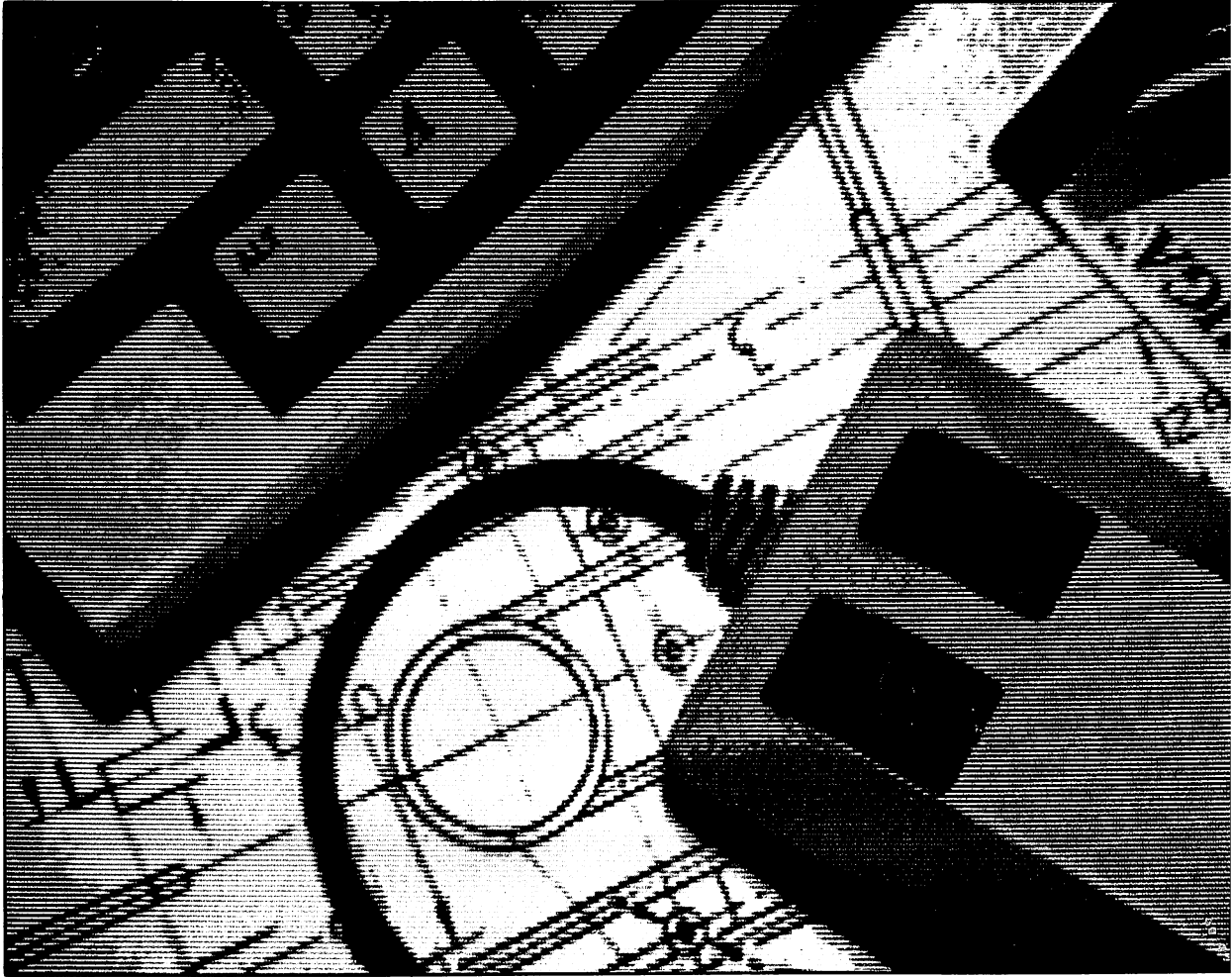AMIGA™

Intuition

The AMIGA User Interface

INTUITION

THE AMIGA USER INTERFACE

=ROBERT J. MICAL=

AND

SUSAN DEYL

COMMODORE-AMIGA, INC.

PREFACE

This book provides information about the Intuition user interface, and is intended for people who want to use this interface to write application programs for the Amiga. Some familiarity with C language programming is assumed.

The first two chapters of this book are introductory in nature. Each of the next nine chapters concentrates on some aspect of Intuition, and these chapters are organized in the same way. First, there is a complete description of the component in general terms. The second part of each chapter contains complete specifications for the data structure of the component and a brief summary of the function calls that affect the component. The last chapter contains some important programming style guidelines.

Here is a brief overview of the contents of each chapter:

Chapter 1, *Introduction.* A brief overview of the implementation and goals of the user interface, how the user sees an Intuition application, and an approach to using Intuition.

Chapter 2, *Getting Started with Intuition.* A summary of Intuition components and a sample program that shows the header files, how to access the Intuition library, and some fundamental Intuition structures.

Chapter 3, *Screens.* Discussion of the fundamental display component of Intuition. How to use the standard screens and how to design and use screens of your own.

Chapter 4, *Windows.* Description of the windows through which applications carry out their input and output. How to define and open windows according to the needs of your application.

Chapter 5, *Gadgets.* The multi-purpose input devices that you can design and attach to your screens, windows, requesters, and alerts.

Chapter 6, *Menus.* Designing the menu items that Intuition forms into a complete menu system for your window. How the user's choices of commands and options are transmitted to the application.

Chapter 7, *Requesters and Alerts.* Description and instructions for using the requesters, information exchange boxes that block input to the window until the user responds. How to use the alerts, which are emergency communication devices.

Chapter 8, *Input and Output Methods.* When and how to use the Message Port for input and the Console Device for input and output. How to use the message port messages.

Chapter 9, *Images, Line Drawing, and Text.* Using the Intuition graphics, border and text structures and functions. Introduction to using the general Amiga graphics facilities in Intuition applications.

Chapter 10, *Keyboard and Mouse.* Using the input from the keyboard and mouse (or other controller).

Chapter 11, *Other Features.* Information about the Preferences program, features that affect the entire display, and notes for assembly language programmers.

Chapter 12, *Style.* Guidelines and cautions for making the interface consistent and easy to use.

Appendix A contains a complete description of each Intuition function.

Appendix B contains the Intuition include file.

Appendix C contains some internal Intuition procedures for advanced users.

The *Glossary* contains definitions of all the important terms used in the book.

You will find related information in the following Amiga manuals:

- o *AmigaDOS Reference Manual*
- o *AmigaDos User's Manual*
- o *AmigaDos Technical Reference Manual*
- o *Amiga ROM Kernel Manual*

INTUITION

Table of Contents

INTUITION

List of Figures

Chapter 1

INTRODUCTION

Welcome to Intuition, the Amiga user interface.

What is a user interface? This sweeping phrase covers all aspects of getting input from and sending output to the user. It includes the innermost mechanisms of the computer and rises to the height of defining a philosophy to guide the interaction between man and machine. Intuition is, above all else, a philosophy turned into software.

This user interface philosophy is simple to describe: the interaction between the user and the computer should be simple, enjoyable, and consistent; in a word, intuitive. Intuition supplies a bevy of tools and environments which can be used to meet this philosophy.

Intuition was designed with two major goals in mind. The first is to give users a convenient, constant, colorful interface with the functions and features of both the Amiga operating system and the programs that run in it. The other goal is to give application designers all the tools they need to create this colorful interface, and to free them of the responsibility of worrying about any other programs that may be running at the same time, competing for the same display and resources.

The Intuition software manages a many-faceted windowing and display system for input and output. This system allows full and flexible use of the Amiga's powerful multi-tasking, multi-graphic, and sound synthesis capabilities. Under the Amiga Executive operating system, many programs can reside in memory at the same time and share the system's resources with one another. Intuition allows these programs to display their information in overlapping windows without interfering with each other, and provides an orderly way for the user to decide which program to work with at any given instant, and how to work with that program.

Intuition is implemented as a library of C-language functions. These functions are also available to other high-level language programmers and to assembly-language programmers via alternate interface libraries. Application programmers use these routines along with simple data structures to generate program displays and interface with the user.

A program can have full access to all the functions and features of the machine by opening its own *virtual terminal*. Upon opening a virtual terminal, your program will seem to have the entire machine and display to itself. It may then display text and graphics to its terminal, and it may ask for input from any number of sources, while ignoring the fact that any number of other programs may be performing these same operations. In fact, your program can open several of these virtual terminals and treat each one as if it were the only program running on the machine.

The user sees each virtual terminal as a *window*. Many windows can appear on the same display. Each window can be the virtual terminal of a different application program, or several windows can be created by the same program.

The Amiga also gives you extremely powerful graphics and audio tools for your applications. There are many display modes and combinations of modes (for instance, four display resolutions, Hold-and-Modify mode, dual-playfield mode, different color palettes, double-buffering, and more) plus animation, speech and music synthesis. You can combine sound, graphics, and animation in your Intuition windows. As you browse through this book, or peruse it carefully,

you'll find lots of creative ways to turn Intuition and the other Amiga tools into your very own personal kind of interface.

How the User Sees an Intuition Application

From the user's viewpoint, the Amiga environment is colorful and graphic. Application programs can use graphics as well as text in the windows, menus, and the other display features described below. You can make liberal use of *icons* (small graphic objects symbolic of an option, command, or object such as a document or program) to help make the user interface clear and attractive.

The user of an Amiga application program, or of the AmigaDOS operating system, sees the environment through windows, each of which can represent a different task or context. Each window provides a way for the user and the program to interact. This kind of user interface minimizes the context the user must remember. The user manipulates the windows, *screens* (the background for windows), and the contents of the windows with a mouse or other controller. At his or her convenience, the user can switch back and forth between different tasks, such as coding programs, testing programs, editing text, and getting help from the system. Intuition remembers the state of partially completed tasks while the user is working on something else.

The user can change the shape and size of these windows, move them around on the screen, bring a window to the foreground, and send a window to the background. By changing the arrangement of the windows, the user can select which information is visible and which terminal will receive input. While the user is shaping and moving the windows around the display, your program can ignore the changes. As far as the application is concerned, its virtual terminal covers the entire screen, and outside of the virtual terminal there's nothing but a user with a keyboard and a mouse (and any other kind of input device, including joysticks, graphics tablets, light pens, and music keyboards).

Screens can be moved up or down in the display, and moved in front of or behind other screens. In the borders of screens and windows there are control devices, called *gadgets*, that allow the user to modify the characteristics of screens and windows. For instance, there is a gadget for changing the size of a window and a gadget for depth arrangement of the screens.
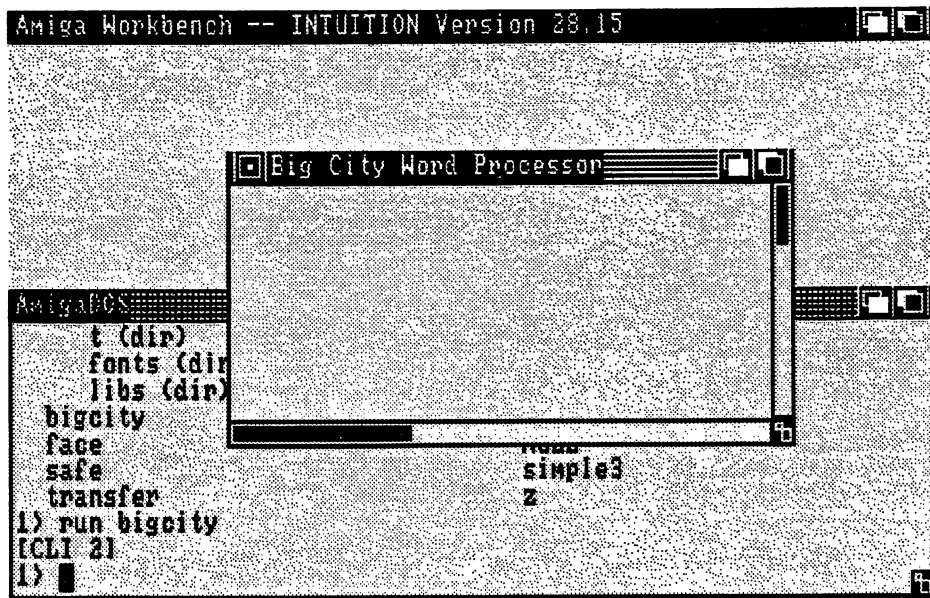
Figure 1-1: A Screen With Windows

Applications can use a variety of custom gadgets. For example, the program might use a gadget to request that the user type in a string of characters. Another gadget might be used to adjust the sound volume or the color of the screen.

At any time, only one window is active in the sense that only one window receives input from the user. Other windows, however, can work on some task that doesn't require input. For the active window, the screen's title bar can be used to display a list of menus (called the *menu bar*) at the user's command. By moving the mouse pointer along the menu bar, the user can view a list of menu items for each menu category on the menu bar. Each item in the list of menus can have its own sub-item list.
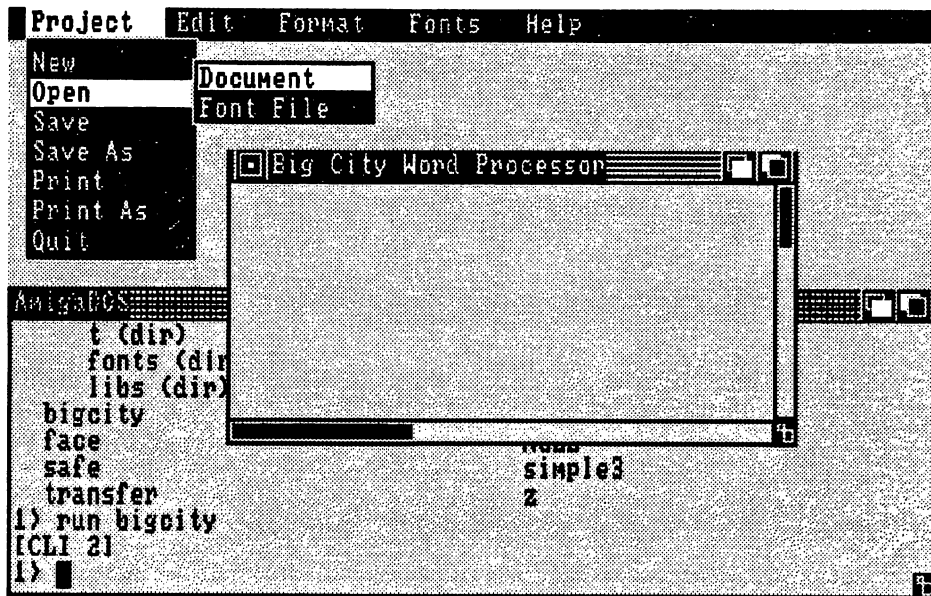
Figure 1-2: Menu Items and Sub-Items

Menus present lists of options and commands. The user can make choices from menus by using the mouse pointer and buttons. Applications can also provide the user with key-sequence shortcuts, as an alternative to the mouse. Intuition supplies certain key-sequence shortcuts automatically.

Windows can present the user with special *requester* boxes, invoked by the system or by applications. Requesters provide extended communication between the user and the application. When a requester is displayed, interaction with that window is halted until the user takes some action. The user, however, can make some other window active and deal with the requester later. If you wish, you can let the user bring up a requester on demand.
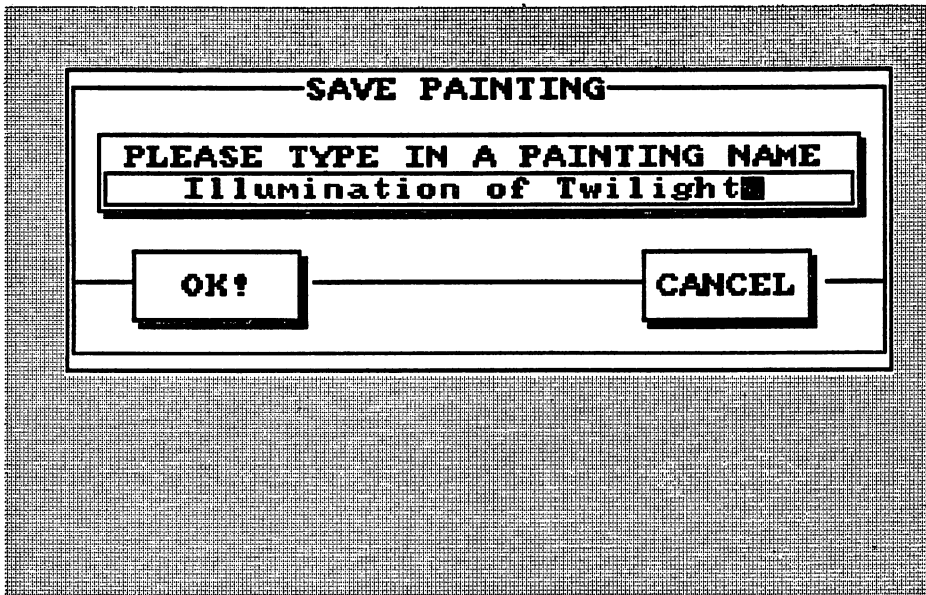
Figure 1-3: A Requester

The *alert* is another kind of special information exchange device invoked by the system or applications. The alert display is dramatic. It appears in red and black at the top of the display, with text and a blinking border. Alerts are meant for serious problems or when the user must take some action immediately. The application may also try to get the user's attention by flashing the screen or windows in a complementary color.

```
ALERT:  System Out of Memory Error
Press Left Button to Retry         Press Right Button to Abort
            Guru Meditation Number 0x07000000
```
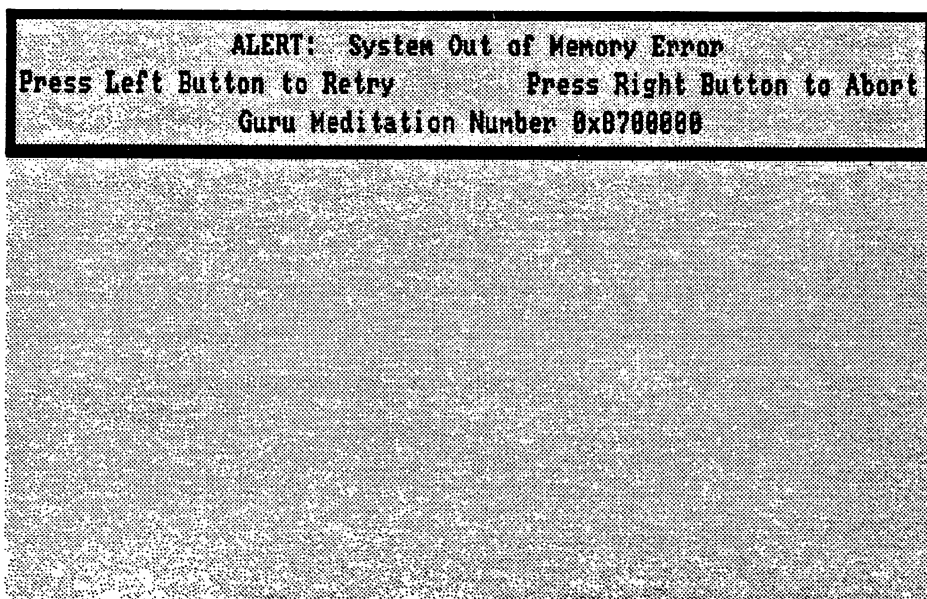
Figure 1-4: An Alert

The Right Approach to Using Intuition

Intuition is a very flexible program environment, with a vast number of features and defaults available to you. The tools and devices are well-defined and easily accessible. While there are many default values for you to rely on, there are few restrictions placed on you, so that your own creativity can flow. You are encouraged to take advantage of the many Intuition features. Doing so serves two purposes: you spend less time implementing user-interaction mechanisms of your own since Intuition already provides a wide range of them for you, and the user of your code gets to work in an environment that doesn't change radically from one application to another.

For example, you can define the windows for your program in one of the standard screens provided by Intuition. Then you can use the standard system requesters and gadgets and simple menu facilities. On the other hand, you can design a custom screen using your own choice of modes and colors. You can use Intuition's standard imagery for your windows and gadgets, or you can design completely custom graphics if you like. You can create your own pointer, and your menu items can combine elaborate graphic images and text strings. You can also choose to mix pre-defined features and custom designs. Your creative freedom is practically limitless under Intuition.

No matter how simple or fanciful your program design, it will fit within the basic Intuition framework of windows and screens, gadgets, menus, requesters and alerts. The users of the Amiga will come to understand these basic Intuition elements, and will come to trust that the building blocks remain constant. This consistency ensures that a well-designed program will be

understandable to the naive user as well as the sophisticate. This is the essence and beauty of the Intuition philosophy.

Chapter 2

GETTING STARTED WITH INTUITION

Chapter 1 gave you an overview of how the Intuition components work together. This chapter contains a quick overview of Intuition's components and a simple sample program demonstrating how to open an Intuition window and screen.

Intuition Components

Intuition's major components are summarized in the following list.

o   Windows provide the means for obtaining input from the user and the normal destination for the program's output.

o   Screens provide the background for opening windows.

o   The mechanisms for interaction between users and applications are:

   *   Menus present users with options and give them an easy way of entering commands.

   *   Requesters provide a menu-like exchange of information.

   *   Gadgets are the main method of communication.

   *   Alerts are for emergency communications.

   *   The mouse is the user's primary tool for making selections and entering commands.

   *   The keyboard is used for entering text and as an alternate shortcut method of entering commands.

   *   Other input devices, like graphics tablets or music keyboards, provide additional means of user input.

o   The methods of program input and output are:

   *   Input through the console device or Intuition Direct Communication Message Ports (known as the IDCMP)

   *   Output through the console device or directly to the graphics, text, and animation library functions

General Program Requirements and Information

This section introduces the basic requirements for an Intuition application by showing you how to create a very simple program, which involves the following.

o   You must include the necessary *header* files. Header files contain all of the definitions of data types and structures, constants, and macros.

o   Because Intuition is implemented as a library, you must declare a pointer variable named *IntuitionBase* and call *OpenLibrary()* before you can use any of the Intuition functions.

o   You open a window by initializing the data of a NewWindow structure and then calling *OpenWindow()* with a pointer to that structure.

o   You open a screen by initializing the data of a NewScreen structure and then calling *OpenScreen()* with a pointer to that structure.

o   Finally, in the example we write some simple text to a window, demonstrating how simple it is to use the graphics library with your window.

