is just the layout gadget. The
              menu
                    ** is also properly disposed of. !!! This means there is no need to call
                    ** DisposeObject() on layoutGad !!!
                    */
                    DisposeObject(winObj);
                }

                /* Close out the rest of the opened resources and exit. */
                CloseLibrary(LayoutBase);
            }
            CloseLibrary(WindowBase);
        }
        CloseLibrary((struct Library *)IntuitionBase);
    }
    return 0;
}
```

User Data

As with GadTools menus, you can also store a 32-bit value of your choosing with each menu item. This could, for example, be a pointer to a much larger amount of data to be associated with the menu item.

Reaction offers you more options to use your stored 32-bit value. When creating the window object you can also supply the **WINDOW_MenuUserData** tag to advise Reaction how to handle the user data supplied with each menu item.

The options are:

Tag Data	Function
WGUD_HOOK	UserData is a pointer to a Hook (struct Hook *).
WGUD_FUNC	UserData is a pointer to a function.
WGUD_IGNORE	UserData will not be interpreted by Reaction.

By default, **WINDOW_MenuUserData** is set to **WGUD_IGNORE** meaning that Reaction will not interpret it in any way and it can be used the same way as it was under GadTools in earlier releases.

If **WINDOW_MenuUserData** is set to **WGUD_HOOK**, the hook will be called via Utility Library's **CallHookPkt()** function each time a menu item is selected which allows you to perform custom processing on the item's selection. Your callback hook function will be called with the following signature (the object is the Reaction Window Object):

*ULONG myHookFunc(struct Hook *h, Object *wo, struct IntuiMessage *imsg);*

As per usual hook calling convention, h will be in A0, wo will be in A2, and imsg will be in A1.

NOTE: If WINDOW_MenuUserData is set to WGUD_HOOK then ALL menu items will have the UserData field called as a Hook so you will need to ensure that they ALL have valid Hook pointers assigned.

If **WINDOW_MenuUserData** is set to **WGUD_FUNC**, the function will be called directly each time a menu item is selected, allowing you to perform custom processing on the item's selection. Your callback function will be called with the following signature (the object is the Reaction Window Object):

*ULONG myFunc(Object *wo, struct IntuiMessage *imsg);*

The wo argument will be supplied in A0, and imsg will be in A1.

NOTE: If WINDOW_MenuUserData is set to WGUD_FUNC then ALL menu items will have the UserData field called as a callback function so you will need to ensure that they ALL have valid function pointers assigned.

The preferred option is to use a Hook call instead for future compatibility, and also compatibility with AmigaOS 4.x.

New and Updated Menu Functions

Function	Min Version	Description
LendMenus()	**39**	Lend a windows menu action to another window.
OpenWindow()	30	**V39** - Adds support for NewLook Menus.
OpenWindowTagList()	36	**V39** - Adds support for NewLook Menus
CreateMenus()	36	**V39** - Adds support for Image Objects.
LayoutMenuItems()	36	**V39** - Accepts new tags: GTMN_FrontPen GTMN_NewLookMenus GTMN_Checkmark GTMN_AmigaKey
LayoutMenus()	36	**V39** - Accepts new tags: GTMN_FrontPen GTMN_NewLookMenus GTMN_Checkmark

Chapter 12

BITMAPS AND RTG

THE COVETED TECHNICAL REFERENCE ADDENDUM

ROM Kernel Reference Manual

CHANGES & ADDITIONS

The Amiga computer was launched in 1985 and the high performance microcomputer dazzeled the world with superb graphics, sound, multiwindow and multitasking capabilities. 36 years later many of the concepts the Amiga introduced impressively still holds. And the retro community is still embracing this platform producing breathtaking new software and hardware.

Written by developers with intricate knowledge of the Amiga system software, the ***ROM Kernel Reference Manual: Changes & Additions*** provides tutorials and examples showing the changes and additions to the Amiga's system software. This coveted addendum to the old Amiga Technical Reference Series tries to pick up where the original series left off covering every change since release 2. It includes:

- Complete coverage of the BOOPSI classes previously known as ClassAct or ReAction
- Details of graphics programming in an AGA and RTG world
- A full overview of everything that has changed since Release 2

For the serious Amiga developer who wants to create up-to-date software for the amazing retro computing platform, the ***ROM Kernel Reference Manual: Changes & Additions*** is the definitive source of information on how to use all the changes and additions the Amiga system software has gained since Release 2

It is not an official continuation of the original Amiga Technical Reference series, but it tries to stay true to the spirit of the original series. You should still read the original series as it provides the backstory to many of the topics covered in this book.