Simple Program: Opening a Window

First, here is the simplest program, which does nothing more than open a very plain window:

```
/******************************************************************************
 *
 * Simple OpenWindow() program
 *
 ******************************************************************************/

#include <exec/types.h>
#include <intuition/intuition.h>
struct IntuitionBase *IntuitionBase

#define INTUITION_REV   29    /* You must be sure this is correct */
#define MILLION         1000000

main()
{
    struct NewWindow NewWindow;
    struct Window *Window;
    LONG i;

    /* Open the Intuition library.  The result returned by this call is
     * used to connect your program to the actual Intuition routines
     * in ROM.  If the result of this call is equal to zero, something
     * is wrong and the Intuition you requested is not available, so
     * your program should exit immediately
     */
    IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", INTUITION_REV);
    if (IntuitionBase == NULL) exit(FALSE);

    /* Initialize the NewWindow structure for the call to OpenWindow() */
    NewWindow.LeftEdge = 20;
    NewWindow.TopEdge = 20;
    NewWindow.Width = 300;
    NewWindow.Height = 100;
    NewWindow.DetailPen = 0;
    NewWindow.BlockPen = 1;
    NewWindow.Title = "A Simple Window";
    NewWindow.Flags = SMART_REFRESH | ACTIVATE;
    NewWindow.IDCMPFlags = NULL;
    NewWindow.Type = WBENCHSCREEN;
    NewWindow.FirstGadget = NULL;
    NewWindow.CheckMark = NULL;
    NewWindow.Screen = NULL;
    NewWindow.BitMap = NULL;
    NewWindow.MinWidth = 0;
    NewWindow.MinHeight = 0;
    NewWindow.MaxWidth = 0;
    NewWindow.MaxHeight = 0;

    /* Try to open the window.  Like the call to OpenLibrary(), if
```

```
 * the OpenWindow call is successful, it returns a pointer to
 * the structure for your new window.  If the OpenWindow() call
 * fails, it returns a zero.
 */
if (( Window = (struct Window *)OpenWindow(&NewWindow) ) == NULL)
    exit(FALSE);

/* Do nothing a million times.  How long do you think it will take for
 * the Amiga to do nothing a million times?  Try it and see!
 */
for (i = 0; i < MILLION; i++) ;

/* Finally, close the Window, and then exit */
CloseWindow(Window);
}
```

See how easy it is to create a window under Intuition? This is a complete program, which can be compiled as is. If run, this program would open the window shown in Figure 2-1.
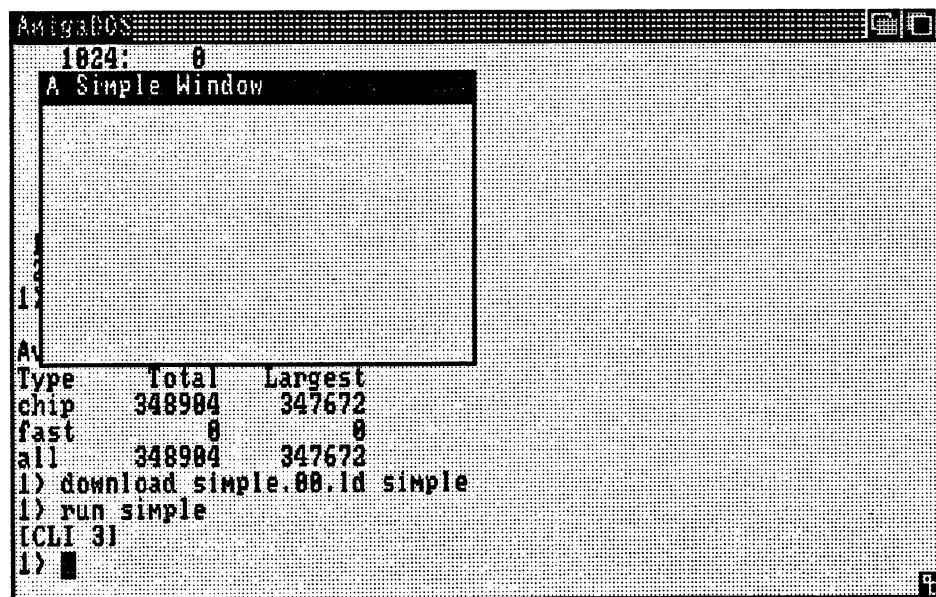
Figure 2-1: A Simple Window

The *Type* variable in NewWindow describes the screen-type you want for this window. With the *Type* variable set to WBENCHSCREEN, you are specifying to Intuition that you wish this window to open in the Workbench screen. With the *Flags* variable initialized to SMART_REFRESH and ACTIVATE, you are specifying that you want this window to take advantage of Intuition's SMART_REFRESH mode of window display and you want this window to become the active window when it opens. The rest of the NewWindow variables are set to simple, safe default values. Consequently, there's not much you can do with this window.

Simple Program: Adding the Close Gadget

So let's make a slightly fancier window. Let's change the window so that you can close it when you like, rather than having it close automatically.

First, we'll ask Intuition to attach a WINDOWCLOSE gadget Then, we'll ask Intuition to tell our program when someone activates that WINDOWCLOSE gadget.

To ask for the WINDOWCLOSE gadget, change the *Flags* variable to include the WINDOW-CLOSE flag:

NewWindow.Flags = WINDOWCLOSE | SMART_REFRESH | ACTIVATE;

Now, to have Intuition tell us that the gadget has been activated requires several things. We must:

o   tell Intuition that we want to know about the event;

o   wait for the event to happen;

o   close the window and exit.

We instruct Intuition to tell us about the event by specifying one of the event flags in the *IDCMPFlags* variable. The *IDCMP*, to remind you, is Intuition's Direct Communications Message Port system. By setting one of the IDCMP flags, we are requesting Intuition to open a pair of message ports through which we may communicate.

/* Tell us about CLOSEWINDOW events */
NewWindow.IDCMPFlags = CLOSEWINDOW;

Finally, rather than counting to a million and then closing the window, here's how we wait for the CLOSEWINDOW event:

Wait(1 << Window->UserPort->mp_SigBit);
CloseWindow(Window);
exit(TRUE);

The variables *UserPort* and *mp_SigBit* are initialized for you by Intuition, so you can ignore these for now. *Wait()* is a function of the Amiga Executive. It allows your program to do absolutely nothing, and thereby free up the processor for other jobs, until some special event occurs which is of the right type to wake you up again. In our very simple example, we've asked for only one type of event to wake us up, so when we are awoken again we can automatically assume that it was because someone pressed the WINDOWCLOSE gadget of our window. When we start using the IDCMP for more elaborate functions, we will have to get a message from the message port and examine it to see what event has occurred to awaken us.

Simple Program: Adding the Rest of the System Gadgets

Next, try attaching all of the system gadgets to your window:

NewWindow.Flags = WINDOWCLOSE | SMART_REFRESH | ACTIVATE
| WINDOWDRAG | WINDOWDEPTH | WINDOWSIZING | NOCAREREFRESH;

The WINDOWDRAG flag means that you want to allow the user to drag this window around the screen. If you don't set this flag, as in our previous example, then the window cannot be moved.

The WINDOWDEPTH flag specifies that the user can depth-arrange this window with other windows in the same screen. Having set this flag, you can now send this window behind all other windows, or bring this window in front of all other windows.

The WINDOWSIZING flag means that you want to allow the user to change the size of this window. This has two implications. First, resizing a window can sometimes require even SMART_REFRESH windows to be refreshed (to refresh a window means to redisplay the information contained in that window). If you *really* don't care about whether you should refresh your window (in this case we don't care), then you can set the NOCAREREFRESH flag and Intuition will take care of all the refresh details for you. Because we're allowing our window to be sized now, we have to tell Intuition about the minimum and maximum sizes for our window:

NewWindow.MinWidth = 100;
NewWindow.MinHeight = 25;
NewWindow.MaxWidth = 640;
NewWindow.MaxHeight = 200;

If you run the program with these changes, your window will now look like Figure 2-2.
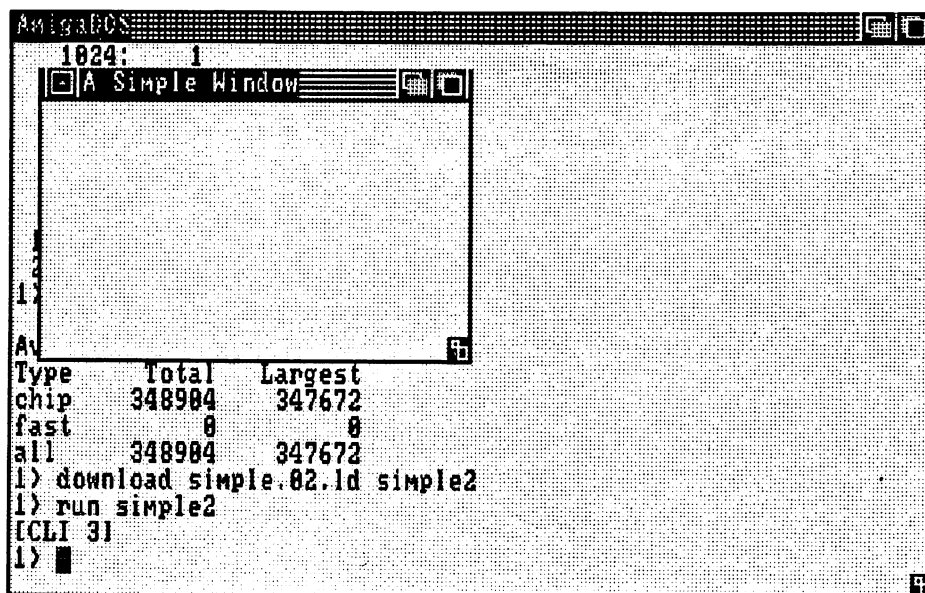
Figure 2-2: A Simple Window With Gadgets

Simple Program: Opening a Custom Screen

To open a custom screen, we must initialize a struct NewScreen block of data, and then call *OpenScreen()* with a pointer to that data.

To open the window, we initialized a NewWindow structure by writing a long series of assignment statements. There is a more compact method for initializing a structure. We will use this shorter method to illustrate opening a custom screen.

```
/* This font declaration will be used for our new screen */
struct TextAttr MyFont =
    {
    "topaz.font",           /* Font Name */
    TOPAZ_SIXTY,            /* Font Height */
    FS_NORMAL,              /* Style */
    FPF_ROMFONT,            /* Preferences */
    };

/* This is the declaration of a pre-initialized NewScreen data block.
 * It often requires less work, and often uses less code space, to
 * pre-initialize data structures in this fashion.
 */
struct NewScreen NewScreen =
    {
    0,                      /* the LeftEdge should be equal to zero */
    0,                      /* TopEdge */
    320,                    /* Width (low-resolution) */
    200,                    /* Height (non-interlace) */
    2,                      /* Depth (16 colors will be available) */
    0, 1,                   /* the DetailPen and BlockPen specifications */
    NULL,                   /* no special display modes */
    CUSTOMSCREEN,           /* the Screen Type */
    &MyFont,                /* use my own font */
    "My Own Screen",        /* this declaration is compiled as a text pointer */
    NULL,                   /* no special Screen Gadgets */
    NULL,                   /* no special CustomBitMap */
    };
```

Here's how we open the screen:

```
if (( Screen = (struct Screen *)OpenScreen(&NewScreen) ) === NULL)
    exit(FALSE);
```

Because we're now opening our window in a custom screen, we have to change our initialization of the NewWindow data slightly. We have to change the *Type* declaration from WBEN-CHSCREEN to CUSTOMSCREEN. Also, we previously set the *Screen* variable to NULL, since we had wanted our window to open in the Workbench screen. Now, we have to set this variable to point to our new custom screen:

```
NewWindow.Type = CUSTOMSCREEN;
NewWindow.Screen = Screen;
```

After we close our window, let's close our screen, too:

```
CloseScreen(Screen);
```

Our program now opens a custom screen and opens a window in that screen.

Simple Program: The Final Version

For a finishing touch, let's write a bit of text to our window. This will require another declaration, another call to *OpenLibrary()*, and a call to the graphics library's *Move()* and *Text()* functions:

```
struct GfxBase *GfxBase;
    GfxBase = OpenLibrary("graphics.library", GRAPHICS_REV);
    Move(Window->RPort, 20, 20);
    Text(Window->RPort, "Hello World", 11);
```

All together, our program looks like the following.

```
/******************************************************************************
 *
 * "Hello World"
 *
 ******************************************************************************/

#include <exec/types.h>
#include <intuition/intuition.h>

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

#define INTUITION_REV   29
#define GRAPHICS_REV    29
#define MILLION         1000000


struct TextAttr MyFont =
    {
    "topaz.font",           /* Font Name */
    TOPAZ_SIXTY,            /* Font Height */
    FS_NORMAL,              /* Style */
    FPF_ROMFONT,           /* Preferences */
    };

/* This is the declaration of a pre-initialized NewScreen data block.
 * It often requires less work, and often uses less code space, to
 * pre-initialize data structures in this fashion.
 */
struct NewScreen NewScreen =
    {
    0,                      /* the LeftEdge should be equal to zero */
    0,                      /* TopEdge */
    320,                    /* Width (low-resolution) */
    200,                    /* Height (non-interlace) */
    2,                      /* Depth (16 colors will be available) */
    0, 1,                   /* the DetailPen and BlockPen specifications */
    NULL,                   /* no special display modes */
    CUSTOMSCREEN,           /* the Screen Type */
    &MyFont,                /* use my own font */
    "My Own Screen",        /* this declaration is compiled as a text pointer */
    NULL,                   /* no special Screen Gadgets */
    NULL,                   /* no special CustomBitMap */
    };


main()
{
    struct Screen *Screen;
    struct NewWindow NewWindow;
```

```
      struct Window *Window;
      LONG i;

      /* Open the Intuition library. The result returned by this call is
       * used to connect your program to the actual Intuition routines
       * in ROM. If the result of this call is equal to zero, something
       * is wrong and the Intuition you requested is not available, so
       * your program should exit immediately
       */
      IntuitionBase = (struct IntuitionBase *)
          OpenLibrary("intuition.library", INTUITION_REV);
      if (IntuitionBase == NULL) exit(FALSE);

      GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", GRAPHICS_REV);
      if (GfxBase == NULL) exit(FALSE);

      if ((Screen = (struct Screen *)OpenScreen(&NewScreen)) == NULL)
          exit(FALSE);

      NewWindow.LeftEdge = 20;
      NewWindow.TopEdge = 20;
      NewWindow.Width = 300;
      NewWindow.Height = 100;
      NewWindow.DetailPen = 0;
      NewWindow.BlockPen = 1;
      NewWindow.Title = "A Simple Window";
      NewWindow.Flags = WINDOWCLOSE | SMART_REFRESH | ACTIVATE
          | WINDOWSIZING | WINDOWDRAG | WINDOWDEPTH;
      NewWindow.IDCMPFlags = CLOSEWINDOW;
      NewWindow.Type = CUSTOMSCREEN;
      NewWindow.FirstGadget = NULL;
      NewWindow.CheckMark = NULL;
      NewWindow.Screen = Screen;
      NewWindow.BitMap = NULL;
      NewWindow.MinWidth = 100;
      NewWindow.MinHeight = 25;
      NewWindow.MaxWidth = 640;
      NewWindow.MaxHeight = 200;

      if (( Window = (struct Window *)OpenWindow(&NewWindow) ) == NULL)
          exit(FALSE);

      Move(Window->RPort, 20, 20);
      Text(Window->RPort, "Hello World", 11);

      Wait(1 << Window->UserPort->mp_SigBit);
      CloseWindow(Window);
      CloseScreen(Screen);
      exit(TRUE);
}
```

The display created by the final version looks like Figure 2-3.
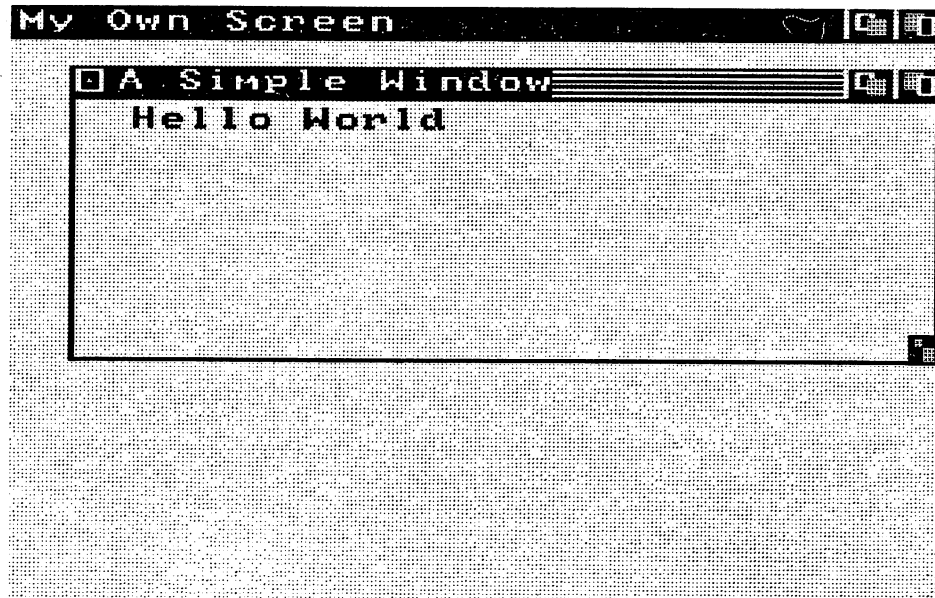
Figure 2-3: Intuition's " Hello World" Program

Chapter 3

SCREENS

We begin the discussion of Intuition components with screens, because they are the basis for all Intuition displays. They set up the environment for overlapping windows and they give you easy access to all the Amiga display modes and graphics features. In this chapter you will learn how to use the standard screens provided by Intuition and how to create your own custom screens.

The first section of this chapter is a general description of screens, including their characteristics and their place in the overall Intuition scheme.

The next section describes the standard screens, including the Workbench. Any program can open its windows in one of the standard screens.

The largest section of this chapter concerns custom screens. You will want to use a custom screen if none of the standard screens supplies display modes that match your needs, or if you want to do anything with a screen's display other than simply open windows. In the last section you will find an overview of the process of creating and opening a custom screen, the specifications for the custom screen structure, and all the functions you use in working with custom screens.

About Screens

The screen is Intuition's basic unit of display. By using an Intuition screen, you can create a video display with any combination of the many Amiga display modes. Certain basic parameters of the video display (such as fineness of vertical and horizontal resolution, number of colors, and color choices) are defined by these modes. By combining modes, you can have many different types of displays. For example, the display may show 8 different colors of low-resolution pixels, or 32 colors in interlace mode (high resolution of lines). For a description of all the different display modes, see the "Custom Screens" section below.

Every other Intuition display component is defined with respect to the screen in which it is created. Each screen's data structure contains definitions that describe the modes for the particular screen. Windows inherit their display parameters from the screens in which they open, so a window that opens in a given screen always has the same display modes and colors as that screen. If your program needs to open windows that differ from each other in their display characteristics, you can open more than one screen.

Screens are always the full width of the display. This is because the Amiga hardware allows very flexible control of the video display, but imposes certain minor restrictions. Sometimes it is not possible to change display modes in the middle of a scan line. Even when it is possible, it is usually aesthetically unpleasant or visually jarring to do so. To avoid these problems, Intuition imposes its own display restriction that allows only one screen (one collection of display modes)

per video line. Because of this, screens can be dragged vertically but not horizontally. This allows screens with different display modes to overlap, but prevents any changes in display mode within a video line. This is contrasted with windows which (as you will see in the next chapter) can be any width, can overlap in any way, and can be dragged vertically and horizontally.

Screens provide display memory, which is the RAM where all imagery is first rendered and then translated by the hardware into the actual video display. The Amiga graphics structure that describes how rendering is done into display memory is called a *RastPort*. The RastPort also has pointers into the actual display memory locations. The screen's display memory is also used by Intuition for windows and other high-level display components that overlay the screen. Application programs that open custom screens can use the screen's display memory in any way they choose.

Screens are rectangular in shape, and when they first open they usually cover the entire surface of the video display, although they can be shorter than the height of the display. Like windows, the user can drag screens up or down and depth-arrange them by using special control mechanisms called gadgets. Unlike windows, the user can't change the size of screens and can't drag screens left or right.

The dragging and depth-arrangement gadgets come with all Intuition screens. These gadgets reside in the title bar at the top of the screen. Also in the title bar there may be a line of text identifying the screen and its windows. You can have other gadgets besides the ones that are automatically supplied with every screen. You can place gadgets of your own anywhere in the screen's title bar.

Figure 3-1 shows a screen with open windows. The depth-arrangement gadgets (front gadget and back badget) are at the extreme right of the screen title bar. The drag dadget occupies the entire area of the screen title bar not occupied by other gadgets. The user changes the front-to-back order of the displayed screens by using a controller (such as a mouse) or the keyboard cursor control keys to move the Intuition pointer within one of the depth-arrangement gadgets. When the user clicks the left mouse button (known as the *select button*), the screen's depth arrangement is changed.

```
Amiga Workbench -- INTUITION Version 28.15                      ▣▯

                              New CLI▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓  ▣▯
                              New CLI task 3
                              3) snoop
                              Alloc/Free Memory Snooper #2
                              (does not use caller's stack
         AmigaDOS▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓ ▣▯
            frags                      RETURN) to quit...
            idemo
            prefs
            safe
            snoop
         1)
         Available Memory
         Type      Total    Largest
         chip      306888   301128
         fast           0        0
         all       306888   301128
         1)■
```
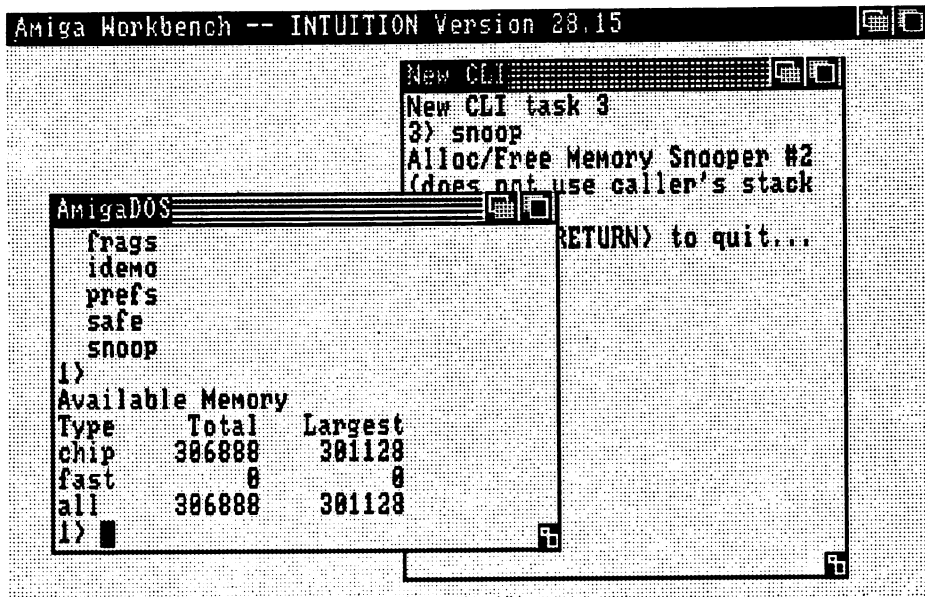
Figure 3-1: A Screen and Windows

The user drags the entire screen up or down on the video display by moving the pointer within the drag gadget, holding down the left mouse button while moving the pointer, and finally releasing the button when the screen is in the desired location.

The screen's title bar has many uses. It's also used to display a window's menus when the user asks to see them. Typically, when the user presses the right mouse button (known as the *menu button*), a list of menu topics called a menu list appears across the title bar. Figure 3-2 shows the same screen after the user has displayed the menu list.

```
 Colors    Special    Brushes
All rights √NOT QUITE THE EYEBALL ⌐R
Version 28 ALMOST THE EYEBALL   ⌐J
CLI Versic                      ⌐M
Use date t
Date DD-MM
Thursday 1
1> dir
      c (dir)
      1 (dir)      ⊡=>RJ<=     ⊞⊡
      devs (dir
      s (dir)
      t (dir)
      fonts (di
      libs (dir
   debug                       face
   idemo                       safe
   test                        transfer
1> run idemo
[CLI 2]
1> █
```
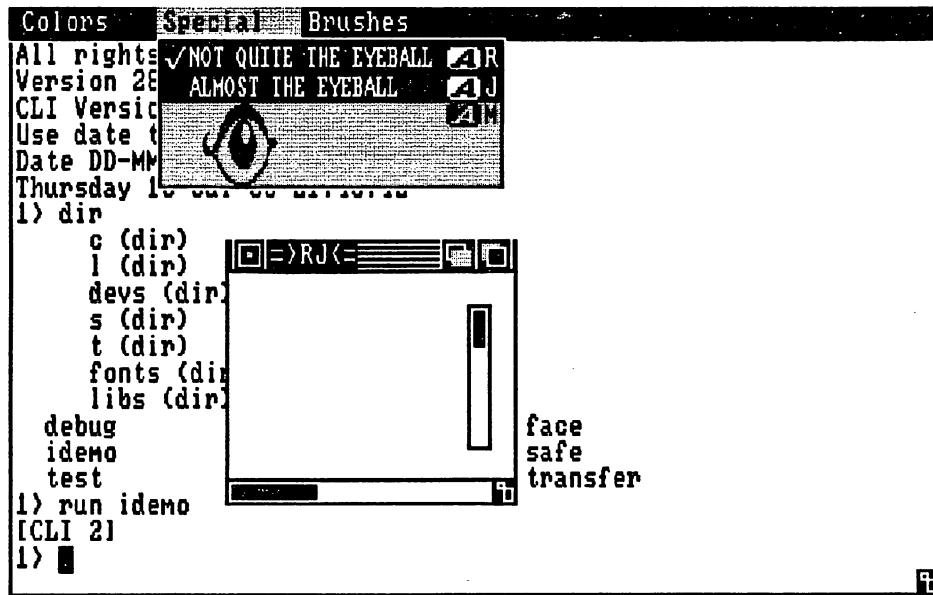
Figure 3-2: Screen and Windows with Menu List Displayed

By further mouse movement and mouse button manipulation, the user can see a list of menu items and sub-items for each of the topics in the menu list. The menu list, menu items, and sub-items that are displayed pertain to the currently active window, which is the window receiving the user's input. There is only one active window at any time. The screen containing the active window can be thought of as the active screen. Because there is only one active window, there can be only one active menu list at a time. The menu list appears on the title bar of the active screen. Menus are handled by the Intuition menu system. See Chapter 6, "Menus", for more information about how you put together menus and attach them to windows.

Both you and the user will find working with screens much like working with windows—for you, the data structures and the functions for manipulating screens and windows are similar. For the user, moving and arranging screens will require the same steps as moving and arranging windows. However, the user will be less aware of screens than of windows, since user input and application output occur mostly through windows.

There are two kinds of screens—standard screens supplied by Intuition and custom screens created by you. Standard screens are described in the next section, and the remainder of the chapter deals with custom screens.

Standard Screens

Standard screens differ from custom screens in three basic ways.

o Standard screens close and go away if all their windows go away. Only the Workbench standard screen (described below) differs in this regard. You can think of the Workbench as the default screen—if all its windows close, it remains open. When all other screens close, if the Workbench isn't already open it will open then.

o Standard screens are opened differenty. There is no function you call to explicitly open a standard screen. You simply specify the screen type in the window structure, and Intuition opens the screen if it's not already open. This is contrasted with custom screens, which you must explicitly open yourself before opening a window in the screen.

o You are free to design and change the characteristics of your custom screen practically any way you choose, but you should not change the colors, display modes, and other parameters of standard screens. These parameters have been pre-defined so that more than one application may open windows in a standard screen and be able to depend upon constant display characteristics. For instance, a business package that runs in a standard screen may expect the colors to be reasonable for a dither pattern in a graph. If you change the colors, then that program's graphics display will not be able to share the screen with you, thereby defeating the purpose of standard screens.

All of the Intuition standard screens are the full height and width of the video display area. Intuition manages standard screens and any program may open its windows in any of the standard screens. An application can display more than one window in a standard screen at the same time, and more than one application can open a window in a standard screen at the same time.

The standard screens currently available are:

o Workbench
o Others, as described in "Appendix B: Intuition Include File."

WORKBENCH

The Workbench is both a screen and an application.

The Workbench is the Intuition standard screen. It is a high-resolution (640 pixels x 200 lines) 4-color screen. The default colors are blue for the background, white and black for details, and orange for the cursor and pointer.
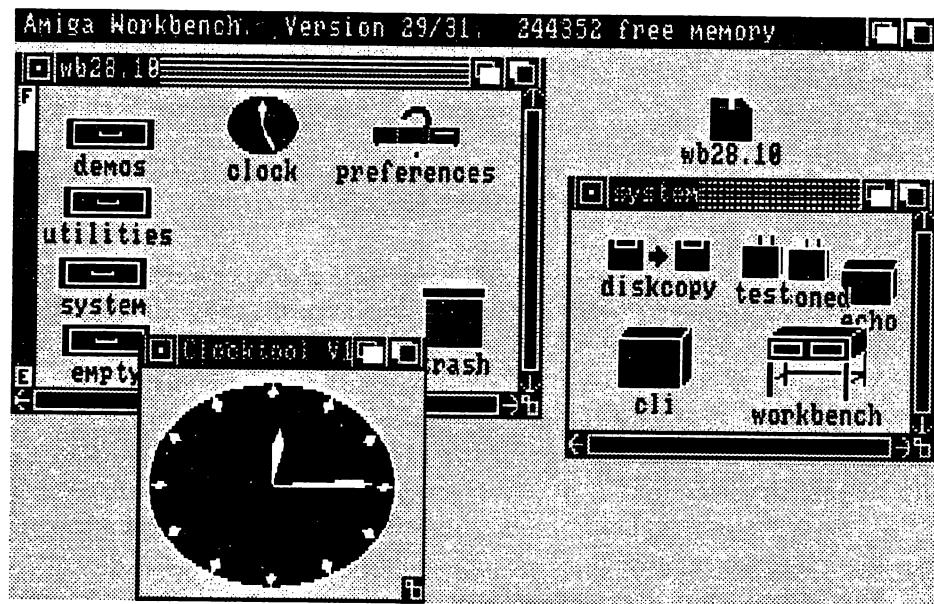
Figure 3-3: The Workbench Screen and the Workbench Application

The Workbench screen is used by both the Amiga Command Line Interface (CLI) and the Workbench tool. If you want to use the Workbench as a screen for the windows of your program, you just specify a window type of WBENCHSCREEN in the data structure called NewWindow, which you initialize when opening a window.

Any application program can use the Workbench screen for opening its windows. Developers of text-oriented applications are especially encouraged to open in the Workbench screen. This is convenient for the user because many windows will open in the same standard screen, requiring less movement between screens. Using the Workbench screen is also very memory-efficient because you won't be allocating the memory for your own custom screen.

The Workbench screen is the only one, besides your own custom screen, that you can explicitly close. Also the Workbench screen does not close when all the windows in it are closed, as do the other standard screens, and it automatically reopens when all other screens close down.

If your application needs more memory than what's available, it can attempt to reclaim the memory used by the Workbench screen by calling *CloseWorkBench()*. You should, however, call *OpenWorkBench()* as your program exits. It is good Intuition programming practice to always attempt to reopen the Workbench screen when your program is terminating, whether or not you called *CloseWorkBench()*. If all programs do this, it will help to present the user with as consistent and dependable interface as possible, since the Workbench screen will be available as much as possible.

The Workbench application program allows users to interact with the Amiga file system, using icons (small graphics images) to represent files. Intuition treats the Workbench application as a special case, communicating with it in extraordinary ways. For example, you can open or close the Workbench screen by calling the Intuition functions *OpenWorkBench()* and *CloseWorkBench()*, even though the Workbench tool may have open windows in the screen.

You have access to a body of library functions which allow you to create and manipulate the Workbench application's objects and iconography. The functions in the library allow you to

create disk files which the user can handle within the context of the Workbench program. For more information about the Workbench library, see the *AmigaDOS Developer's Manual.*

The user can change the colors of the Workbench screen via the Preferences program. For more information about Preferences, see Chapter 11, "Other Features".

Custom Screens

Typically, you create your own screen when you need a specific kind of display that's not offered as one of the standard screens, or when you want to modify the screen or its parameters directly—as in changing colors or directly modifying the *Copper* list or display memory. The Copper is the display-synchronized coprocessor that handles the actual video display by directly affecting the hardware registers. For example, you might need a display in which you can have movable sprite objects. Or, you might have your own display memory that you want to use for the screen's display memory. Or, you may want to allow the user to play with the colors of a display that you've created. If you want to do these sorts of things, you'll have to create a custom screen; such operations not allowed in standard screens.

Custom screens don't automatically close when all windows in them close (as system standard screens do, except for the Workbench). If you've opened a custom screen, you have to call *CloseScreen()* before your program exits. Otherwise, your screen stays around forever.

You can create two kinds of custom screens:

o   one that is entirely managed by Intuition, or

o   one where you use the Amiga graphics primitives to write directly into the display memory or otherwise directly modify the screen display characteristics. In this case, you have to take on some of the responsibility of managing the display.

These screen types are described in the following paragraphs.

INTUITION-MANAGED CUSTOM SCREENS

If you want this kind of custom screen, you still have a great deal of latitude in creating custom effects. You can set any or all of the following screen parameters:

o   Height of the screen and starting point of the screen when it first opens

o   Depth of the screen, which determines how many colors you can use for the display

o   Choice of the available colors for drawing details, such as gadgets, and doing block fills, such as the title bar area

o   Display modes—high or low resolution, interlace or non interlace, sprites, and dual playfields

o   Custom gadgetry

o   Initial display memory

You can also use the special Intuition graphics, line, and text structures and functions for rendering into windows in your custom screen. See Chapter 9, "Images, Line Drawing, and Text", for details about these.

APPLICATION-MANAGED CUSTOM SCREENS

In this kind of custom screen, you still use the same structures and functions. However, another dimension is added when you directly access the display memory. You can now use all of the Amiga graphics primitives to do any kind of rendering you want. You can do color animation, scrolling, patterned line drawing and patterned fills, and much, much more. Although you can still combine such a screen with other Intuition features like windows, menus, and requesters, certain interactions can trash the display. The interactions described in the next paragraph involve what happens when you write to the custom screen while windows and menus are being displayed and moved over the screen.

First, Intuition does not save background screen information when a window is opened, sized, or moved. Screen areas that are subsequently revealed are restored to a blank background color, obliterating any data you write into the display memory area of your screen. Second, menus are protected from data being output to the windows behind them but not from data being output to screens. When a menu is on the screen, all underlying windows are locked against graphical output to prevent such output from trashing the menu display. Menus cannot, however, lock graphical output to the display memory of a screen. So be very careful about writing to a screen that has or can have menus displayed in it. You can easily overstrike the menus and obliterate the information contained in them.

In summary, you must keep in mind that the user can modify the display by moving things around (by using window gadgets) or making things appear and disappear (menus and requesters). If you want to write directly to a custom screen's display memory, you have to design the pieces to interact together without conflict. It's not impossible to do, but you must be careful and think your design through in detail. If you want complete control of the screen display memory and are willing to give up some windowing capabilities (such as menus and window sizing and dragging), then you should use a custom screen. If you want to control the display memory *and* run windows and menus in the custom screen, then you need to deal with the hazards. Always bear in mind that playing with screen displays in this way requires an intricate knowledge of how screens and windows work, and you should not attempt it lightheartedly.

But what if you want a screen with your own display memory, one you can manipulate any way you choose, but you still want access to all the windowing and menu capabilities without worry? There is a special kind of window that satisfies all of these needs—the Backdrop window which always stays in the background and can be fashioned to fill the entire display area. Writing to this kind of window is almost as flexible as writing directly to display memory and requires only a little more overhead in memory image and performance. Menus and ordinary windows can safely reside over this window. You can also cause the screen's title bar to disappear behind a Backdrop window by calling the *ShowTitle()* function, thereby filling the entire video display with your display memory. This is the Intuition-blessed way to fill the entire display and still exist in an Intuition environment. For more information about setting up Backdrop windows, see Chapter 4, "Windows".

When you are using the graphics primitives (functions) to render in your custom screen, the functions sometimes require pointers to the graphics display memory structures that lie beneath the Intuition display. These graphics structures are the RastPort, ViewPort, and View. For more information and details about how to get the pointers into the display memory, see Chapter 9, "Images, Line Drawing, and Text".

Screen Characteristics

The following characteristics apply to both standard screens and custom screens. Keep in mind, however, that you should not change the characteristics of any of the standard screens.

DISPLAY MODES

You can use any or all of the following display modes in your custom screens. The windows that open in a screen inherit the screen's display modes and colors.

There are two modes of horizontal display: low-resolution and high-resolution. In low-resolution mode, there are 320 *pixels* across a horizontal line. In high resolution mode, there are 640 pixels across. A pixel is the smallest addressable part of the display and corresponds to one bit in a bit-plane. Twice as much data is displayed in high resolution mode. Low resolution mode gives you twice as many potential colors, 32 instead of 16.

There are two modes of vertical display: interlace and non-interlace. You can have 200 vertical lines of display in non-interlaced mode and 400 lines in interlaced mode. Twice as much data is displayed in interlaced mode. Typically, applications use non-interlaced mode, which requires half as much memory and creates a display that doesn't have the potential for flickering, as interlaced displays tend to do. Intuition supports interlace because some applications will want to use it; for instance, a computer-aided design package running on a high-phosphor-persistence monitor, will want to use it.

In *sprite* mode, you can have up to 8 small moving objects on the display. You define sprites with a simple data structure and move them by specifying a series of x,y coordinates. Sprites can be up to 16 bits wide and any number of lines tall, can have 3 colors (plus transparent), and pairs of sprites can be joined to create a 15-color (plus transparent) sprite. They're also reusable vertically so you can really have more than 8 at one time. The Amiga GELS system described in the *Amiga ROM Kernel Manual* provides just such a multi-plexing, or interleaving, of sprites for you. Chapter 4, "Windows", contains a brief description of a sprite used as a custom pointer.

Dual playfield mode is a special display mode where you can have two display memories. This gives you two separately controllable and separately scrollable entities that you can display at the same time, one in front of the other. With this mode, you can have some really interesting displays, because wherever the front display has a pixel which selects color register 0, that pixel is displayed as if it were transparent. You can see through these transparent pixels into the background display. In the background display, wherever a pixel selects color register 0, that pixel is displayed in whatever color is in color register 0.

Hold-and-modify mode gives you extended color selection.

If you want to use sprites, dual playfields, or hold-and-modify, you should read about all of their features in the *Amiga ROM Kernel Manual.*

DEPTH AND COLOR

Screen *depth* refers to the number of bit-planes in the the screen display. This affects the colors you can have in your screen and in the windows that open in that screen.

Display memory for a screen is made up of one or more of bit-planes, each of which is a contiguous series of memory words. When they're displayed, the planes are overlapped so that each pixel in the final display is defined by one bit from each of the bit-planes. For instance, each pixel in a 3-bit-plane display is defined by 3 bits. The binary number formed by these 3 bits specifies the color register to be used for displaying a color at that particular pixel location. In this case, the color register would be one of the 8 registers numbered 0 through 7. The 32 system color registers are completely independent of any particular display. You load colors into these registers by specifying the amounts of red, green and blue that make up the colors. To load colors into the registers, you use the graphics primitive *SetRGB()*. Table 3-1 shows the relationship between screen depth, number of possible colors in a display, and the color registers used.

Table 3-1: Screen Depth and Color

DEPTH	MAXIMUM NUMBER OF COLORS	COLOR REGISTER NUMBERS
1	2	0-1
2	4	0-3
3	8	0-7
4	16	0-15
5	32	0-31

The maximum number of bit-planes in a screen depends upon two of the display modes—dual playfields and hold-and-modify. For a normal display you can have from 1 to 5 bit-planes. For dual playfields, you can have from 2 to 6 bit-planes, which are divided between the two playfields. For hold-and-modify mode you need 6 bit-planes.

The color registers are also used for the "pen" colors. If you specify a depth of 5, for instance, then you also have 32 choices (in low-resolution mode) for the *DetailPen* and *BlockPen* fields in the structure. *DetailPen* is used for details such as gadgets and title bar text and *BlockPen* is used for block fills, like all of the title bar area not taken up by text and gadgets.

TYPE STYLES

When you open a custom screen, you can specify a text font for the text in the screen title bar and the title bars of all windows that open in the screen. A font is a specification of type size and type style. The system default font is called "Topaz". Topaz is a fixed-width font and comes in two sizes.

o 8 display lines tall with 80 characters per line in a 640-pixel high-resolution display (40 characters in low-resolution)

o 9 display lines tall with 64 characters per line in a high resolution display (32 characters in low-resolution)

On a television screen, you may not be able to see all 640 pixels across a horizontal line. On any reasonable television, however, a width of 600 pixels is a safe minimum, so you should be able to fit 60 columns of the large Topaz font. Note that font is a Preferences item and the user can choose either the 80- or 60-column (8- or 9-line) default, whichever looks best on his or her own monitor. You can use or ignore the user's choice of default font size.
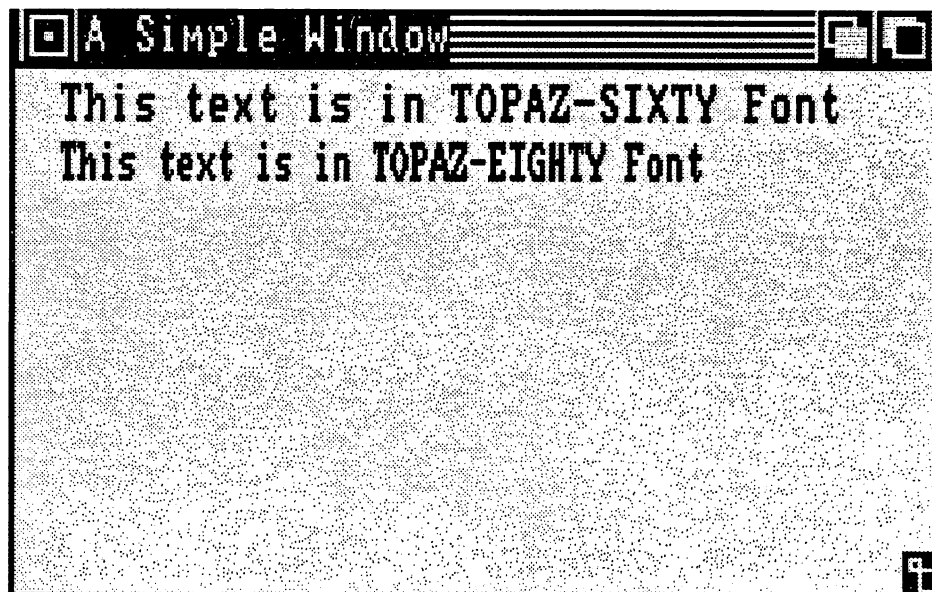
Figure 3-4: Topaz Font in 60 Column and 80 Column Types

If you want the default Topaz font in the default size currently selected by the user, set the *Font* field in the screen structure to NULL. If you want some other font, you specify it by creating a *TextAttr* structure and setting the screen's *Font* field to point to the structure. See the *Amiga ROM Kernel Manual* for more information about text support primitives.

HEIGHT, WIDTH, AND STARTING LOCATION

When you open a custom screen, you specify the initial starting location for the top line of the screen in the *Top* and *Left* fields of the screen structure. After that, the user can drag the screen up or down. You must always set the *Left* field (the x coordinate) to 0. (This parameter is included only for upward-compatibility with future versions of Intuition.)

You specify the dimensions of the screen in the *Height* and *Width* fields. You can set the screen *Height* field to any value equal to or less than the maximum number of lines in the display. For non-interlace mode, you can have a maximum of 200 lines and for interlace mode, 400 lines. You set the width to either 320 for low-resolution mode or 640 for high-resolution mode.

In setting the *Top* and *Height* fields, you must take into consideration a minor limitation of this release of Intuition and the graphics library. The bottom line of the screen cannot be above the bottom line of the video display. Therefore, the top position plus the height should not be such that the bottom line of the screen will be higher than the bottom line of the video display. To illustrate, you can have a display like this:

BACK SCREEN

Valid Placement of Screens

FRONT SCREEN
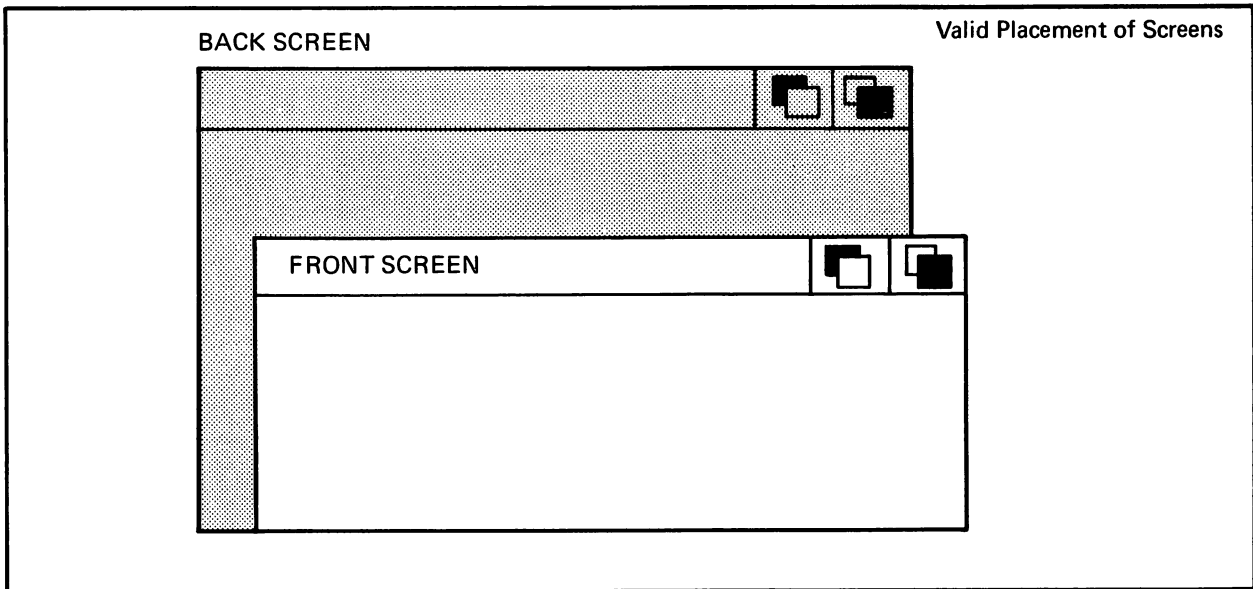
Figure 3-5: Acceptable Placement of Screens

but not like this:
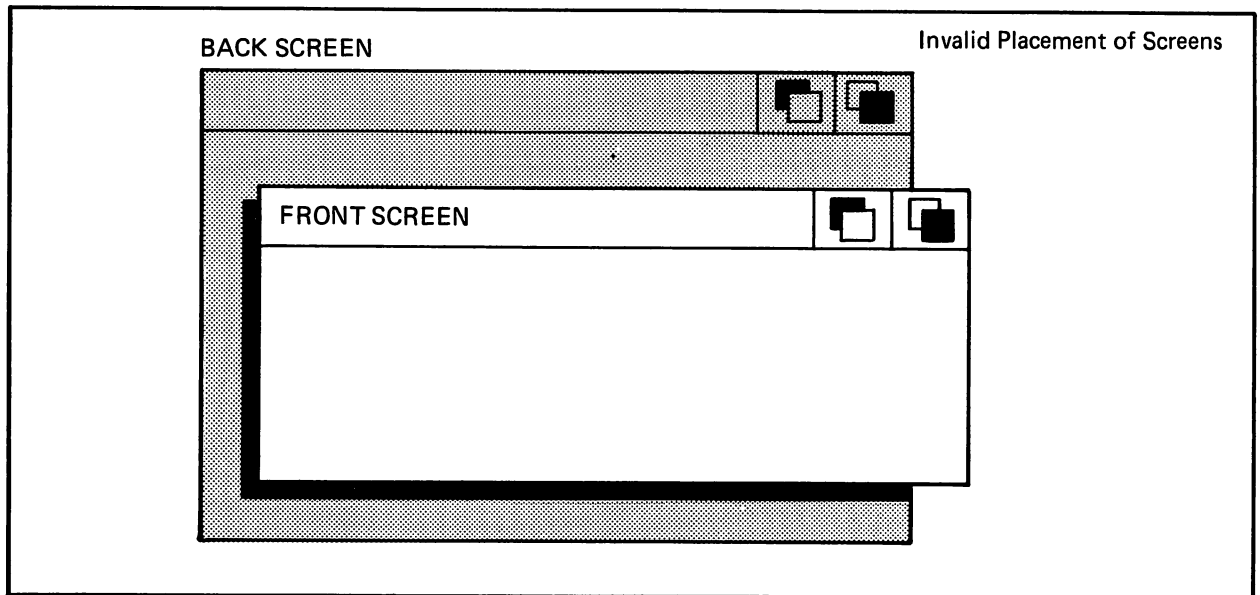
Figure 3-6: Unacceptable Placement of Screens

SCREEN TITLE

The screen title is used for two purposes:

o   to identify the screen like an identification tab on a file folder, and

o   to designate which window is the active one.

Although the initial screen title is set in the NewScreen structure, it can change according to the preferences of the windows that open in the screen. Each screen has two kinds of titles that can be displayed in the screen title bar:

o   A "default" title, which is specified in the NewScreen structure and is always displayed when the screen first opens.

o   A "current" title, which is associated with the currently active window. When the screen is first opened, the current title is the same as the default title. The current title depends upon the preferences of the currently active window.

Each window can have its own title, which appears in its own title bar, and its "screen title", which appears in the screen's title bar. When the window is the active window, it can display its screen title in the screen's title bar. The function *SetWindowTitles*() allows you to specify, change, or delete both the window's own title and its screen title.

Screen title display is also affected by calls to *ShowTitle()*, which coordinates the display of the screen title and windows that overlay the screen title bar. Depending upon how you call this function, the screen's title bar can be behind or in front of any special Backdrop windows that open at the top of the screen. By default, the title bar is displayed in front of a Backdrop window when the screen is first opened. Non-Backdrop windows always appear in front of the screen title bar.

You can change or eliminate the title of the active screen by calling *SetWindowTitles()*.

CUSTOM GADGETS

You can attach a linked list of custom gadgets to your custom screen. Each screen custom gadget must have the gadget flag SCRGADGET set in the gadget structure. See Chapter 5, "Gadgets", for information about creating custom gadgets and linking them together.

Using Custom Screens

To create a custom screen, you follow these steps:

1. Initialize a NewScreen structure with the data describing the screen you desire.

2. Call *OpenScreen()* with a pointer to the NewScreen structure. The call to *Open-Screen()* returns a pointer to your new screen (or it returns zero if your screen couldn't be opened).

   After you call *OpenScreen()*, the NewScreen structure is no longer needed. If you've allocated memory for it, you can free this memory.

Before you create a NewScreen structure, you need to decide on the following:

o The height of the screen in lines and where on the display the screen should begin; that is, its y-position.

o How many colors you want; the color you want for the background; the color for rendering text, graphics, and details such as borders; and the color for filling block areas such as the title bar.

o Horizontal resolution (320 or 640 pixels in a horizontal line) and vertical resolution (200 non-interlace or 400 interlace lines high).

o The text font to use for this screen and all windows that open in this screen.

o Text to be displayed in the screen's title bar

o Custom gadgets to be added to the screen

o Whether you want your own display memory for this screen or you want Intuition to allocate the display memory for you

NEWSCREEN STRUCTURE

Here are the specifications for the NewScreen structure:

```
struct NewScreen
{
    SHORT LeftEdge, TopEdge, Width, Height, Depth;
    UBYTE DetailPen, BlockPen;
    USHORT ViewModes;
    USHORT Type;
    struct TextAttr *Font;
    UBYTE *DefaultTitle;
    struct Gadget *Gadgets;
    struct BitMap *CustomBitMap;
};
```

The meanings of the variables and flags in NewScreen structure are as follows.

LeftEdge   Initial x-position for the screen.

This field is not currently used by Intuition; however, for upward compatibility, always set this field to 0.

TopEdge   Initial y-position of the screen.

Set this field to an integer or constant representing one of the lines on the screen.

Width   Width of the screen.

Set this field to 320 for low-resolution mode or 640 for high-resolution mode.

Height   Height of the screen in number of lines.

Set this field to up to 200 for non-interlace mode and 400 for interlace mode.

Depth   Number of bit-planes in the screen.

Set this field from 1 to 6.

DetailPen, BlockPen

DetailPen—color register number for details such as gadgets and text in the title bar.

BlockPen—color register number for block fills, such as the title bar area.

ViewModes

These flags select display modes. You can set any or all of them:

HIRES

Selects high resolution mode (640 pixels across). The default is 320 pixels across.

INTERLACE
Selects interlace mode (400 lines). The default is 200 lines.

SPRITES
Set this flag if you are want to use sprites in the display.

DUALPF
Set this flag if you want two playfields.

HAM
Set this flag if you want hold-and-modify.

Type   Set this to CUSTOMSCREEN. You may also set the CUSTOMBITMAP flag if you have your own BitMap and display memory which you want used for this screen (see *CustomBitMap* below).

Font   A pointer to the default *TextAttr* structure for this screen and all Intuition-managed text that appears in the screen and its windows. Set this to NULL if you want to use the default Intuition font.

DefaultTitle
A pointer to a null-terminated line of text that will be displayed in the screen's title bar, or just NULL if you want a blank title bar.

Null-terminated means that the last character in the text string is NULL.

Gadgets   A pointer to the first gadget in a linked list of your custom gadgets for this screen.

CustomBitMap
A pointer to a BitMap structure if you you have your own display memory which you want used as the display memory for this screen. You inform Intuition that you want to supply your own display memory by setting the flag CUSTOMBITMAP in the *Types* variables above, creating a BitMap structure that points to your display memory, and having this variable point to your BitMap.

SCREEN STRUCTURE

If you've successfully opened a screen by calling the *OpenScreen()* function, you receive a pointer to a Screen structure. The following list shows the variables of the Screen structure which may be of interest to you. This isn't a complete list of the Screen variables, only the more useful ones. Also, most of these variables are for use by advanced programmers, so you may choose to ignore them for now.

TopEdge   Examine this to see where the user has positioned your screen.

MouseX, MouseY
> You can look here to see where the mouse is with respect to the upper-left corner of your screen.

ViewPort, RastPort, BitMap, LayerInfo
> For the hard-core graphics users, these are actual instances of these graphics structures (NOTE: *not* pointers to structures). For simple use of custom screens, these structures can be ignored completely.

BarLayer This is the pointer to the Layer structure for the screen's title bar.

SCREEN FUNCTIONS

The following functions affect screen display.

Opening a Screen

This is the basic function to open an Intuition custom screen according to the parameters specified in NewScreen. This function sets up the screen structure and substructures, does all the memory allocations, and links the screen's ViewPort into Intuition.

> *OpenScreen (NewScreen)*

Showing Screen Title Bar

This function causes the screen's title bar to be displayed or not, according to your specification of the *ShowIt* variable and the position of the various types of windows that may be opened in the screen.

> *ShowTitle (Screen, ShowIt)*

>> The screen's title bar can be behind or in front of any Backdrop windows that are opened at the top of the screen. The title bar is always concealed by other windows, no matter how this function sets the title bar.

>> The variable *Screen* is a pointer to a screen structure.

>> Set the variable *ShowIt* to Boolean TRUE or FALSE according to whether the title is to be hidden behind Backdrop windows:

>> - When *ShowIt* is TRUE, the screen title bar is shown in front of Backdrop windows.

- When *ShowIt* is FALSE, the screen title bar is always behind any window.

Moving a Screen

With this function, you can move the screen vertically.

> *MoveScreen (Screen, DeltaX, DeltaY)*
>
>> Moves the screen in a vertical direction by the number of lines specified in the *DeltaY* argument. (*DeltaX* is here for upward-compatibility only, and is currently ignored).
>>
>> *Screen* is a pointer to the screen structure.

Changing Screen Depth Arrangement

These functions change the screen's depth arrangement with respect to other displayed screens.

> *ScreentoBack (Screen)*
>
>> Sends the specified screen to the back of the display.

> *ScreentoFront (Screen)*
>
>> Brings the specified screen to the front of the display.

Closing a Screen

The following function unlinks the screen and ViewPort and deallocates everything. It ignores any windows attached to the screen. All windows should be closed first. Attempting to close a window after the screen is closed will crash the system. If this is the last screen displayed, Intuition attempts to reopen the Workbench.

> *CloseScreen (Screen)*
>
>> The variable *Screen* is a pointer to the screen to be closed.

Handling the Workbench

These functions are for opening, closing, and modifying the Workbench screen.

OpenWorkBench()

>This routine attempts to open the Workbench screen. If there's not enough memory to open the screen, this routine fails. Also, if the Workbench tool is active, it will attempt to reopen its windows.

CloseWorkBench()

>This routine attempts to close the Workbench screen. If another application (other than the Workbench tool) has windows opened in the Workbench screen, this routine fails. If only the Workbench tool has opened windows in the Workbench screen, then the Workbench tool will graciously close its windows and allow the screen to close.

WBenchToFront(), WBenchToBack()

>If the Workbench screen is opened, calling these routines will cause it to be depth-arranged accordingly. If the Workbench screen isn't opened, these routines have no effect.

Advanced Screen and Display Functions

These functions are for advanced users of Intuition and graphics. They are used primarily in programs that want to make changes in their custom screens (for instance, in the Copper instruction list). These functions cause Intuition to incorporate a changed screen and merge it with all the other screens in a synchronized fashion. For more information about these functions, see Chapter 11, "Other Features".

MakeScreen (Screen)

>This function is the Intuition equivalent of the lower-level *MakeVPort()* graphics library function. This performs the *MakeVPort()* call for you, synchronized with Intuition's own use of the screen's ViewPort. The variable *Screen* is a pointer to the screen which has the ViewPort that you want remade.

RethinkDisplay()

>This procedure performs the Intuition global display reconstruction. This includes massaging some of Intuition's internal state data, rethinking about all of the Intuition screen ViewPorts and their relationship to one another, and, finally, reconstructing the entire display by merging the new screens into the Intuition View structure. This function calls the graphics primitives *MrgCop()* and *LoadView()*.

This routine remakes the entire Intuition display. It performs a *MakeVPort()* (graphics primitive) on every Intuition screen, and then calls *RethinkDisplay()* to recreate the view.

Chapter 4

WINDOWS

In the last chapter, you learned about Intuition screens, the basic unit of display. This chapter covers the windows supported by those screens. The first half of the chapter provides a general description of windows in general—including the different ways of handling the I/O of the virtual terminal; preserving the display when windows get overlapped; how you open windows and define their characteristics; and how you get the pre-defined gadgets for shaping, moving, closing, and depth-arranging windows. This section also defines the different kinds of special windows that extend even further the capabilities of the Intuition windowing system. You will also see how you can customize your windows by adding individual touches like your own custom pointer.

In the second half of the chapter, you get all the details you need for designing your own windows—an overview of the process of creating and opening a window, the specification for the window structure, and brief descriptions of the functions you can use for windows.

About Windows

The windows you open can be colorful, lively, and interesting places for the user to work. You can use all of the standard Amiga graphics, text, and animation primitives (functions) in every one of your windows. You can also use the quick and easy Intuition structures and functions for rendering images, text, and lines into your windows. The special Intuition features that go along with windows, like the gadgets and menus, can be visually exciting as well.

Each window can open an Intuition Direct Communication Message Port (IDCMP), which offers a direct communication channel with the underlying Intuition software, or the window can open a Console Device for input and output. Either of these communication methods turns the window into a visual representation of a virtual terminal, where your program can carry on its interaction with the user as if it had the entire machine and display to itself. Your program can open more than one window and treat each separately as a virtual terminal.

Both you and the user deal with each individual window as if it were a complete terminal. The user has the added benefit of being able to arrange the terminals front to back, shrink them or expand them, or overlap them.
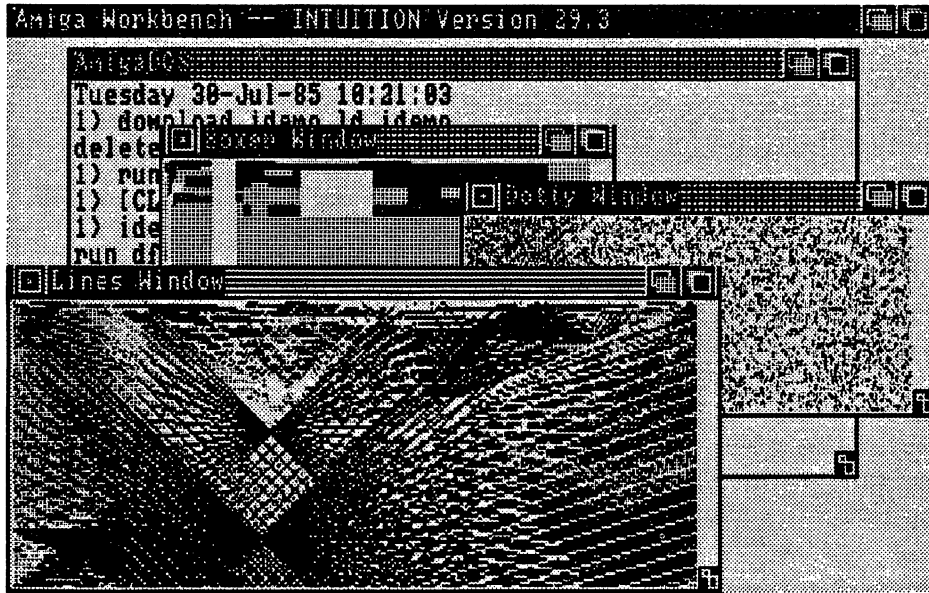
Figure 4-1: A High-Resolution Screen and Windows

Windows are rectangular display areas, but this doesn't begin to describe the forms they can take. The user can shape windows by making them wider or longer or both to reveal more of the information being output by the program. The user can shrink windows into long, narrow strips or small boxes to reveal other windows or make room for other windows to open. Multiple windows can be overlapped and the user can bring a window up front or send it to the bottom of the stack with a click of the mouse button. While the user is doing all this shaping and rearranging and stacking of windows, your program doesn't have to pay any attention. To the program, there is nothing out there but a user with a keyboard and a mouse (or, in place of a mouse, there could be a joystick or graphics tablet or practically any other input device).

Your program can open as many of these virtual terminal windows as the memory configuration of your Amiga will allow. Each window opens in a specific screen and several windows may open in the same screen. Even windows opened by different programs may coexist in the same screen.

Your program can open windows for any purpose. For example, different windows of an application can represent:

o   different interpretations of an object, such as the same data represented as a bar chart and a pie chart

o   related parts of a whole, such as the listing and output of a program

o   different parts of a document or separate documents being edited simultaneously

You open a window by specifying its structure and issuing a call to a function that opens windows. After that, you can output to the user and receive input while Intuition manages all the user's requests to move, shape, and depth-arrange the window. Intuition lets you know if the user makes a menu choice, chooses one of your own custom gadgets, or wants to close the window. If you need to know when the user changes the window's size or moves the pointer,

Intuition will tell you about that, too.

Custom gadgets, menus, input/output, and controllers are dealt with in their own chapters. The balance of this section deals with some important window concepts. You'll need to know about these before proceeding with the specific instructions for opening your own windows in the "Using Windows" section.

WINDOW INPUT/OUTPUT

You can choose from two different paths for input and two different paths for output. Each path satisfies particular needs. The two paths for user input are:

o   Intuition Direct Communications Message Ports

The Message Ports give you mouse (or other controller) events, keyboard events, and Intuition messages in their most raw form, and also supply the way for your program to communicate *to* Intuition.

o   Console Device

This gives you processed input data, including keycodes translated to ASCII characters and Intuition event messages converted to ANSI escape sequences. If you wish, you can also get raw (untranslated) input through the Console Device.

There are also two paths for program output:

o   Text is output through the Console Device, which formats and supplies special text primitives and text functions such as auto-line wrapping and scrolling.

o   Graphics are output through the general-purpose Amiga graphics primitives, which provide rendering functions like area fill and line drawing and animation functions.

If you use the Console Device for input or output or both, you need to open it after opening your window. If you want the IDCMP for input, you specify one or more of the IDCMP flags in the NewWindow structure. This automatically sets up a pair of Message Ports, one for Intuition and one for you. Although the IDCMP doesn't offer text formatting or keycode translation, it has many special features that you may want, and requires less RAM and processing overhead.

To select your I/O methods, you should read Chapter 8, "Input and Output Methods".

OPENING, ACTIVATING, AND CLOSING WINDOWS

Before your program can open a window, you need to initialize a NewWindow structure. This structure contains all the arguments needed to define and open a window, including initial position and size, sizing limits, color choices for window detailing, gadgets to attach, how to preserve the display, IDCMP flags, window type if it's one of the special windows, and the screen to open in.

A window is opened and displayed by a call to the *OpenWindow()* function whose only argument is a pointer to the NewWindow structure. After successfully opening a window, you receive a pointer to another structure, the Window structure. If you are opening the window in a custom screen, you need to call *OpenScreen()* before opening the window.

Only one window is active in the system at a time. The active window is the one that's receiving user input through a keyboard and mouse (or some other controller). The active window looks different from inactive windows. In relation to inactive windows, some areas of the active window are displayed more boldly. In particular, the title bars of inactive windows are covered with a faint pattern of dots, rendering them slightly less distinct. This is called "ghosting". See Figure 4-1 for an example of the appearance of inactive windows. When the user brings up a menu list in the screen title bar, the active window's menu list is displayed. Also, the active window has an input cursor when an input request is pending, and the active window receives system messages.

Your program doesn't have to worry about whether one of its windows is active or not. The inactive windows can just wait for the user to get back to them, or they can be doing some background task that doesn't require user input. The job of activating windows is mostly left up to the user, who activates a window by moving the pointer into the window and clicking the right mouse button. There is, however, an ACTIVATE flag in the NewWindow structure. Setting this flag causes the window to become active when it opens. If the user is doing something else when a window opens with the ACTIVATE flag set, input is immediately redirected to the newly opened window. You will probably want to set this flag in the first window opened when your program starts up. Windows opened after the first one don't need to have the ACTIVATE flag set, though they may. It's up to you to design the flow of information and control.

After your window is opened, you can discover when it's activated and inactivated by setting the IDCMP flags ACTIVEWINDOW and INACTIVEWINDOW. If you set these flags, you will receive a message every time the user activates your window or causes your window to become inactive by activating some other window.

Although there is a window closing gadget, a window does not automatically close when the user selects this gadget. Intuition sends you a message about the user's action. You can then do whatever clean-up is necessary, such as replying to any outstanding Intuition messages or verifying that the user really meant to close the window, and then call *CloseWindow()*.

If the user closes the last window in a standard screen other than the Workbench screen, the screen also closes.

When the active window is closed, the previously active window may become the active window. The window (call it Window A) that was active when this one was *opened* will become the active window. If Window A is already closed, then the window (if any) that was active when Window A opened will become the active window, and so on.

SPECIAL WINDOW TYPES

Intuition's special windows give you some very useful bonus features, in addition to all the normal window features. The Backdrop window stays anchored to the back of the display and gives you a way to take over the display without taking over the machine. The Borderless window gives you a window with no drawn border lines. The window with the fanciful (some even say whimsical) name, Gimmezerozero, gives you all the border features *plus* the freedom to ignore borders altogether when you're rendering into the window. Finally, there's SuperBitMap,

a window that not only gives you your own display memory in which to render, but frees you from ever worrying about preserving the window when the user sizes it or overlaps it with another window.

Notice that these are not necessarily separate, discrete window types. You can combine them for even more special effects. For instance, you can create a Backdrop, Borderless window that fills the entire screen and looks like a normal computer display terminal.

Borderless Window Type

This window is distinguished from other windows by having no default borders. With normal windows, Intuition creates a thin border around the perimeter of the window, allowing the window to be easily distinguished from other windows and the background. When you ask for a Borderless window, you do not get this default thin border; however, your window can still have borders. It can have borders based solely on the location of border gadgets, and whether or not you've supplied title text, or it may have no gadgets or text and thus no visible borders, and no border padding, at all. You can use this window to cover the entire video display. It's especially effective combined with a Backdrop window. This combination forms a window that you can render in almost as freely as writing directly to the display memory of a custom screen. It has the added benefit that you can render in it without running the risk of trashing menus or other windows in the display.

If you use a Borderless window that doesn't cover the entire display, then be aware that its lack of borders may tend to cause visual confusion on the screen. Since windows and screens share the same color palette, borders are often the only way of distinguishing a window from the background.

Set the BORDERLESS flag in the NewWindow structure to get this window type.

Gimmezerozero Window Type

The unique feature of a Gimmezerozero window is that there are actually two "planes" to the window: a larger, outer plane where the window title, gadgets, and border are rendered; and a smaller, inner plane (also called the *inner window*) into which you can draw freely without worrying about the window border and its contents. The top-left coordinates of the inner window are always (0,0), regardless of the size or contents of the outer window; thus the name "Gimmezerozero".

The area into which you can draw is formally defined as the area within the variables Border-Left, BorderTop, BorderRight, and BorderBottom. These variables are computed by Intuition when the window is opened. To render into normal windows with the graphics primitives (for instance to draw a line from the top left to somewhere else in the window), you have to start the line away from the window title bar and borders. Otherwise, you risk drawing the line over the title bar and any gadgets that may be in the borders. In a Gimmezerozero window, you can just draw a line from (0,0) to some other point in the window without worrying at all about things in the window borders.

The Gimmezerozero window has the disadvantages of using more RAM and degrading performance in moving and sizing of windows. There can be a noticeable performance lag, especially when several Gimmezerozero windows are open at the same time.

There are some special variables in the Window structure which pertain only to Gimmezerozero windows. The GZZMouseX and GZZMouseY variables can be examined to discover the position of the mouse relative to the inner-window or the window. The GZZWidth and GZZHeight variables can be used to discover the width and height of the inner-window.

The Console Device gives you another kind of encumbrance-free window. If you are using the Console Device, any formatted text you are outputting goes into an inner window automatically and you don't need to worry about gadgets. Therefore, you don't need a Gimmezerozero window just for the purpose of text output. See the Chapter 8, "Input and Output", for more information about this aspect of the Console Device.

Requesters in a Gimmezerozero window appear relative to the inner window. If you are bringing up requesters in the window, you may wish to take this into consideration when deciding where to put the requesters. See Chapter 7, "Requesters and Alerts" for more information about requester location.

To specify a Gimmezerozero window, you set the GIMMEZEROZERO flag in the Window structure's flags. All system gadgets you attach to this type of window will automatically go into the gadget bit-map; however, if you are attaching custom gadgets and you want the gadgets to appear in the border (*not* in the inner-window), be sure to set the GZZGADGET flag in your gadget structures. If you don't, Intuition will render custom gadgets in the display of the inner-window.

Backdrop Window Type

The Backdrop window, as its name implies, always opens in the back of the Intuition screen. Its great advantage is that other windows can overlap it and be depth-arranged without ever going behind the Backdrop window. Because of this characteristic, you can use the Backdrop window as a primary display surface while opening other auxiliary windows on top of it.

The Backdrop window is like normal windows except:

o   It always opens behind all other windows (including other Backdrop windows that you might have already opened).

o   The only system gadget you can attach is the close-window gadget. (You can attach your own gadgets, as usual.)

o   Normal windows in the same screen open in front of all Backdrop windows and always stay in front of them. No amount of depth arranging will ever send a non-Backdrop window behind a Backdrop window.

You might want to use a Backdrop window, for example, in a simulation program where the environment is rendered in the Backdrop window while the simulation controls exist in normal windows that float above the environment. Another example is a sophisticated graphics program where the primary work surface is on the Backdrop window while auxiliary tools are made available in normal windows in front of the work surface.

You can often use a Backdrop window instead of rendering directly in the display memory of a custom screen. If you want to render into your background with the graphics primitives, you may even prefer a Backdrop window to a custom screen because you do not run the danger of writing to the window at the wrong time and trashing a menu that is being displayed. In fact, if you also set the BORDERLESS flag and you create a window that's the full-screen width and

height, you get a window that fills the entire screen and stays in the background. If you also specify no gadgets, there will be no borders at all. Finally, if you add a call to *ShowTitle()* with an argument of FALSE, the window will conceal the screen title. All of these steps result in a window that fills the entire video display, has no borders, and stays in the background!

To get the Backdrop feature, you set the BACKDROP flag in the Window structure.

SuperBitMap Window

SuperBitMap is both a window type and a way of preserving and redrawing the display. This window is like other windows except you get your own bit-map instead of using the screen's bit-map. The windowing system displays some portion of the window's bit-map in the screen's raster according to the dimensions and limits you specify and the user's actions. You can make the bit-map any size as long as the window sizing limits are set accordingly.

This window is very handy where you want to give the user the flexibility of scrolling around and revealing any portion of the bit-map. You can do this because the entire bit-map is always available to be displayed.

To get this type of window, you set the SUPER-BITMAP flag in the window structure and set up a BitMap structure. You probably want to set the GIMMEZEROZERO flag also, so that the borders and gadgets will be rendered in a separate bit-map. You need to be certain that the size-limiting variables in the window structure are properly set, considering the size of the bit-map and how much of it you want to display.

For complete information about SuperBitMap, see "Setting Up a SuperBitMap Window" later in this chapter.

WINDOW GADGETS

The easiest way for a user to communicate with a program running under Intuition is through the use of window gadgets. There are two basic kinds of window gadgets—system gadgets that are pre-defined and managed by Intuition, and your own custom application gadgets.

System Gadgets

System gadgets are supplied for the user to manage these aspects of window display: size and shape of windows, location of windows on the screen, and depth arrangement. Also, there is a system gadget for the user to tell the application when he or she is ready to close the window. These gadgets save you a lot of work because, with the exception of the close gadget, your program never has to pay any attention to what the user does with them. On the other hand, if you want to be notified when the user sizes the window because of some special rendering you may be doing in the window, Intuition will let you know. For more information, read about IDCMP functions in the Chapter 8, "Input and Output Methods".

In the Window structure, you define the starting location and starting size of a window and a maximum and minimum height and width for sizing the window. When the window opens, it appears in the location and in the size you have specified. After that, however, the user

normally has the option of shaping the window within the limits you have set, moving the window about on the screen, and depth arranging it—either sending it into the background behind all the other displayed windows or bringing it into the foreground. To give the user this freedom, plus the ability to request that the window be closed, you can attach system gadgets to the window. The graphic representations of these gadgets are pre-defined, and Intuition always displays them in the same well-publicized locations in the window borders. In the window structure, you can set flags to request that all, some, or none of these system gadgets be attached to your window. The system gadgets and their locations in the window are:

o   a *sizing* gadget in the lower right of the window. With the sizing gadget, the user can stretch or shrink the height and width of the window. You set the maximum and minimum limits for sizing. You can specify whether this gadget is located in the right border or bottom border, or in both borders.

o   two *depth arrangement* gadgets in the upper right of the window. One sends the window behind all other displayed windows (back gadget) and the other brings the window to the front of the display (front gadget).

o   a *drag* gadget, which occupies every part of the window title bar not taken up by other gadgets. The drag gadget allows the user to move the window to a new location on the screen. A title in the title bar does not interfere with drag gadget operation.

o   a *close* gadget in the upper left of the window, which allows the user to request that the window be closed.

The picture below shows how all the system window gadgets look and where they are located in the window borders.
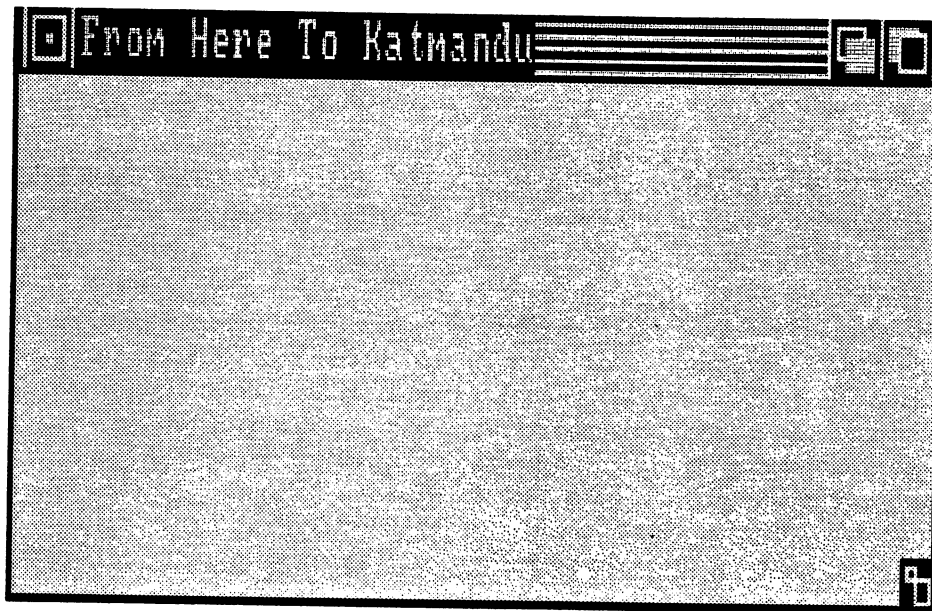
Figure 4-2: System Gadgets for Windows

Application Gadgets

Application gadgets come in four flavors — proportional, Boolean, string, and integer. You can use application gadgets to request various kinds of input from the user, and that input can affect the application in any way you like. You design them as text or graphic images to go anywhere in the window. For application gadgets, you define a data structure for each one and create a linked list of these structures. To attach your list of gadgets to a window, you set a pointer in the NewWindow structure to point to the first gadget in the list. For details about creating gadgets, see the chapter called "Gadgets".

WINDOW BORDERS

Intuition offers you several possibilities for handling window borders. You can take advantage of the fancy border features, such as automatic double border lines around the window and automatic padding of borders to allow for gadgets. If you'd rather, you can eliminate borders completely, or you can use the Gimmezerozero window that gives you all the border features and then lets you ignore them.

The actual border *lines* are drawn around the perimeter of the window, and are mostly distinct from the border *area* where border gadgets are placed. Intuition automatically draws a double border around a window unless you ask for something different. This nominal border consists of an outer line around the entire window, rendered in the *BlockPen* color and within this a second line rendered in the *DetailPen* color. The two "pen" colors are defined in the NewWindow structure.

The default minimum thickness of the border areas depends upon certain parameters set in the definition of the underlying screen, certain choices the user has made with Preferences, and the default font. If the window is not a special Borderless window, then the borders will be at least the default thickness. Intuition adjusts the size of a window's border areas to accommodate system gadgets or your own application gadgets.

You can find the thickness of the border areas in the variables *BorderLeft, BorderTop, Border-Right,* and *BorderBottom.* These variables are computed when the window is opened and can be found in the Window structure. You may want to use them if you are drawing lines into the window with graphics primitives, where you need to specify a set of coordinates as the beginning and ending points for the line. In a typical window, you can't specify a line from (0,0) to (50,50) because you may draw a line over the window title bar. Instead, you would use the border variables to specify a line from (0+BorderLeft, 0+BorderTop) to (50+BorderLeft), 50+BorderTop). This may look clumsy, but it offers a way of avoiding a Gimmezerozero window, which is much more convenient to use but requires extra memory and impacts performance.

For the top border, in addition to the system gadgets and your own gadgets, you can specify a window title. The window title bar does not appear at all unless you specify one of the following:

    o   a window title, or

    o   any of the system gadgets for window dragging, window depth arranging, or window closing.

Usually, borders are drawn automatically and adjusted *within* the dimensions you specify in the NewWindow structure. In the special Borderless and Gimmezerozero windows, borders are handled differently. A Borderless window has no drawn borders and no automatic border spacing or padding. If you have system gadgets or your own gadgets with a border flag set, borders may be visually defined by the gadgets. A Gimmezerozero window places the borders and gadgets in their own bit-map separate from the window's bit-map. This means you can render freely into the entire surface of the window without worry about scribbling over the gadgets.

You can specify whether or not your application gadgets reside in the borders, and in which border, by setting a flag in the gadget structure. See Chapter 5, "Gadgets", for more information about gadgets and how to place them where you want them.

PRESERVING THE WINDOW DISPLAY

When a window is revealed after having been overlapped, the display has to be redrawn. Intuition offers three ways of preserving the display:

   o   In the *Simple Refresh* method, your program redraws the display.
   o   In the *Smart Refresh* method, Intuition keeps a copy of the display in RAM buffers.
   o   In the *SuperBitMap* method, you allocate an entirely separate display memory for your window.

Smart Refresh and SuperBitMap use the window's idea of its display memory space to save the parts of the window that are not currently being displayed. Windows and other high-level display components, like menus and gadgets, have a "virtual" understanding of their display memory. The application can ignore other windows being displayed and write into its own virtual memory area. The Amiga graphics software then takes these requests to render to virtual display memory and translates them into real operations that render into save buffers (for Smart Refresh) or into areas of a private bit-map (for SuperBitMap) maintained by the application.

The following figures compare the three different methods of refreshing the window display.
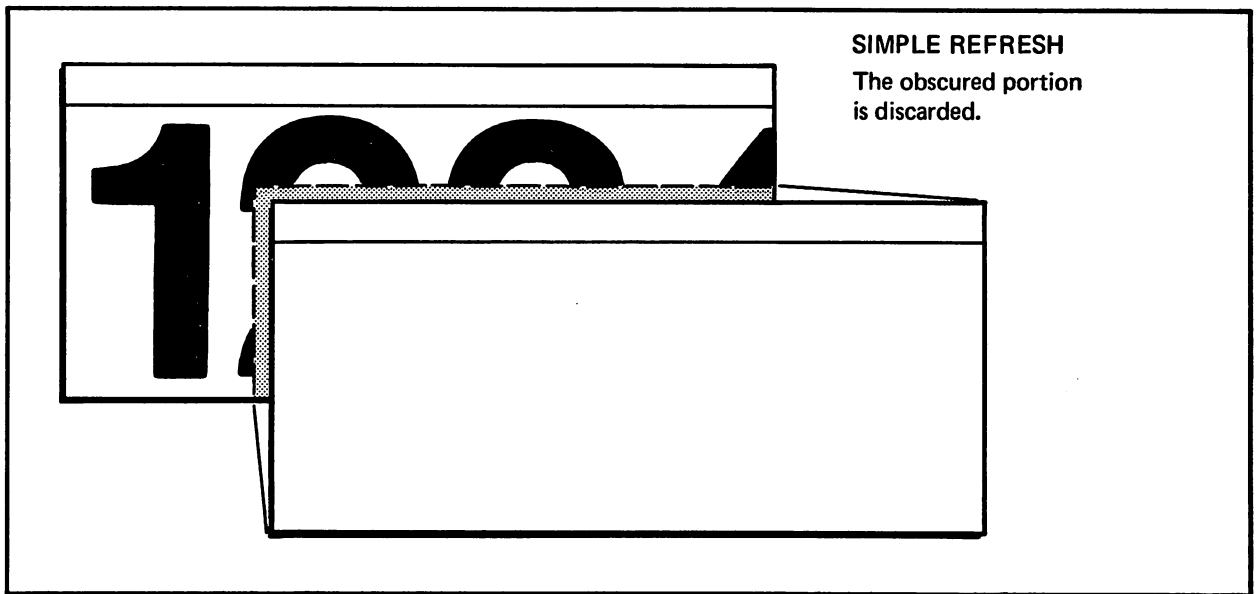
Figure 4-3: Simple Refresh

SMART REFRESH

The obscured portion is preserved offscreen.

Figure 4-4: Smart Refresh

SUPER BIT MAP

Portions of your offscreen bitmap are shown onscreen.
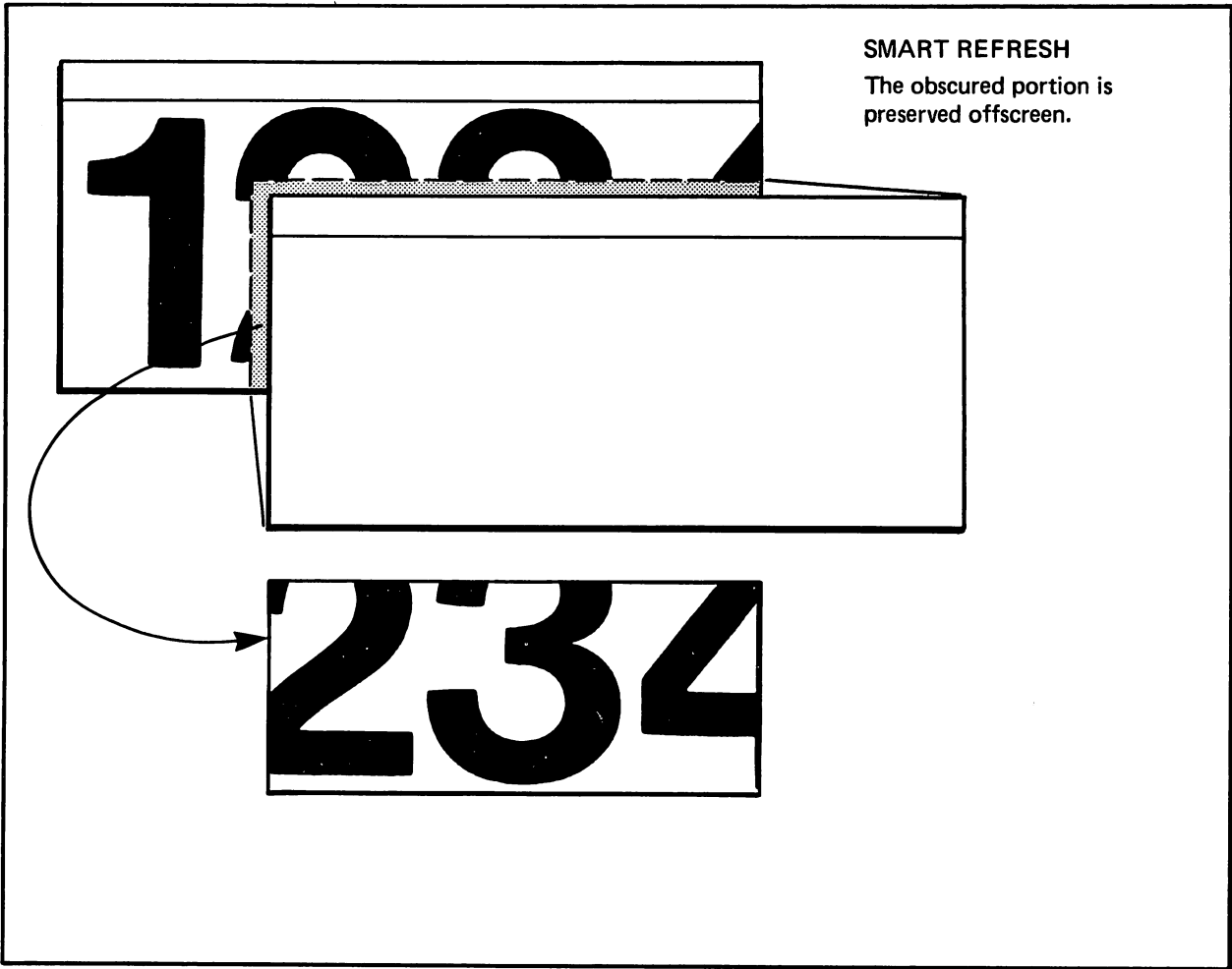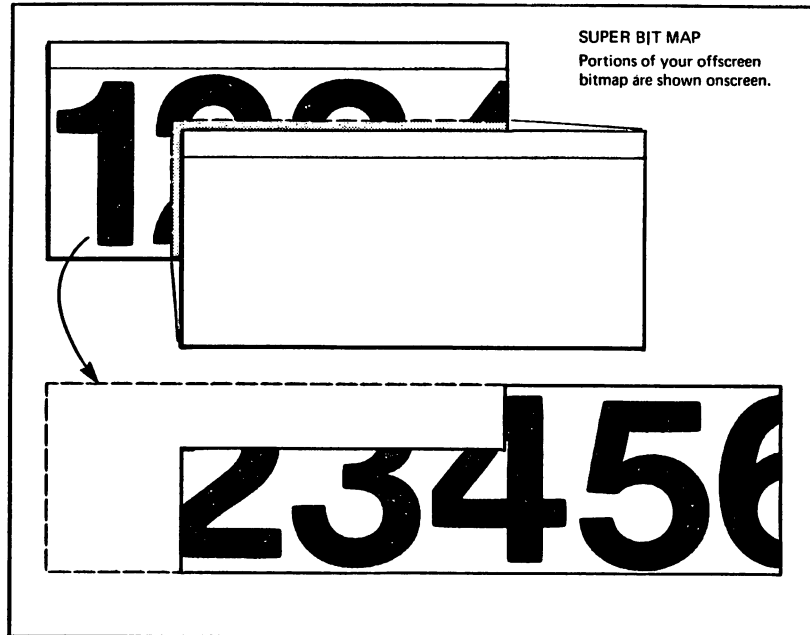
Figure 4-5: SuperBitMap Refresh

The three methods of preservation are explained below. You must choose one of them.

Simple Refresh

With this redrawing method, Intuition doesn't need to remember anything about windows that are overlapped. For the most part, the program is responsible for redrawing the window. If the user sizes the window larger on either axis or reveals a window that was overlapped, the program must redraw the display. However, if the user merely drags the window around, Intuition preserves it and redisplays it in the new location. Simple Refresh tends to be slower than other methods, but it is memory efficient since no RAM is consumed in saving the obscured portions of a window. Simple refresh uses the screen's display memory for the window's display.

Smart Refresh

With this redrawing method, Intuition keeps all information about the window in RAM, whether the window is currently concealed or is up front. If the user reveals a window that was overlapped, Intuition recreates the display for you. If the sizing gadget is attached, the application can still recreate a portion of the display when the user makes the window larger. Smart refresh uses the screen's display memory for the window display and requires extra buffers for the off-screen portions of the window (portions not currently being displayed). Smart Refresh uses more display memory but it redraws the display faster than Simple Refresh.

SuperBitMap

This is both a special type of window and a method of redrawing the display. When you choose this method of redrawing, you get your own bit-map to use as display memory instead of using the screen's display memory. You make this bit-map as large as the window can get (or larger). You never have to worry about redisplay after the window is uncovered because the entire display is always there in RAM. For more information about SuperBitMap, see the "Special Windows" section in this chapter.

REFRESHING THE WINDOW DISPLAY

If you open either a Simple Refresh or Smart Refresh window, you may be asked to refresh part of your display at some time or other. When a Simple Refresh window is moved or sized, or when other windows are moved or sized in such a way that areas of a Simple Refresh window are revealed, the window will have to be refreshed. With Smart Refresh windows, the window has to be sized larger on either axis to generate a REFRESHWINDOW event.

You find out that your window needs refreshing via either source of input, the IDCMP or the Console Device. You get a message telling you that your window needs to be refreshed. The message is of class REFRESHWINDOW. Every time you learn that you should refresh your window, you must take some action, even if it's just the acceptable minimum action described below.

When you are asked to refresh your window, before you actually start to refresh it you should call the Intuition function *BeginRefresh()*. This function makes sure that refreshing is done in the most efficient way, only redrawing those portions of your window that really need to be redrawn. The rest of the rendering commands are discarded.

After you call *BeginRefresh()*, you redraw the display. Then, after the display is redrawn, you call *EndRefresh()* to restore the state of the internal structures.

Even if you don't want to redraw right now, you should at least call *Begin/EndRefresh()* each time you are asked to refresh your window. This helps Intuition and the layer library to keep things sorted and organized.

If you are opening a window which you will never care to refresh, no matter what happens to or around it, then you can avoid the requirement of calling *Begin/EndRefresh()* by setting the NOCAREREFRESH flag in the NewWindow structure when you open your window.

WINDOW POINTER

The active window contains a pointer for the user to make selections from menus, select gadgets, and so on. The user moves the pointer around with a mouse controller, but other kinds of controllers or the keyboard cursor keys can be substituted.

Pointer Position

If your program needs to know about pointer movements, you can either look at the position variables or arrange to receive broadcasts each time the pointer moves. The position variables *MouseX* and *MouseY* always contain the current pointer x and y coordinates, whether or not your window is the active one. If you elect to receive broadcasts, you get a set of x,y coordinates each time the pointer moves These coordinates are relative to the upper left corner of your window and are reported in the resolution of your screen, even though the pointer's visible resolution is always in low resolution mode (Note that the pointer is actually a sprite).

If your window is a Gimmezerozero window, you can examine the variables GZZMouseX and GZZMouseY to find the position of the mouse relative to the upper-left corner of the inner-window.

To get broadcasts about pointer movements, either InputEvents or Message Port messages, you must set the REPORTMOUSE flag in your window structure. Thereafter, whenever your window is active, you'll get a broadcast every single time the pointer moves. This can be a lot of messages, so be prepared to handle them efficiently. If you want to change whether or not you are following mouse movements, you can call *ReportMouse()*.

You can also get broadcasts about pointer movements by setting the flag FOLLOWMOUSE in your application gadget structures. If this flag is set in a gadget, you get the current pointer position as long as that gadget is selected by the user. This can result in a lot of messages, too.

Custom Pointer

You can set up your window with a custom pointer to replace the default arrow pointer. To define the pointer, set up a sprite data structure. Sprites are one of the general-purpose Amiga graphics structures. To place your custom pointer in the window, you call *SetPointer()*. To remove your custom pointer from the window, you call *ClearPointer()*. Both of these functions take effect immediately if yours is the active window.

Also, you can change the colors of the Intuition pointer. The Intuition pointer is always sprite 0. To change the colors of sprite 0, call the graphics library routine *SetRGB4()*. Refer to Chapter 12, "Style", for more information about this.

See the last section of this chapter for a complete example of a custom pointer.

GRAPHICS AND TEXT IN WINDOWS

There are two ways of rendering graphics, lines, and text into windows. You can use all of the Amiga graphics, animation, and text primitives in any window. Also, you can use the quick and easy Intuition structures and functions to display Intuition Image, Border, or IntuiText structures in windows. Note that the Border structure is a general purpose line-drawing mechanism. See chapter 9, "Images, Line Drawing, and Text", for more information about these topics.

WINDOW COLORS

The number of colors you can use for the window display and the actual colors that will appear in the color registers are defined by the screen where the window opens. In the window structure, you specify two color register numbers ("pens"), one for the border outline text, and gadgets and one for block fills (like the title bar). These pen colors are also a function of the screen. You can specify different colors for the pens than those used by the screen or you can use the screen's pen colors.

WINDOW DIMENSIONS

In the NewWindow structure, you define the dimensions and the starting location of your window on the screen. If you are letting the user change the size and shape of the window, you also need to specify the minimum size to which the window can shrink and the maximum size to which it can grow. If you don't ask that the window sizing gadget be attached to the window, then you don't need to initialize any of these maximum and minimum variables.

In setting all these size dimensions, you need to bear in mind the the horizontal and vertical resolutions of the screen where you are opening the window.

If you want to change the sizing limits after you've opened the window, you can call *Window-Limits()* with the new values.

Using Windows

To create a window, you follow these steps:

1. Initialize a NewWindow structure.
2. When you are ready to display the window, call *OpenWindow()* with a pointer to the NewWindow structure.
3. After calling *OpenWindow()*, the NewWindow structure is no longer needed.

When creating a NewWindow structure, you need to decide on:

o   The screen in which the window will appear
o   The window's characteristics
   *   Which system gadgets you want
   *   Preservation method for the window display
   *   Special window features—Gimmezerozero, Borderless, Backdrop, or SuperBitMap
   *   Type of input from the Intuition Direct Communications Message Ports (if any)
   *   Pointer movement broadcasts
   *   Other characteristics, such as starting position and size, color of the pens used to draw borders and fill blocks
   *   Custom images, such as a custom "check mark" for the menus or a custom pointer

NEWWINDOW STRUCTURE

Here are the specifications for the NewWindow structure:

```
struct NewWindow
{
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    UBYTE DetailPen, BlockPen;
    USHORT IDCMPFlags;
    ULONG Flags;
    struct Gadget *FirstGadget;
    struct Image *CheckMark;
    UBYTE *Title;
    struct Screen *Screen;
    struct BitMap *BitMap;
    SHORT MinWidth, MinHeight;
    SHORT MaxWidth, MaxHeight;
    USHORT Type;
};
```

Here are the meanings of the fields in the NewWindow structure. Some of the fields contain variables to which you need to assign a value, some contain flag bits to set or unset, and some are pointers to other structures.

LeftEdge, TopEdge, Width and *Height*

These describe where your window will first appear on the screen and how large it will be initially. These dimensions are relative to the top-left corner of the screen, which has the coordinates (0,0):

LeftEdge   The initial x-position, which represents the offset from the first pixel on the line, pixel 0.

TopEdge   The initial y-position, which represents how many lines down from the top (line 0) you want the window to begin.

Width   The initial width in pixels

Height   The initial height in lines

DetailPen and *BlockPen*

Contain the "pen" numbers used to render details of the window. The colors associated with the pens are a function of the screen. If you supply a value of -1 for either of these, you will get the screen's value for that pen by default.

DetailPen   The pen number (or -1) for the rendering of window details like gadgets or text in the title bar

BlockPen   The pen number (or -1) for window block fills (like the title bar) and the outer rim of the window border.

Flags variable
You can set any of the following flags.

To get system gadgets, you set the applicable flags. They are:

WINDOWSIZING
Allows the user to change the size of the window. Intuition places the window sizing gadget in the lower-right of your window. By default, the right border is adjusted to accommodate the sizing gadget, but you can change this with the following two flags, which work in conjunction with WINDOWSIZING. The sizing gadget can go in either the right or bottom border (or both) of the window.

o SIZEBRIGHT
Puts the sizing gadget in the right border. This is the default.

o SIZEBBOTTOM
Puts the sizing gadget in the bottom border.

You might wish to set SIZEBBOTTOM to put the sizing gadget in the bottom border if you want all possible horizontal bits, for instance, for 80-column text, and are willing to sacrifice vertical space.

WINDOWDEPTH
Allows the user to change the window's depth arrangement with respect to all other currently displayed windows. Intuition places the window depth-arrangement gadgets in the upper-right of the window.

Setting this flag selects both the UPFRONT gadget to bring the window into the foreground and the DOWNBACK gadget to send it behind other currently displayed windows.

WINDOWCLOSE
When the user selects this gadget, Intuition transmits a message to your application. It's up to the application to call *CloseWindow()* when ready. Setting this flag attaches the standard close gadget to the upper-left of the window.

WINDOWDRAG
This turns the entire title bar of the window into a drag gadget, allowing the user to move the window into a different position on the screen by placing the pointer anywhere in the window title bar and dragging.

NOTE: Even if you do not specify a text string in the *Text* variable shown below, a title bar appears if you use any one of the system gadgets WINDOWDRAG, WINDOWDEPTH, or WINDOWCLOSE. In that case, the title bar is blank.

GIMMEZEROZERO
Set this flag if you want a Gimmezerozero window.

The following three flags determine how Intuition preserves the display when an overlapped window is uncovered by the user. You *must* select one of the following:

SIMPLE_REFRESH

Every time a portion of the window is revealed, the application program must redraw its display.

SMART_REFRESH

The only time you have to redraw your display is when the user uses the window-sizing gadget to make the window larger.

NOTE: If you don't ask for the WINDOWSIZING gadget when you open a SMART_REFRESH window, then Intuition *never* tells you to redraw this window.

SUPER_BITMAP

Setting this flag means you are allocating and maintaining your own bit-map and display register.

You must also set the *BitMap* field to point to your own BitMap structure.

BACKDROP

Set this flag if you want a Backdrop window.

REPORTMOUSE

Sets the window to receive pointer movements as x,y coordinates.

Also, see the description of the IDCMP flag, MOUSEMOVE, in Chapter 8, "Input and Output Methods".

BORDERLESS

Creates a window with none of the default border padding and border lines.

NOTES: Be careful when you set this flag. It may cause visual confusion on the screen. Also, there may still be some borders if you've selected some of the system gadgets, supplied text for the window's title bar, or specified that any of your custom gadgets go in the borders.

ACTIVATE

When this window opens, it automatically becomes active.

NOTE: Use this flag carefully. It can change where the user's input is going.

NOCAREREFRESH

Set this flag if you don't want to receive messages telling you to refresh your window.

ACTIVEWINDOW and INACTIVEWINDOW

Set these flags to discover that your window has become active or inactive. You can set either or both of these flags.

The *IDCMPFlags* are listed and described in the Appendix A reference section for the *OpenWindow()* function and in Chapter 8, "Input and Output Methods". If any of these flags are set, Intuition creates a pair of Message Ports and uses them selectively for sending input to the task opening this window instead of using the Console Device.

Gadgets

A pointer to the first in the linked list of custom gadget structures which you want included in the window.

CheckMark

A pointer to an instance of a custom image to be used when menu items selected by the user are to be checkmarked. If you just want to use the default checkmark (√), set this field to NULL.

Text

A pointer to a null-terminated text string, which becomes the window title and is displayed in the window title bar.

Intuition draws the text using the colors in the *DetailPen* and *BlockPen* fields, and displays as much as possible of the window title, depending upon the current width of the title bar. You get the screen's default font.

Note: The window title is not an instance of IntuiText; it is simply a string ending in a NULL.

Type

The screen type for this window. The currently available types are WBENCHSCREEN and CUSTOMSCREEN.

NOTE: If you choose CUSTOMSCREEN, you must have already opened opened your custom screen via a call to *OpenScreen()*, and you must copy that pointer into the *Screen* field immediately below.

Screen

If your type is one of the standard screens, then this argument is ignored. If *Type* is CUSTOMSCREEN, this is a pointer to your custom screen structure.

BitMap

If you specify SUPER_BITMAP as the refresh type, this must be a pointer to your own BitMap structure. If you specify some other refresh type, Intuition ignores this field.

The following four variables are used to set the minimum and maximum size to which you allow the user to size the window. If you do not set the flag WINDOWSIZING, then these variables are ignored by Intuition.

If you set any of these variables to 0, that means you want to use the initial setting for that dimension. For example, if *MinWidth* is 0, Intuition gives this variable the same value as the opening *Width* of the window.

NOTE: To change the limits after the window is opened, call *WindowLimits()*.

> *MinWidth*
>> The minimum width for window sizing, in pixels.

> *MinHeight*
>> The minimum height for window sizing, in lines.

> *MaxWidth*
>> The maximum width for window sizing, in pixels.

> *MaxHeight*
>> The maximum height for window sizing, in lines.

WINDOW STRUCTURE

If you've successfully opened a window by calling the *OpenWindow()* function, you receive a pointer to a Window structure. This section describes some of the variables of the Window structure which may be of interest to you. This isn't a complete list of the Window variables, only the more useful ones. You'll find a complete description of the Window structure in Appendix B.

> *LeftEdge, TopEdge, Width and Height*
>> As the user moves and sizes your window, these variables will change to reflect the new parameters.

> *MouseX, MouseY, GZZMouseX, GZZMouseY*
>> These variables always reflect the current position of the Intuition pointer, whether or not your window is currently the active one. The GZZMouse variables reflect the position of the pointer relative to the inner window of Gimmezerozero windows.

> *ReqCount*
>> You can examine the ReqCount variable to discover how many requesters are currently displayed in the window.

> *WScreen*
>> This variable points to the data structure for this window's screen. If you've opened this window in a custom screen of your own making, then you should already know the address of the screen. However, if you've opened this window in one of the standard screens, this variable will point you to that screen's data structure.

RPort

> The RPort variable is a pointer to this window's RastPort. You may need the address of the RastPort when using the graphics, text, and animation functions.

BorderLeft, BorderTop, BorderRight, BorderBottom

> These variables describe the current size of the respective borders that surround the window.

BorderRPort

> With Gimmezerozero windows, this variable points to the RastPort for the outer-window, where the border gadgets are kept.

UserData

> This is a memory location that's reserved for your use. You can attach your own block of data to the window structure by setting this variable to point to your data.

WINDOW FUNCTIONS

Here's a quick rundown of Intuition functions that affect windows. For a complete description of these functions, see Appendix A.

Opening the Window

Use the following function to open a window:

> *OpenWindow (NewWindow)*

>> *NewWindow* is a pointer to a NewWindow structure. This pointer is required by many of the other functions listed below.

Menus

Use the following functions to attach and remove menus:

> *SetMenuStrip(Window, Menu)*

>> Attaches menus to a window, manages the display of windows and reports to the application when the user makes a menu choice.

ClearMenuStrip(Window)

> Removes the menu strip from a window. After this is done, the user can no longer access menus for this window. If you've called *SetMenuStrip()*, you should call *ClearMenuStrip()* before closing your window.

See Chapter 6, "Menus", for complete information about setting up your menus.

Changing Pointer Position Reports

Although you decide when opening the window whether or not you want broadcasts about pointer position, you can change this later with the following function:

> *ReportMouse(Window, Boolean)*

> Change whether or not mouse movements in this window are reported.

Closing the Window

After the user selects the close gadget, you can do whatever you need to do to clean up and then actually close the window with the following function:

> *CloseWindow (Window)*

> Closes a window and if its screen is a standard screen (but not the WorkBench) that would be empty without the window, closes the screen as well.

Requesters in the Window

The following two functions allow requesters to become active:

> *Request (Requester, Window)*

> Activates a requester in the window.

> *SetDMRequest (Window, Requester)*

> Sets up a requester that the user can bring up in the window by double-clicking the menu button.

These two functions disable requesters:

EndRequest (Requester, Window)

    Removes a requester from the window.

ClearDMRequest (Window, Requester)

    Clears the double-click requester, so that the user can no longer access it

Custom Pointers

The following functions apply if you have a custom pointer:

SetPointer (Window, Pointer, Height, Width, Xoffset, Yoffset)

    Sets up the window with a sprite definition for a custom pointer. If the window is the active one, the change takes place immediately.

ClearPointer (Window)

    Clears the sprite definition from the window and resets to the default Intuition pointer.

Changing the Size Limits

The following function changes the limits for window sizing:

WindowLimits (Window, MinWidth, MinHeight, MaxWidth, MaxHeight)

    Changes the maximum and minimum sizing of the window from the initial dimensions in the NewWindow structure. If you don't want to change a dimension, set the corresponding argument to 0. Out-of-range numbers are ignored. If the user is currently sizing the window, new limits take effect after the user releases the select button.

Changing the Window or Screen Title

The following function changes the window title after the window has already been displayed:

SetWindowTitles (Window, WindowTitle, ScreenTitle)

    Changes the window title (and screen title, if this is the active window) immediately. Either *WindowTitle* or *ScreenTitle* can be -1, 0, or a null terminated string:

-1	Don't change this one.
0	Leave a blank title bar
string	Change to the title given in this string.

Refresh Procedures

The following functions allow you to refresh your window in an optimized way:

BeginRefresh (Window)

> Initializes Intuition and layer library internal states for optimized refresh. After you call this procedure, you may redraw your entire window; only those portions that need to refreshed will actually be redrawn, while the other rendering commands will be discarded.

EndRefresh (Window)

> After you've refreshed your window, call *EndRefresh* () to restore the internal states of Intuition and the layer library.

Programmatic Control Of Window Arrangement

These functions allow you to modify the arrangement of your window as if the user was activating the associated system window gadgets:

MoveWindow (Window, DeltaX, DeltaY)

> Allows you to move the window to a new position in the screen.

SizeWindow (Window, DeltaX, DeltaY)

> You can change the size of your window with a call to this procedure.

WindowToFront (Window)

> Causes your window to move in front of all other windows in this screen.

WindowToBack (Window)

> Causes your window to move behind all other windows in this screen.

SETTING UP A SUPERBITMAP WINDOW

For a SuperBitMap window, you need to set up your own bit-map since you won't be using the screen's display memory. To set up the bit-map, you need to:

1. Create a bit-map structure.
2. Allocate memory space for the bit-map.

The general purpose graphics function *InitBitMap()* prepares a BitMap structure. A BitMap structure describes how a linear memory area is organized as a series of one or more rectangular bit-planes. Here is the specification for this function:

> *InitBitMap (bitmap, depth, bitwidth, bitheight)*

The arguments you supply are:

bitmap
> A pointer to the BitMap structure to initialize.

depth
> Number of bit-planes to set up.

bitwidth
> How wide each bit-plane should be, in bits. Should be a multiple of 16.

bitheight
> How high each bit-plane should be, in lines.

The general purpose graphics function *AllocRaster* () allocates the memory space for the Bit-Map. Here is the specification for this function:

> *AllocRaster (width, height)*

The arguments *width* and *height* are the maximum dimensions of the array in bits.

The sample code fragment below shows how you can use these functions in defining the bit-map for your SuperBitMap window:

```
#define WIDTH 640
#define HEIGHT 200
#define DEPTH 3

    struct BitMap BitMap;

    InitBitMap(&BitMap, DEPTH, WIDTH, HEIGHT);
    for (i = 0; i < DEPTH; i++)
        if ((BitMap.Planes[i] = AllocRaster(WIDTH, HEIGHT)) === 0)
            Panic("Hey!  No memory for allocating planes!");
```

SETTING UP A CUSTOM POINTER

Follow these procedures to replace the default pointer with your own custom pointer:

1. Create a sprite data structure.
2. Call *SetPointer()*. If your window is the active window, the new pointer will be attached to the window.

A sprite data structure is made up of words of data. In a pointer sprite, the first two words and the last four words are all 0's. All the other words define the appearance of the pointer, two words for each line. For example, here's the data structure for a sprite shaped like an "X":

```
/* The sprite image for our "X" should have these colors:
*
*     130000031
*     213000313
*     021303130
*     002101300
*     0000.0000   the dot is a zero that marks the pointer hot spot
*     002101300
*     021202130
*     212000213
*     120000021
*
*/
#define XPOINTER_WIDTH 9
#define XPOINTER_HEIGHT 9
#define XPOINTER_XOFFSET -4
#define XPOINTER_YOFFSET -4

USHORT XPointer[] =
    {
    0x0000, 0x0000,     /* one word each for position and control */

    0xC180, 0x4100,
    0x6380, 0xA280,
    0x3700, 0x5500,
    0x1600, 0x2200,
    0x0000, 0x0000,
    0x1600, 0x2200,
    0x2300, 0x5500,
    0x4180, 0xA280,
    0x8080, 0x4100,

    0x0000, 0x0000,
    };
```

This example sprite creates an Intuition pointer that looks like the one shown in Figure 4-6.
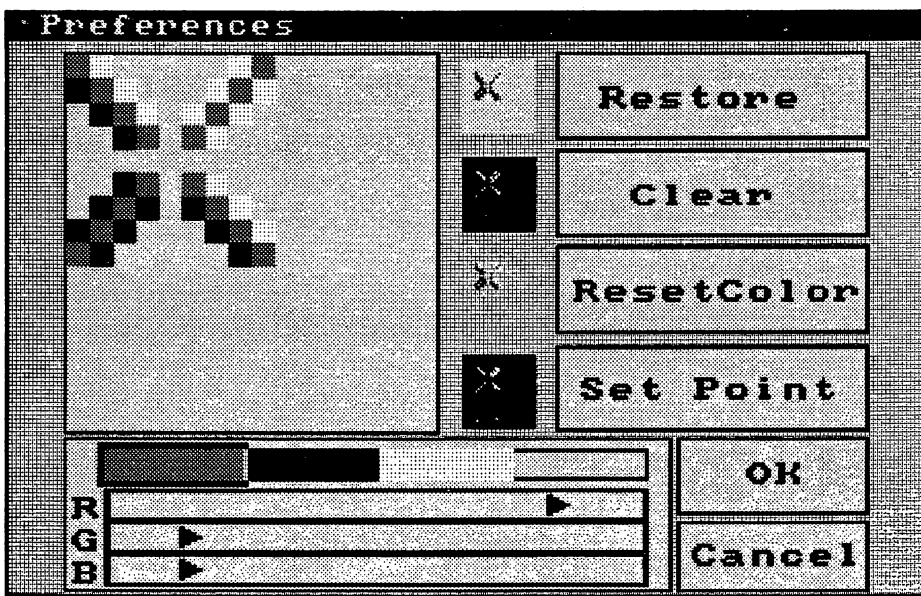
Figure 4-6: The "X"-Shaped Custom Pointer

You call *SetPointer* with the following arguments:

Window
> Pointer to the window to receive this pointer definition.

Pointer
> Pointer to the data definition of a sprite.

Height
> Height of the pointer; can be as tall as you like.

Width
> Width of the sprite (must be less than or equal to 16).

XOffset, YOffset
> Horizontal and vertical offsets for your pointer from Intuition's idea of the current position of the pointer.
>
> For instance, if you specify offsets of 0 for both, then the top left corner of your image is placed at the pointer position. If you specify an Xoffset of -7, your sprite is centered over the pointer position. If you specify an Xoffset of -15, the right edge of your sprite is over the pointer position.

Chapter 5

GADGETS

This chapter describes the workhorses of Intuition — the multi-purpose input devices called gadgets. Most of the user's input to an Intuition application can take place through the gadgets in your screens, windows, and requesters. Gadgets are also used by Intuition itself for handling screen and window dragging and depth arrangement, and window sizing and closing.

The first section of this chapter describes gadgets in general and gives details about gadget features and the different kinds of gadgets. The second section tells about the pre-defined system gadgets for windows and screens. The third section shows how you can create your own gadgets, gives specifications for gadget structures and an outline of the procedures for designing gadgets, and summarizes the functions that relate to using gadgets.

About Gadgets

Gadgets can make the user's interaction with your application consistent, easy, and fun. There are two kinds of gadgets: pre-defined system gadgets and custom application gadgets. The system gadgets help to make the user interface consistent. They are used for depth-arranging and dragging screens and for depth-arranging, dragging, sizing, and closing windows. Since they always have the same imagery and always reside in the same location, they make it easy for the user to manipulate the windows and screens of any application.

Application gadgets add power and fun to Intuition-based programs. These gadgets can be used in a multitude of ways in your programs. You can design your own gadgets for your windows, requesters, and screens.

There are 4 basic types of application gadgets:

   o   Boolean gadgets elicit true/false or yes/no kinds of answers from the user.

   o   Proportional gadgets are very flexible devices that you use to get some kind of proportional setting from the user or to simply display proportional information. With the proportional gadget, you can use imagery furnished by Intuition or design any kind of image you want for the slider or knob used to pick a proportional setting.

   o   String gadgets are used to get text from the user. A number of editing functions are available for users of string gadgets.

   o   The integer gadget is a special class of string gadget which allows the user to enter an integer value only.

Although system gadgets are always in the borders of windows and screens, your own gadgets can go anywhere and can be any size or shape. You can choose from the following ways of

highlighting gadgets to emphasize that the gadget has been selected:

o alternate image or alternate border

o a box around the gadget

o color change

You can elect to have your gadgets change in size as the user sizes the window so they remain proportional to the size of the window. Also, window gadgets can be located relative to one of the window's borders so they move with the borders as the user shapes or sizes the window. If you want the gadget in the border like the system gadgets, Intuition can adjust the border size accordingly.

Typically, the user selects a gadget by moving the pointer within an area called the select box; you define the dimensions of this area. Next, the user takes some action which varies according to the type of gadget. For a Boolean gadget, the user may simply choose an action by clicking the mouse button. For a string or integer gadget, a cursor appears and the user enters some data from the keyboard. For a proportional gadget, the user might either move the knob by dragging it with the mouse or click the mouse button to move the knob by a set increment.

Although you attach a list of pre-defined application gadgets when you define a screen, window, requester, or alert structure, you can make changes to this list later. You can enable or disable gadgets, add or remove gadgets, modify the internal states of gadgets, and redraw some or all of the gadgets in the list.

When one of your application gadgets is selected by the user, your program learns about it from either the IDCMP or the Console Device. See Chapter 8, "Input and Output Methods", for details about the messages you get.

System Gadgets

Intuition automatically attaches system gadgets to every screen. For windows, you specify which system gadgets you want. The system gadgets for screens are dragging and depth arrangement. The system gadgets for windows are dragging, depth arrangement, sizing, and closing.

System gadgets have fixed, well-publicized locations in screens and windows, as shown in Table 5-1.

Table 5-1: System Gadget Placement in Windows and Screens

SYSTEM GADGET	LOCATION
Sizing	Lower-right
Dragging	Entire title bar in all areas not used by other gadgets
Depth arrangers	Top-right
Close	Top-left

Your program need never know that the user selected a system gadget (with the exception of the close gadget), if you don't want it to. You can attach these gadgets to your windows and let Intuition do the work of responding to the user's wishes.

Figure 5-1: System Gadgets in a Low-Resolution Window

The following paragraphs describe each system gadget in detail.

SIZING GADGET

When the user selects the window sizing gadget, Intuition is put into a special state. The user is allowed to elongate or shrink a rectangular outline of the window until the user achieves the desired new shape of the window and releases the select button. The window is then reestablished in the new shape. This may involve asking the application to redraw part of its display. For more information about the application's responsibilities in sizing, see the discussion about

preserving the display in Chapter 4, "Windows".

You attach the sizing gadget to your window by setting the WINDOWSIZING flag in the *Flags* variable of the NewWindow structure when you open your window.

If you are using the IDCMP for input, you can elect to receive a message when the user attempts to size the window. There is a special IDCMP flag, SIZEVERIFY, that allows you to hold off window sizing until you are ready for it. See Chapter 8, "Input and Output Methods", for more information about SIZEVERIFY.

DEPTH ARRANGEMENT GADGETS

The depth arrangers come in pairs—one for bringing the window or screen to the front of the display and one for sending the window or screen to the back. Notice that the actual depth arrangement of windows and screens is transparent to your program. The only time you might learn about it at all is indirectly when Intuition notifies your program that it needs to refresh its display.

You attach the depth arrangement gadgets to your window by setting the WINDOWDEPTH flag in the *Flags* variable of the NewWindow structure when you open your window. You get screen depth arrangement gadgets automatically with every screen you open.

DRAGGING GADGET

The dragging gadgets are also known as "drag bars" because they occupy the entire title bar area that's not taken up by other gadgets. Users can slide screens up and down, much like some classroom blackboards, to reveal more pertinent information. They can slide windows around on the surface of the screen to arrange the display any way they want.

In dragging a window, the user actually drags a rectangular outline of the window to the new position and releases the select button. The window is then reestablished in its new position. As in window sizing, this may involve asking the application to redraw part of its display.

If you want the window drag gadget, set the WINDOWDRAG flag in the *Flags* variable of the NewWindow structure when you open your window. You get the screen drag gadget automatically with every screen you open.

CLOSE GADGET

The close gadget is a special case among system gadgets, because Intuition notifies your program about the user's intent, but doesn't actually close the window. When the user selects the close gadget, Intuition modifies some internal states and then broadcasts a message to your program. It's then up to the program to call *CloseWindow()* when ready. There may be actions that you need or want to take before the window closes; for instance, bring up a Requester to verify that the user really wants to close that window.

To get the window close gadget, set the WINDOWCLOSE flag in the *Flags* variable of the NewWindow structure when you open your window.

Application Gadgets

Intuition gadgets imitate real-life gadgets. They are the switches, knobs, controllers, gauges and keys of the Intuition environment. You can create almost any kind of gadget that you can imagine, and you can have it do just about anything you want it to do. You can create any visual imagery that you like for your gadgets, including combining text with hand-drawn imagery or supplying coordinates for drawing lines. You can also choose a highlighting method to change the appearance of the gadget after it is selected. All of this flexibility gives you the freedom to create gadgets that mimic real devices like light switches or joysticks, as well as the freedom to create devices that satisfy your own unique needs.

RENDERING GADGETS

You can draw your gadgets by hand, specify a series lines for a simple line gadget, or have no imagery at all.

Hand-Drawn Gadgets

Because you are allowed to supply a hand-drawn image, there's no limit to the designs you can create for your gadgets. You can make them simple and elegant, or whimsical and outrageous. You design the imagery using one of Amiga's many art tools, and then translate your design into an instance of an Image structure. Figure 5-2 shows an example of a gadget made of hand-drawn imagery. It also shows how you can use an alternate image when the gadget is selected.
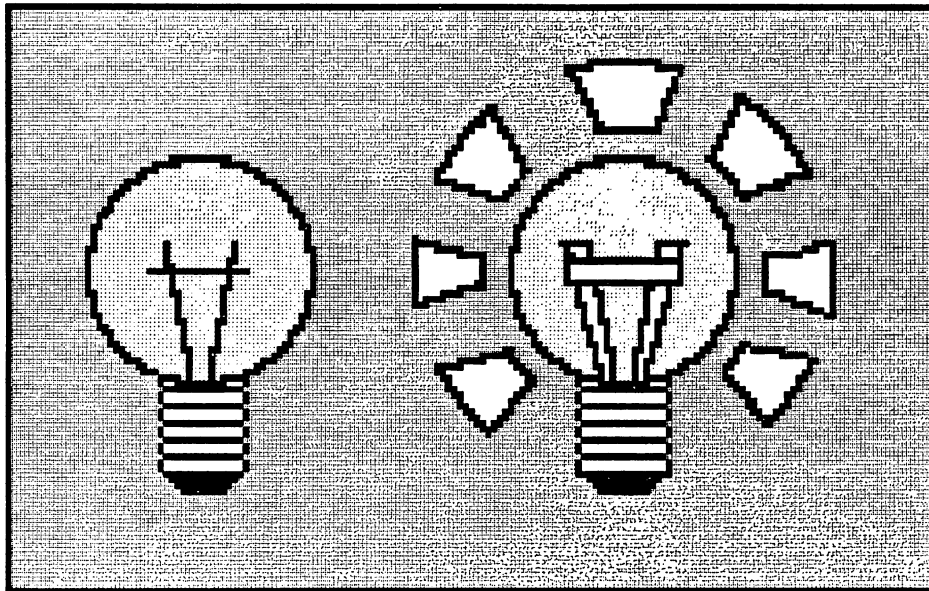
Figure 5-2: Hand-drawn Gadget — Unselected and Selected

You incorporate a hand-drawn image into your gadget by:

o   setting the GADGIMAGE flag in the gadget variable *Flags* to select that this gadget should be rendered as an Image

o   putting the address of your Image structure into the gadget variable *GadgetRender*

For more information about creating an Image structure, see Chapter 9, "Images, Line Drawing, and Text".

Line-Drawn Gadgets

You can also create simple designs for gadgets by specifying a series of lines to be drawn as the imagery of your gadget. You can have these lines go around or through the select box of your gadget, and specify more than one group of lines, each with its own color and drawing mode. You create line-draw imagery for your gadget by first deciding on the color and placement of the lines.

Figure 5-3 shows an example of a gadget that uses line-draw imagery. It also shows an example of the complement mode of highlighting a gadget when it's selected. Furthermore, it shows additional text that's been included in the gadget imagery.
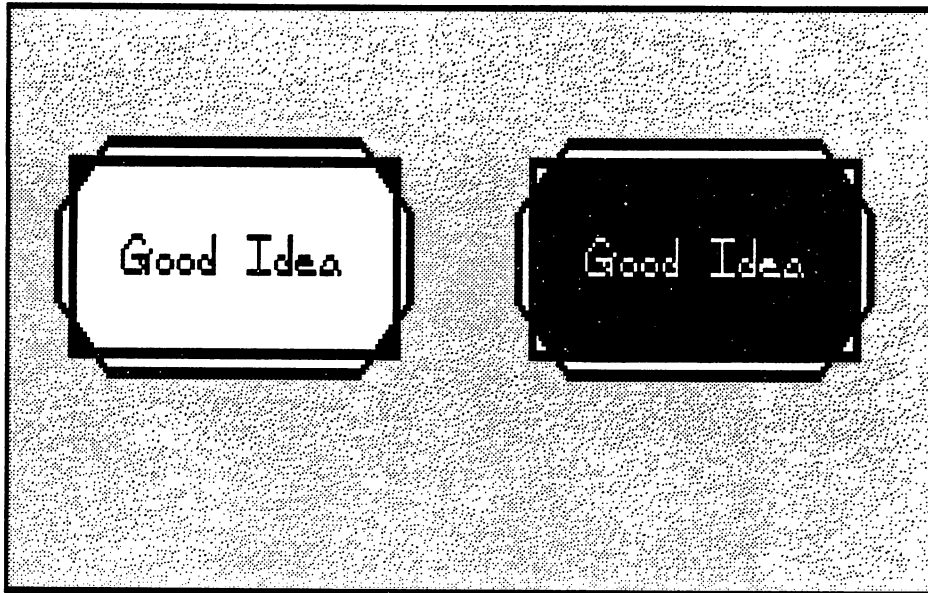
Figure 5-3: Line-draw Gadget — Unselected and Selected

After deciding on the placement and color of your lines, you create an instance of a Border structure to describe your design. You incorporate the Border structure of your line-draw imagery into your gadget by:

o   not setting the GADGIMAGE flag in the gadget's *Flags* variable, thus specifying that this is a Border, not an Image

o   putting the address of your Border structure into the *GadgetRender* variable of your gadget

For more information about creating a Border structure, see Chapter 9, "Images, Line Drawing, and Text".

Gadgets Without Imagery

You can create gadgets that have no imagery at all. You may, for instance, only want to follow the user's mouse activity without cluttering the display with unnecessary graphics. An example of this is the window and screen dragging gadget, where no actual imagery is rendered. The title bar itself sufficiently implies the imagery of the gadget.

You specify no imagery by not setting the gadget's GADIMAGE flag, and by setting the *GadgetRender* variable to NULL.

USER SELECTION OF GADGETS

When the user positions the pointer over a gadget and presses the select button, that gadget becomes "selected" and is immediately highlighted. Intuition has two different ways of notifying your program about gadget selection.

If you want to find out immediately when the gadget has been selected, you can set the GADGIMMEDIATE flag in the *Activation* field of the Gadget structure. When the user selects that gadget, you will get an IDCMP event of class GADGETDOWN. If you set only this flag, then you will hear nothing more about that gadget until it's selected again.

On the other hand, if you want to be absolutely sure that the user wanted to select the gadget, you can set the RELVERIFY flag (for "Release Verify"). When RELVERIFY is set and the user selects the gadget, you will only learn that the gadget was selected if the user still has the pointer over the select box of the gadget when the select button is released. You may want to know this about some gadget selections, for instance the close window gadget, where the consequences may be serious. If you set the RELVERIFY flag, you will learn about these events via an IDCMP message of the class GADGETUP. There are two main benefits to RELVERIFY:

  o   The unsure user gets one last chance to reconsider, and

  o   It helps avoid casual errors caused by the user brushing against or resting fingers on the mouse button.

If you want to receive both a GADGETDOWN and GADGETUP message, set both the RELVERIFY and GADGIMMEDIATE flags.

GADGET SELECT BOX

To use a gadget, the user begins by moving the pointer into the gadget *select box.* You define the location and dimensions of the select box In the Gadget data structure. The location is an offset from one of the corners of the display element (window, screen, or requester) that contains the gadget. You place the left and top coordinates in the *LeftEdge*, and *TopEdge* fields of the gadget structure.

LeftEdge describes a coordinate that is either an absolute offset from the left edge of the element, or a negative offset from the <u>current</u> right edge. The offset method is determined by the GRELRIGHT flag. For instance:

  o   if GRELRIGHT is cleared and *LeftEdge* is set to 5, the select box of the gadget starts 5 pixels from the left edge of the display element

  o   if GRELRIGHT is set and *LeftEdge* is set to -5, the select box of the gadget starts 5 pixels left of the (current) right edge

In the same way, *TopEdge* is either an absolute offset from the top of the element, or a negative offset from the current bottom edge according to how the flag GRELBOTTOM is set.

o   If GRELBOTTOM is cleared, *TopEdge* is an absolute offset from the top of the element.

o   If GRELBOTTOM is set, *TopEdge* is a negative offset from the current bottom edge.

Similarly, the height and width of the gadget can be absolute or relative to the height and width of the display element in which it resides. If you set the width of a window gadget to -28, for example, and you set the gadget's GRELWIDTH flag, then the gadget's select box will always be 28 pixels less than the width of the window. If GRELWIDTH is not set and you set the width of the gadget to 28, the gadget's select box will always be 28 pixels wide. The GRELHEIGHT flag has the same effect on the height of the gadget select box.

Here are some examples of how you can take advantage of the special relativity modes of the select box.

o   Consider the Intuition window sizing gadget. The LeftEdge and TopEdge of this gadget are both defined relative to the right and bottom edges of the window. No matter how the window is sized, the gadget always appears in the lower-right.

o   In the window dragging gadget, the LeftEdge and TopEdge are always absolute in relation to the top-left corner of the window. Also, the Height is always an absolute quantity. The Width of the gadget, however, is defined to be zero. When the Width is combined with the effect of the GRELWIDTH flag, the dragging gadget is *always* as wide as the window.

o   Assume that you are designing a program that has several requesters, and each requester has a pair of "OK" and "CANCEL" gadgets in the lower left and lower right corners of the requester. You can design "OK" and "CANCEL" gadgets that can be used in any of the requesters simply by virtue of their positions relative to the lower left and lower right corners of the requester. Regardless of the size of the requesters, these gadgets appear in the same relative positions.

The GRELRIGHT, GRELBOTTOM, GRELWIDTH, and GRELHEIGHT flags are set in the *Flags* field of the gadget structure.

GADGET POINTER MOVEMENTS

If you set the FOLLOWMOUSE flag for a gadget, you will receive mouse movement broadcasts as long as the gadget is selected. You may want to follow the mouse, for example, in a sound effects program where you use the mouse movement to change some quality of the sound. You might also want to follow the mouse in a game where you use it for aiming a weapon.

The broadcasts you receive differ according to the following flag settings:

o   If you set the GADGIMMEDIATE *and* RELVERIFY flags, you learn that the gadget was selected, get some mouse reports (at least one), and find out that the mouse button was released over the gadget.

o   If you set only the GADGIMMEDIATE flag, you learn that the gadget was selected, and get some mouse reports. Then the mouse reports will stop (when the user releases the select button, though you'll have no way of knowing for sure that this has happened).

o   If you set only the RELVERIFY flag, you get some mysterious, anonymous mouse reports (which may be just what you want) followed, perhaps, by a release event for a gadget.

o   If you set neither the GADGIMMEDIATE nor the RELVERIFY flag, you get only mouse reports. This may be exactly what you want.

The FOLLOWMOUSE, GADGIMMEDIATE, and RELVERIFY flags are all set in the *Activation* field of the Gadget structure.

GADGETS IN WINDOW BORDERS

In windows only, you can elect to put your own gadgets in the borders. In the Gadget structure, you set one or more of the border flags to tuck your gadget away into the window border. Setting these flags also tells Intuition to adjust the size of the window's borders to accommodate the gadget.

Note that the borders are adjusted only when the window is opened. Although you can add and remove window gadgets after the window is opened, with *AddGadget()* and *RemoveGadget()*, Intuition does not readjust the borders.

Note also that you can put a given gadget in more than one border by setting more than one border flag. Ordinarily, it only makes sense to put a gadget into two adjoining borders. If you set both side border flags or both the top and bottom border flags for a particular gadget, you get a window that's all border.

The flags are RIGHTBORDER, LEFTBORDER, TOPBORDER, and BOTTOMBORDER; and you set them in the *Activation* field of the gadget structure.

MUTUAL EXCLUDE

NOTE: As of the time this was published, this feature had not been implemented.

If a gadget is selected and a bit has been set in the *MutualExclude* variable of the gadget, the gadget corresponding to that bit (for example, bit 0 set refers to the first gadget in the gadget list, bit 2 to the third, and so on) becomes disabled. Intuition sets or clears the appropriate bits in the disabled gadgets and changes the display to reflect the new state of affairs. It's up to your program to note internally, as needed, that the other gadgets have been disabled. Note that there is no reason to limit yourself to 32 gadgets in the gadget list. However, the mutual exclude works only on the first 32 gadgets in a list.

GADGET HIGHLIGHTING

You can change the appearance of a selected gadget to let the user know that the gadget has indeed been selected.

You select a highlighting method by setting one of the highlighting bits in *Flags*. Note that you must specify one of the highlighting values. If you don't want any highlighting, then set the GADGHNONE bit in the gadget's *Flag* field.

The methods of highlighting after selection are described below.

Highlighting by Color Complementing

You can highlight by complementing all of the colors in the gadget's select box. In this context, complementing means the complement of the binary number used to select a particular color register. For example, if the color in color register 2 is used (binary 10) in some of the pixels in the selected gadget, those pixels get changed to whatever color is in color register 1 (binary 01). Figure 5-3 (the good-idea figure) shows an example of complement highlighting. Note that only the select box of the gadget is complemented, while the text, which is outside of the select box, is not disturbed. See Chapter 9, "Images, Line Drawing, and Text", for more information about complementing and about color in general.

Highlighting by Drawing a Box

To highlight by drawing a simple border around the gadget's select box, set the GADGHBOX bit in the *Flags* field.

Highlighting with an Alternate Image or Alternate Border

You can supply an alternate Image or Border imagery as highlighting. When the gadget is selected, the alternate Image or Border is displayed in place of the non-highlighted imagery. If the non-highlighted imagery is an Image, the highlight imagery should be an Image as well; the same is true for Border imagery. Figure 5-2 (the light-bulb illustration) shows an example of this method of highlighting. For this highlighting method, you should set the *SelectRender* field of the gadget structure to point to the Image structure or Border structure for the alternate display.

An Image or Border structure contains a set of coordinates that specifies its location when displayed. Intuition renders the image or border relative to the top left corner of the gadget's select box.

For information about how to create an Image or Border structure, see Chapter 9, "Images, Line Drawing, and Text".

GADGET ENABLING AND DISABLING

You can disable a gadget so that it cannot be selected by the user. When a gadget is disabled, its image is "ghosted", and it cannot be selected. Ghosted means that the normal image is overlaid with a pattern of dots, thereby making the image less distinct. Before you first submit your gadget to Intuition, you initialize whether your gadget is disabled by setting or not setting the GADGDISABLE flag in the gadget's *Flags* field. If you always want the gadget to be enabled, you can ignore this flag.

After you've submitted a gadget for Intuition to display, you can change its current enable state by calling *OnGadget()* or *OffGadget()*. If it's a requester gadget, the requester must currently be displayed. If you use *OnGadget()* to enable a previously disabled gadget, its image is returned to its normal, non-ghosted, state.

BOOLEAN GADGET TYPE

This is a simple *TRUE* or *FALSE* gadget. For Boolean gadgets, you can choose from two methods of selecting—*hit select* or *toggle select*:

- o Hit select means that when the gadget is hit (that is, when the user moves the pointer into the select box and presses the mouse select button) the gadget becomes selected and the select highlighting method is employed. When the mouse select button is released, the gadget is unselected and unhighlighted.

- o Toggle select means that when the gadget is hit, it toggles between selected and unselected. That is, if the user selects the gadget, it remains selected when the user releases the button. To "unselect" the gadget, the user has to repeat the process of hitting the gadget. You can have the imagery reflect the selected/unselected state of the gadget by supplying an alternate image as the highlighting mode of the gadget. When the gadget is selected, the chosen highlighting method is employed.

Note the following two flags that have an effect upon toggle-selection:

- o If you want a gadget to be toggle-selected, you need to set the TOGGLESELECT flag in the *Activation* field of the Gadget structure.

- o The SELECTED flag in Gadget structure *Flags* determines the initial and current on/off selected state of a toggle-selected gadget. If SELECTED is set, the gadget will be highlighted. You can set the SELECTED flag before submitting the gadget to Intuition if you like. You can examine this flag at any time to determine whether the user has selected this gadget.

If a Boolean gadget is selected by the user, the application will hear about it. If it's never selected, the application will never know. This differs from string or proportional gadgets, which always are set to some value, even if that value is the one initialized by you.

PROPORTIONAL GADGET TYPE

This is an enormously flexible input device. You can use one of these to get a proportional setting from the user or to display a proportional value to the user. Best of all, you can use the same gadget to accomplish both of these feats.

The user can adjust the setting of a proportional gadget to specify how much of some measurable data or attribute is desired. For instance, the user may adjust a proportional gadget to specify a location in a text file or a desired volume setting. The current setting of a proportional gadget may also be set by the program as an indicator of how much of some measurable data or attribute is visible or available. For instance, the proportional gadget of a text editor's

window might show how many lines are currently being displayed out of the total lines in the text file. A graphics program may want to allow the user to set the amount of red, green, and blue in a color, and so provides a proportional gadget for each of the three hues. The graphics program would initialize these settings to designate how much red, green, and blue is already contained in the color. An audio program may deal with the volume of the sound being produced by providing a gadget that allows the user to set the volume and to see what the current volume is in relation to the highest and lowest possible volume settings.

Proportional gadgets can do all of these things for you, and much more, because they can take many shapes and sizes and get fractional settings on either the vertical or horizontal axis, or both.

A proportional gadget has several parts that work together to give the gadget its flexibility. They are the the *pot* variables, the *body* variables, the *knob*, and the *container*.

o   The *HorizPot* and *VertPot* variables contain the actual proportional values. The word *pot* is short for *potentiometer*, which is an electrical analog device that can be used to adjust some variable value. The proportional gadget pots enable the user or program to set how much of the total data is visible or available. Because they represent fractional parts of a whole, the values in these variables ranges from 0 to (almost) 1. The data, then, ranges from none visible or available to all of it visible or available.

There are two pot variables because proportional gadgets are adjustable on the horizontal axis or the vertical axis or both. For example, a gadget that allows the user to center the screen on the video display, or center his gunsights on a fleeing enemy, has to be adjustable on both axes.

Pot variables are typically initialized to 0, and change while the user is playing with the gadget. You can initialize the pot variables to whatever you want. In the case of the color gadget, you might want to initialize them to some current color. You may read the values in the pots any time you want after you have submitted the gadget to the user via Intuition. They will always have the current settings as adjusted by the user.

o   The *HorizBody* and *VertBody* variables describe the increment, or typical step value, by which the pot variables change. For example, the proportional gadgets for color mixing might allow the user to add or subtract a color by 1/16 of the full value each time, as there are 16 possible settings for each RGB (red, green, blue) component of a color on the Amiga. The proportional gadget for centering the screen might allow the user to move the screen vertically a line at a time, or you may choose to have the step increment be many lines, and leave the fine-resolution tuning to the use of the gadget's knob.

Body variables are also used in conjunction with the auto-knob (described below) to display for the user how much of the total quantity of data is directly available. For instance, if the user is working on a text file that's 15 lines long, and 5 lines of the file are currently visible in the window, then you can graphically represent the total size of the file by setting the body variable to one third (0xFFFF / 3 = 0x5555). In this case, the auto-knob would fill one third of the container, which represents the proportion of the visible text lines to the total number of text lines. Also, the user can tell at a glance that clicking in the container (not on the knob) will advance the text file by one-third in any direction, to the next "window" of data.

You can set the two body variables to the same or different increments. When the user clicks in the container, your pot variables are adjusted by the amount set in the body variables.

o   The *knob* is the object actually manipulated by the user to change the pot variables by the increments specified in the body variables. The knob is directly analogous to proportional controls like the volume knob on a radio, if the Intuition knob is restricted to one axis of movement. If the knob is free to move on both axes, it's more analogous to, say, a control-stick of an airplane. The user can move the knob by placing the pointer on it and dragging it on the vertical or horizontal axis or by moving the pointer near it (within the select box) and clicking the mouse button. With each click, the pot variable is increased or decreased by one increment, defined by the settings of the Body variables. The current position of the knob reflects the pot value. For instance, in the color selection gadget, the knob slides in a long narrow container. As the user moves the knob to the right, more of that color is added. When the knob is halfway along the container, the value in *HorizPot* is also half-way. For a picture of this color selection gadget, see the Preferences display in Figure 11-2.

You can design your own imagery for the knob or use Intuition's very handy *auto-knob.* The auto-knob is a rectangle that can move on either axis and changes its length or height according to the current body settings. The auto-knob also proportionally changes in size when the user sizes the window. Therefore, you can place an auto-knob in a proportional gadget that sizes relative to the size of a window, and the auto-knob will always be proportionally correct. For example, consider a proportional gadget with auto-knob being used as a scroll bar in the right border of a window. If the VertBody variable is set to show that 1/3 of a text file is being displayed in the window, the auto-knob fills 1/3 of the container. If the user makes the window (and therefore the container) larger, the auto-knob gets larger, too, so that it still visually represents 1/3. For an example of such a scroll bar, see Figure 5-4. This is yet another visual aide for the user, which makes the user-interface of the Amiga as intuitive to use as possible.

o   The *container* is the area in which the knob can move. It is actually the select box of the gadget. The size of the container, like that of any other gadget select box, can be relative to the size of the window.

The pot variable is a 16-bit word that contains a value ranging from 0 to 0xFFFF. This value range represents a fixed-point fraction that ranges from 0 to (almost) 1. You need to convert the current setting of the pot variable to a number that you can use. The proportional gadget example below shows how to do this conversion.

```
/*****************************************************************
*
* Conversion of a pot variable
*
****************************************************************/

#define MAXSECONDS    4        /* an arbitrary assignment */
#define MILLION       1000000    /* a real assignment */


    LONG RealTime;
    SHORT Seconds, MicroSeconds;
```

The next line converts the 16-bit fraction into a 32-bit intermediate value which expresses

integer and fractional parts of the constant MAXSECONDS. The integer portion is in the upper 16 bits, and the fractional remainder is in the lower 16.

$$\text{RealTime} = (\text{PropInfo.HorizPot} + 1) * \text{MAXSECONDS};$$

This line gets the number of seconds, which is the integer portion:

$$\text{Seconds} = \text{RealTime} >> 16;$$

Because the lower 16 bits represent only a fraction, we must multiply this by some other meaningful constant to have it mean something real. Because we want the fractional portion to represent microseconds and there are a million microseconds to the second, we'll multiply the fraction by a MILLION. Then, in the integer portion (the upper 16 bits) we'll find the whole number of millionths of a second contained in *RealTime*. (By the way, in the lower 16 bits of the multiplication, which we shift away into the bottomless bit bucket, is a fraction representing the fractional part of a millionth of a second contained in *RealTime*. If we wanted to be technically accurate, we should test whether this fraction is greater than or equal to 0x8000 (one half), and round our MicroSeconds result up if it is.)

$$\text{MicroSeconds} = ((\text{RealTime} \ \& \ \text{0xFFFF}) * \text{MILLION}) >> 16;$$

You set up a proportional gadget like any other gadget except for the extra PropInfo data structure (shown below under "Using Application Gadgets"). Carry out the following procedures to set up the PropInfo structure:

o If you want the auto-knob, set the AUTOKNOB flag and set *GadgetRender* to point to an Image. In this case, you don't initialize the Image structure.

   If you want your own knob imagery instead, *GadgetRender* points to a real Image or Border structure.

o Set either or both of the FREEHORIZ and FREEVERT flags according to the direction(s) you want the knob to move.

o Initialize either or both of the *HorizPot* and *VertPot* variables to their starting values.

o Set either or both of the *HorizBody* and *VertBody* variables to the increment you want. If there is no data to show or the total amount displayed is less than the area in which to display it, set the body variables to the maximum (0xFFFF).

o The remaining variables and flags are used by Intuition.

In the Gadget structure, set the *GadgetType* field to PROPGADGET and set the *SpecialInfo* field to point to an instance of a PropInfo structure.

After the gadget is displayed, you can call *ModifyProp()* to change the flags and the pot and body variables. The gadget's internal state will be recalculated and the imagery will be redisplayed to show the new state.

If you receive a message telling you that the user has played with this gadget, you can examine the KNOBHIT flag in the PropInfo structure. This flag tells you whether the user hit the knob, or hit in the container but not on the knob itself. If the flag is set, the user has hit the knob

and moved it.

STRING GADGET TYPE

A string gadget prompts the user to enter some text. Like a proportional gadget, a string gadget can also be used in many different ways. String gadgets also require their own special structure, called the StringInfo structure.

A string gadget consists of a container and buffers to hold the strings. You supply two buffers for the string gadget. The input buffer contains the "initial" string, and the other is an optional "undo" buffer. You pre-set the initial string. By pre-setting the string, you give the user the choice of editing the initial string or simply accepting the default initial string.

If you've given Intuition an undo buffer, the string in the gadget reverts to the initial setting when the user types "Right AMIGA - Q". (To type this key sequence, the user holds down the right AMIGA key while pressing the Q key.)

You specify the size of the container into which the user types the string. Like the container of the proportional gadget, the container for the string gadget is its select box. As the user types text into a string gadget, the characters appear in the gadget's container.

You can change the justification of the string as it's displayed in the container. The default is left-justification. If the flag STRINGCENTER is set, the text is center-justified; if STRINGRIGHT is set, the text is right-justified.

A very important and useful feature of the string gadget is that you can supply a buffer to contain more text than will fit in the container. This allows you to get text strings from the user that are much longer than the visible portion of the buffer. Intuition maintains the cursor position and scrolls the text in the container as needed.

You can initialize the input buffer to any starting value, as long as the initial string is terminated with a null. If you want to initialize the buffer to the null string (no characters), you must put a null character in the first position of the buffer. After the gadget is de-selected by the user (either by hitting the RETURN key or by using the mouse to select some other operation), you can examine this buffer to discover the current string.

String gadgets feature "auto-insert", which allows the user to insert ASCII characters wherever the cursor is. The following simple editing functions are available to the user:

Table 5-2: Editing Keys and Their Functions

KEY(S)	FUNCTION
← or →	Move the cursor around the current string.
SHIFT ← or →	Move the cursor to the beginning or end of current string.
DEL	Delete the character under the cursor.
BACKSPACE	Delete character to left of cursor.
RETURN	Terminate input and de-select the gadget. If the RELVERIFY activation flag is set, notify the program that the user has selected this gadget.
Right AMIGA - Q	Undo (cancel) the last editing change to the string. To allow the user to undo, you have to supply another buffer as large as the input buffer. All string gadgets can share the same undo buffer

You can supply any type of image for the rendering of this gadget—Image or Border type—or no image at all. For this release, you must specify that the highlighting is of type GADGHCOMP (complementary), and you cannot supply an alternate image for highlighting.

The string gadget inherits the font and input attributes of the window or screen. If you haven't done anything fancy, the strings will appear in the default font with simple ASCII key translations. If you are using the Console Device for input, you can set up alternate key-mapping any way you like. If you do, Intuition will use your key map. Nevertheless, you get the default system font. See the *Amiga ROM Kernel Manual* for more information about the Console Device and key-mapping.

For a string gadget, you set the *GadgetType* field to STRGADGET in the Gadget structure. Also set the *SpecialInfo* field to point to an instance of a StringInfo structure. This structure contains buffer and container information.

INTEGER GADGET TYPE

The integer gadget is really a special sort of string gadget. You initialize it as you do a string gadget, except that you also set the flag LONGINT in the gadget's *Activation* variable. The user interacts with an integer gadget using exactly the same rules, except Intuition filters the input and allows the user to enter only a unary sign and digits. The integer gadget returns a signed 32-bit integer in the StringInfo variable *LongInt*.

To initialize an integer gadget, in this release, you need to pre-set the buffer by putting an initial integer string in it. It is *not* sufficient to initialize an integer gadget by setting a value in the *LongInt* variable.

To specify that this string gadget is an integer gadget, set the flag LONGINT in the gadget's *Activation* variable.

COMBINING GADGET TYPES

You can make some very useful gadgets by combining types. As an example, you can make a horizontal or vertical scroll bar with a proportional gadget and two Boolean gadgets. Figure 5-4 shows an example.
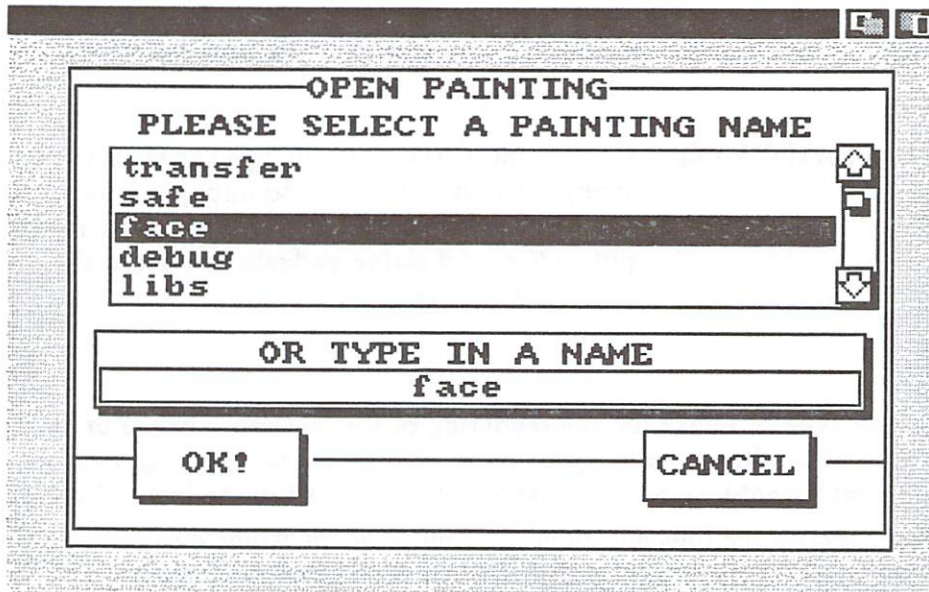
Figure 5-4: Example of Combining Gadget Types

If the scroll bar goes in the right border of the window, then you may wish to place the system sizing gadget in the right border by setting the *Flag* SIZEBRIGHT in the window structure. Remember that the sizing gadget has to fit in either the right or bottom border. You get to choose which. If you are going to cause the right edge border to be wide enough to accommodate a scroll bar, then you might as well put the sizing gadget there, too.

Using Application Gadgets

To create application gadgets, you follow these steps:

1. Create a structure for each gadget.
2. Create a linked list of gadgets for each display element (screen, window, or requester) that has gadgets attached to it.
3. Set the *Gadgets* variable in your screen, window, or requester structure to point to the first gadget in the list.

Each gadget structure includes specifications for:

o   either an image or a border or NULL for no imagery

o   the select box of the gadget, which is the zone Intuition uses to detect if the user is selecting that gadget

o   left and top offsets that are either are absolute or relative to the current borders of the window, requester, or screen containing the gadget

o   width and height dimensions that are absolute or relative to the current size of the window, requester, or screen containing the gadget

o   gadget type—Boolean, integer, proportional, or string

o   the method of highlighting the gadget, if any

o   how you want Intuition to behave while the user is playing with your gadget

GADGET STRUCTURE

Here is the general specification for a gadget structure:

```
struct Gadget
{
        struct Gadget *NextGadget;
        SHORT LeftEdge, TopEdge, Width, Height;
        USHORT Flags;
        USHORT Activation;
        USHORT GadgetType;
        APTR GadgetRender;
        APTR SelectRender;
        struct IntuiText *GadgetText;
        LONG MutualExclude;
        APTR SpecialInfo;
        USHORT GadgetID;
        APTR UserData;
}
```

Here are the meanings of the variables and flags in the gadget structure:

NextGadget

A pointer to the next gadget in the list. The last gadget in the list should have a NextGadget value of NULL.

LeftEdge, TopEdge, Width, Height

These variables describe the location and dimensions of the select box of the gadget. Both locations and dimensions can be either absolute or relative to the edges and size of the window, screen, or requester that contains the gadget.

LeftEdge and *TopEdge* are relative to one of the corners of the base structure, according to how GRELRIGHT and GRELBOTTOM are set in the *Flags* variable below.

Width and *Height* can be either absolute dimensions or a negative increment to the width and height of a requester, screen, or alert or the current width and height of a window, according to how the GRELWIDTH and GRELHEIGHT flags are set below.

Flags

The *Flags* field is shared by your program and Intuition. See the section below called "Flags" for a complete description of all the flag bits.

Activation

This field is used for information about some gadget attributes. See the *Activation Flags* section below for a description of the various flags.

GadgetType

This field contains information about gadget type and in what sort of display element the gadget is to be displayed.

You <u>must</u> set one of the following flags to specify the type:

BOOLGADGET
Boolean gadget type.

STRGADGET
String gadget type.

For an integer gadget, also set the LONGINT flag. See the "Flags" section below.

PROPGADGET
Proportional gadget type.

The following flags tell Intuition if the gadget is for a screen, requester, or Gimmezerozero window:

SCRGADGET
Set this bit if this gadget is a screen gadget, clear it if not.

GZZGADGET
If this gadget is for a Gimmezerozero window, setting this flag puts the gadget in the special bit-map for gadgets and borders (and out of your inner window). If you don't set this flag, the gadget will go into your inner window. If the destination of this gadget is not a Gimmezerozero window, clear this bit.

REQGADGET
If this gadget is a Requester gadget, set this bit, otherwise clear it.

GadgetRender
This is a pointer to the Image or Border structure containing the graphics of this gadget.

If this field is set to NULL, no rendering will be done.

NOTE: To tell Intuition what sort of data is pointed to by this variable set or clear the *Flag* bit, GADGIMAGE.

SelectRender
This field contains a pointer to an alternate Image or Border for highlighting.

NOTE: You specify that you want *SelectRender* by setting the GADGHIMAGE flag. You specify which type, Image or Border, by setting the same GADGIMAGE bit that you set for *GadgetRender* above. SelectRender must point to the same data type as GadgetRender.

GadgetText
If you want text printed after this gadget is rendered, set this field to point to an IntuiText structure. The offsets in the IntuiText structure are relative to the top left of the gadget's select box.

Set this field to NULL if the gadget has no associated text.

MutualExclude
When this feature is implemented, you will use these bits to describe which if any of the other gadgets are mutually excluded by this one.

Meanwhile, Intuition ignores this field.

SpecialInfo
> If this gadget is of type proportional, string, or integer, this variable points to an instance of a PropInfo or StringInfo data structure, respectively. The structure contains the special information needed by the gadget.
>
> If the gadget is not of type proportional, string, or integer, this variable is ignored.

GadgetID
> Strictly for your own use. Assign any value you'd like here. This variable is ignored by Intuition.
>
> Typical uses in C are in switch and case statements, and in assembly language, table lookup.

UserData
> A pointer to any general data you'd care to associate with this particular gadget. This variable is ignored by Intuition.

FLAGS

Following are the flags you can set in the *Flags* variable of the gadget structure.

GADGHIGHBITS
> Combinations of these bits describe what type of highlighting you want when the user has selected this gadget. There are four highlighting methods to choose from. You must set one of the four flags below.

GADGHCOMP
> Complements all of the bits contained within this gadget's select box.

GADGHBOX
> Draws a box around this gadget's select box.

GADGHIMAGE
> Displays an alternate Image or Border.

GADGHNONE
> Set this flag if you want no highlighting.

GADGIMAGE
> Use this bit if you have not set *GadgetRender* to NULL. Set this flag if the gadget should be rendered as an Image, clear the flag if it's a Border.
>
> This bit is also used by *SelectRender*.

GRELBOTTOM
Set this flag if the gadget's *TopEdge* variable describes an offset relative to the bottom of the display element containing it. Clear this flag if *TopEdge* is relative to the top.

GRELRIGHT
Set this flag if the gadget's *LeftEdge* variable describes an offset relative to the right edge of the display element containing it. Clear this flag if *LeftEdge* is relative to the left edge.

GRELWIDTH
Set this flag if the gadget's *Width* variable describes an increment to the width of the display element containing the gadget. Clear this flag if *Width* is an absolute value.

GRELHEIGHT
Set this flag if the gadget's *Height* variable describes an increment to the height of the display element containing the gadget. Clear this flag if *Height* is an absolute value.

SELECTED
Use this flag to pre-select the on/off selected state for a toggle-selected gadget. If the flag is set, the gadget starts off being on and it is highlighted. If the flag is clear, the gadget starts off in the unselected state.

GADGDISABLED
If this flag is set, this gadget is disabled. If you want to enable or disable a gadget later on, you can change the current state with the routines *OnGadget()* and *OffGadget()*.

You don't need to use this flag if you want the gadget to always remain enabled.

ACTIVATION FLAGS

Here are the flags you can set in the *Activation* variable of the Gadget structure:

TOGGLESELECT
When this bit is set, the on/off selected state of the gadget (and its imagery) toggles each time it is hit.

You preset the selection state with the gadget *Flag* SELECTED and later discover the selected state by examining SELECTED.

GADGIMMEDIATE
Set this bit if you want to know immediately when the user selects this gadget.

RELVERIFY
This is short for "Release Verify." Set this bit if you want this gadget selection broadcast to your program only if the user still has the pointer positioned over this gadget when releasing the select button.

ENDGADGET
> This flag pertains only to gadgets attached to requester. To make a requester go away, the user has to select a gadget that has this flag set.
>
> See Chapter 7, "Requesters and Alerts", for more information about requester gadget considerations.

FOLLOWMOUSE
> When the user selects a gadget that has this flag set, you will receive mouse position broadcasts every time the mouse moves at all.

You can use the following flags in window gadgets to adjust the size of a window's borders when you want to tuck your own window gadgets out of the way into the window border:

RIGHTBORDER
> If this flag is set, the width and position of this gadget are used in deriving the width of the window's right border.

LEFTBORDER
> If this flag is set, the width and position of this gadget are used in deriving the width of the window's left border.

TOPBORDER
> If this flag is set, the height and position of this gadget are used in deriving the height of the window's top border.

BOTTOMBORDER
> If this flag is set, the height and position of this gadget are used in deriving the height of the window's bottom border.

The following flags apply to string gadgets:

STRINGCENTER
> If this flag is set, the text in a string gadget is center-justified when rendered.

STRINGRIGHT
> If this flag is set, the text in a string gadget is right-justified when rendered.

LONGINT
> If this flag is set, the user can construct a 32-bit signed integer value in a normal string gadget. You must also pre-set the string gadget input buffer by putting an initial integer string in it.

ALTKEYMAP
> This flag specifies that you have an alternate key-map. You also need to put a pointer to the key-map in the StringInfo structure variable *AltKeyMap*.

SPECIALINFO DATA STRUCTURES

Following are the specifications for the structure pointed to by the *SpecialInfo* pointer in the Gadget structure.

PropInfo Structure

This is the special data required by the proportional gadget.

```
struct PropInfo
{
    USHORT Flags;
    USHORT HorizPot;
    USHORT VertPot;
    USHORT HorizBody;
    USHORT VertBody;
    USHORT CWidth;
    USHORT CHeight;
    USHORT HPotRes, VPotRes;
    USHORT LeftBorder;
    USHORT TopBorder;
};
```

The meanings of the fields in this structure are as follows:

Flags

General purpose flag bits:

AUTOKNOB
> Set this if you want to use the auto-knob.

FREEHORIZ
> If set, the knob can move horizontally.

FREEVERT
> If set, the knob can move vertically.

KNOBHIT
> This is set when this knob is hit by the user.

PROPBORDERLESS
> Set this if you want your proportional gadget to appear without a border drawn around its container.

Initialize these variables before the gadget is added to the system; then look here for the current settings:

HorizPot
> Horizontal quantity percentage.

VertPot
> Vertical quantity percentage.

These variables describe what percentage of the the entire body of the stuff is actually shown at one time:

HorizBody
> Horizontal body.

VertBody
> Vertical body.

Intuition sets and maintains the following variables:

CWidth
> Container real width.

CHeight
> Container real height.

HPotRes, VPotRes
> Pot increments.

LeftBorder
> Container real left border.

TopBorder
> Container real top border.

StringInfo Structure

This is the special data required by the string gadget.

```
struct StringInfo
{
        UBYTE *Buffer;
        UBYTE *UndoBuffer;
        SHORT BufferPos;
        SHORT MaxChars;
        SHORT DispPos;
        SHORT UndoPos;
        SHORT NumChars;
        SHORT DispCount;
        SHORT CLeft, CTop;
        struct Layer *LayerPtr;
        LONG LongInt;
        struct KeyMap *AltKeyMap;
};
```

The meanings of the fields in this structure are given in the following.

You initialize the following variables and Intuition maintains them:

Buffer
> Pointer to a buffer containing the start and final string. The string you write into this buffer should be null-terminated.

UndoBuffer
> Optional pointer to a buffer for undoing the current entry. If you are supplying an undo buffer, the memory location should be as large as the buffer for the start and final string. Because only one string gadget can be active at a time under Intuition, all of your string gadgets can share the same undo buffer. However, the undo buffer must be large enough to hold the largest buffer for start and final strings.

MaxChars
> must be number of characters in the buffer, including the terminating NULL.

BufferPos
> Initial character position of the cursor in the buffer.

DispPos
> Buffer position of the first displayed character.

Intuition initializes and maintains these variables for you:

UndoPos
> Character position in the undo buffer.

NumChars
> Number of characters currently in the buffer.

DispCount
> Number of whole characters visible in the container.

CLeft, CTop
> Top left offset of the container.

LayerPtr
> The Layer containing this gadget.

LongInt
> After the user has finished entering an integer, you can examine this variable to discover the value if this is an integer string gadget.

AltKeyMap
> This variable points to your own alternate keymap; you must also set the ALTKEYMAP bit in the *Activation* flags of the gadget:

GADGET FUNCTIONS

These are brief descriptions of the functions you can use to manipulate gadgets. For complete descriptions see Appendix A, "Intuition Function Calls".

Adding and Removing Gadgets from Windows or Screens

Use the following functions to add a gadget to or remove a gadget from the gadget list of a window or screen.

> *AddGadget(AddPtr, Gadget, Position)*
>> Adds a gadget to the gadget list of a window or screen.
>>
>> *AddPtr* is a pointer to the window or screen.
>> *Gadget* is a pointer to the Gadget.
>> *Position* is where the new gadget should go in the list.

> *RemoveGadget(RemPtr, Gadget)*
>> Removes a gadget from the gadget list of the specified window or screen.
>>
>> *RemPtr* is a pointer to the window or screen from which gadget is to be removed.
>> *Gadget* is a pointer to the gadget to be removed;

Disabling or Enabling a Gadget

The following functions disable or enable a gadget in a window, screen, or requester.

OnGadget(Gadget,Ptr,Requester)
Enables the specified gadget.

> *Gadget* points to the Gadget you want enabled.
> *Ptr* points to a screen or window.
> *Requester* points to a requester, or is NULL.

OffGadget(Gadget, Ptr, Requester)
Disables the specified Gadget.

> *Gadget* points to the gadget to be disabled.
> *Ptr* points to a screen or window structure.
> *Requester* points to a requester or is NULL.

Redraw the Gadget Display

This function redraws all of the gadgets in the gadget list of a screen, window, or requester, starting with the specified gadget. You might want to use this if you have modified the imagery of your gadgets and want to display the new imagery. You might also use it if you think some graphic operation has trashed the imagery of the gadgets.

RefreshGadgets(Gadgets, Ptr, Requester)
> *Gadgets* points to the gadget where the redrawing should start.
> *Ptr* points to the window or screen.
> *Requester* points to a requester or is NULL.

Modifying a Proportional Gadget

Use this function to modify the current parameters of a proportional gadget.

ModifyProp(Gadget, Ptr, Requester, Flags, HorizPot, VertPot, HorizBody, VertBody)
> Modifies the parameters of a proportional gadget. The gadget's internal state is recalculated and the imagery is redisplayed.

Chapter 6

MENUS

This chapter shows how to set up the menus that let the user choose from your program's commands and options. The Intuition menu system handles all of the menu display from menu data structures that you set up. If you wish, some or all of your menu selections can be graphic images instead of text.

The first section of this chapter describes menus, menu items, and sub-items; how they are displayed; and how your program finds out about the user's menu selections. The second section gives complete specifications for all the menu structures and menu-related functions.

About Menus

Intuition's menu system provides you with a convenient way to group together and display the functions and options that your application presents to the user. For instance, in a word-processor environment, menus may provide the following functions:

- o access to text files
- o edit functions
- o search and replace
- o formatting
- o multiple fonts
- o a general help facility

Or in a game, menus may provide the user with choices about how to:

- o load a new game or save the current one
- o get hints
- o bring up special information windows
- o set the difficulty level
- o auto-annihilate the enemy

Menu commands are either actions or attributes. Actions are represented by verbs and attributes by adjectives. An attribute stays in effect until canceled, while a command is executed and then forgotten. You can set up menus so that some attribute items are mutually exclusive (selecting an attribute cancels the effects of one or more other attributes), or you can allow a

number of attributes to be in effect at the same time. For example, an adventure game might have a menu list for things that the hero is holding in his hand. He could hold several small, lightweight objects, but holding the heavy sword excludes holding anything else. In a database program, you might be able to choose to send a report to a file, to the window, or to a printer. You could, for example, send it to both window and printer, while the "file" option excludes the other two.

After you set up a linked list of menu structures (called a *menu strip*) and attach the list to a window, the menu system handles the menu display. Using this list and any graphic images you have designed, the menu system displays the menu bar text that appears across the screen title bar when requested by the user. It also creates the lists of menu items and sub-menus that appear at the user's request. The application doesn't have to worry about menus until Intuition sends a message with news that the user has selected a menu item. This message gives the application the number of the selected item.

You can enable and disable menus and menu items during the display of the window and make changes to the menus you previously attached to a window. Disabling an item prevents the user from selecting it, and disabled items are ghosted to look different from enabled items.

Menu items can be graphic images or text. When the user positions the pointer over an item, the item can be highlighted through a variety of techniques and have a check mark placed next to it. Next to the menu items, you can display command-key alternatives.

To activate the menu system, the user presses the mouse menu button (or an appropriate command-key sequence) to display the menu bar in the screen title area. The menu bar displays a list of topics (called *menus*) that have menu items associated with them.
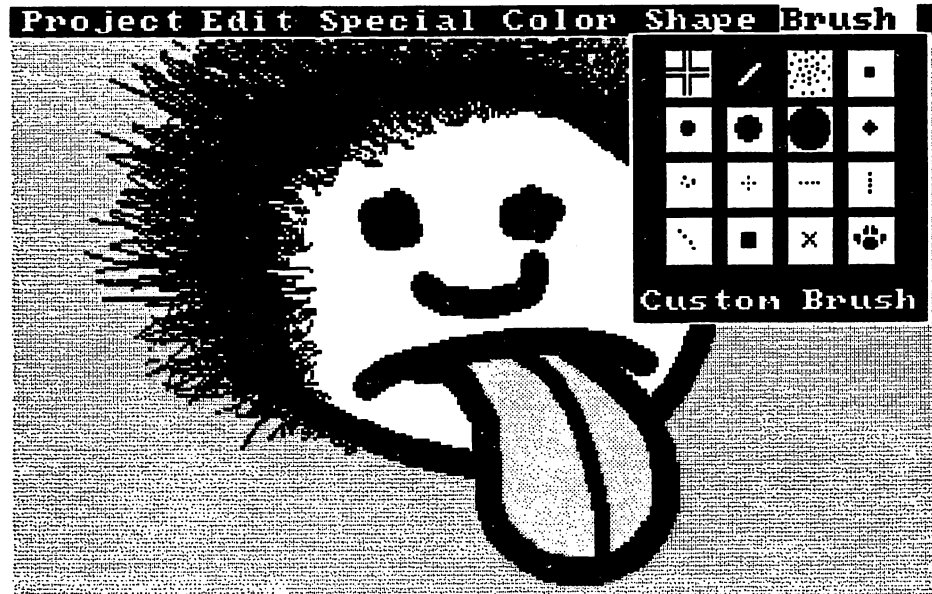
Figure 6-1: Screen with Menu Bar Displayed

When the user moves the mouse pointer to a topic in the menu bar, a list of menu items appears below the topic name. To select an item, the user moves the mouse pointer in the list of menu items while holding down the menu button, releasing the button when the pointer is over the desired item. If an item has a sub-item list, moving the pointer over the item reveals a list of sub-items. The user moves the pointer over one of the sub-items and makes a selection in

the same way as an item is selected. If there is a command-key sequence alternative, the user can make menu selections with the keyboard instead of the mouse. Furthermore, the user can select multiple items by:

o   pressing and releasing the mouse select button without releasing the menu button. This selects that item and keeps the user in "menu state" so that other items can be selected.

o   holding down both mouse buttons and moving the pointer over several items. This is called "drag-selecting".

SUBMITTING AND REMOVING MENU STRIPS

Once you have constructed a menu strip, you submit it to Intuition using the function *Set-MenuStrip()*. You must always remove every menu strip that you've submitted. When you want to remove the menu strip, you call *ClearMenuStrip()*. If you want to change the menu strip, you call *ClearMenuStrip()*, change the menu, and resubmit it with *SetMenuStrip()*.

The flow of events for menu operations should be:

o   *OpenWindow()*

o   Zero or more iterations of

    -    *SetMenuStrip()*
    -    *ClearMenuStrip()*

o   *CloseWindow()*

Clearing the menu strip before closing the window avoids any of the problems that can occur if the user is accessing menus when you close the window.

ABOUT MENU ITEM BOXES

The item box is the rectangle containing your menu items or sub-items. You do not have to describe the size and location of the item or sub-item boxes directly. You describe the size indirectly by how you place items and sub-items. Intuition figures out the size of the minimum box required. It then adjusts the size of the box to make sure your menu display conforms to certain design philosophy constraints for items and sub-items.

Items
o   The item box must start no further right than the leftmost position of the menu header's select box.

o   The item box must end no further left than the rightmost position of the menu header's select box.

o   The top edge of the menu box must overlap the screen's title bar by one line.

Sub-items

o   The sub-item box must overlap its item's select box somewhere.
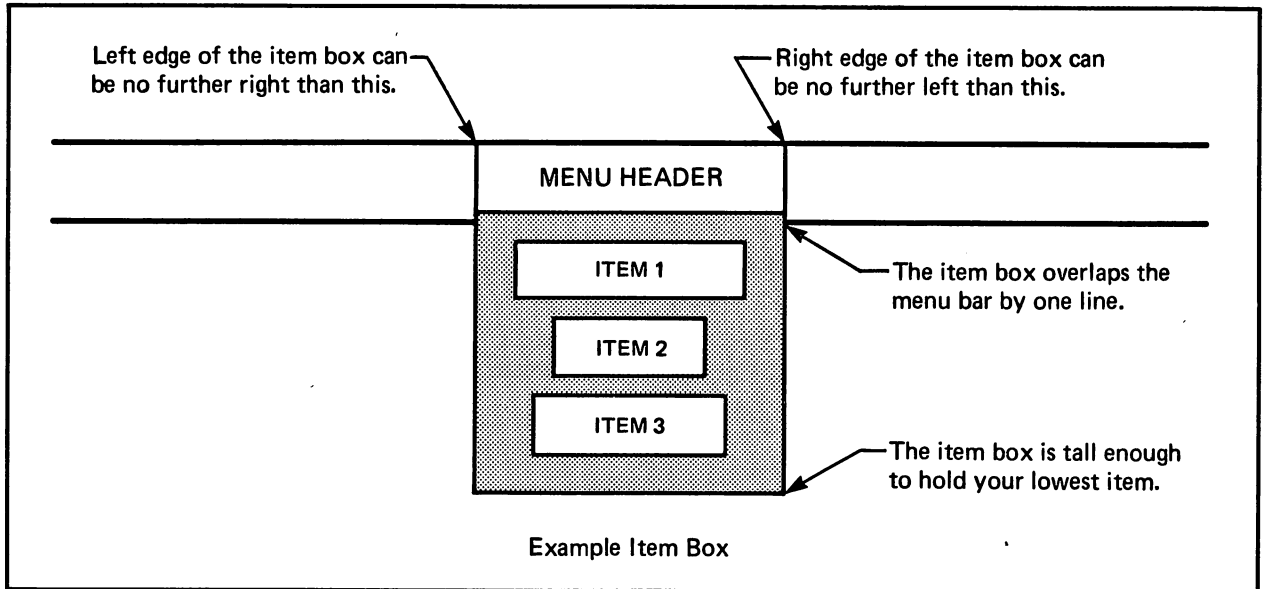
Figure 6-2: Example Item Box

```
                 ┌──────────────────┐
                 │   MENU HEADER    │
                 ├──────────────────┤
                 │░░░░░░░░░░░░░░░░░░░│
                 │░┌──────────────┐░│
                 │░│    ITEM 1    │░│        ┌─The subitem box must overlap
                 │░└──────────────┘░│        │ the item's select box somewhere.
                 │░┌──────────────┐░│        │ (It does not matter where.)
                 │░│    ITEM 2    │░│
                 │░└──────────────┘░│
                 │░┌──────────────┐▓▓▓▓▓▓▓▓▓▓▓▓
                 │░│    ITEM 3    │▓┌──────────────┐▓
                 │░└──────────────┘▓│  SUBITEM 1   │▓
                 │░░░░░░░░░░░░░░░░░░▓└──────────────┘▓
                              ▓┌──────────────┐▓
                              ▓│  SUBITEM 2   │▓
                              ▓└──────────────┘▓
                              ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓

                      Example Subitem Box
```
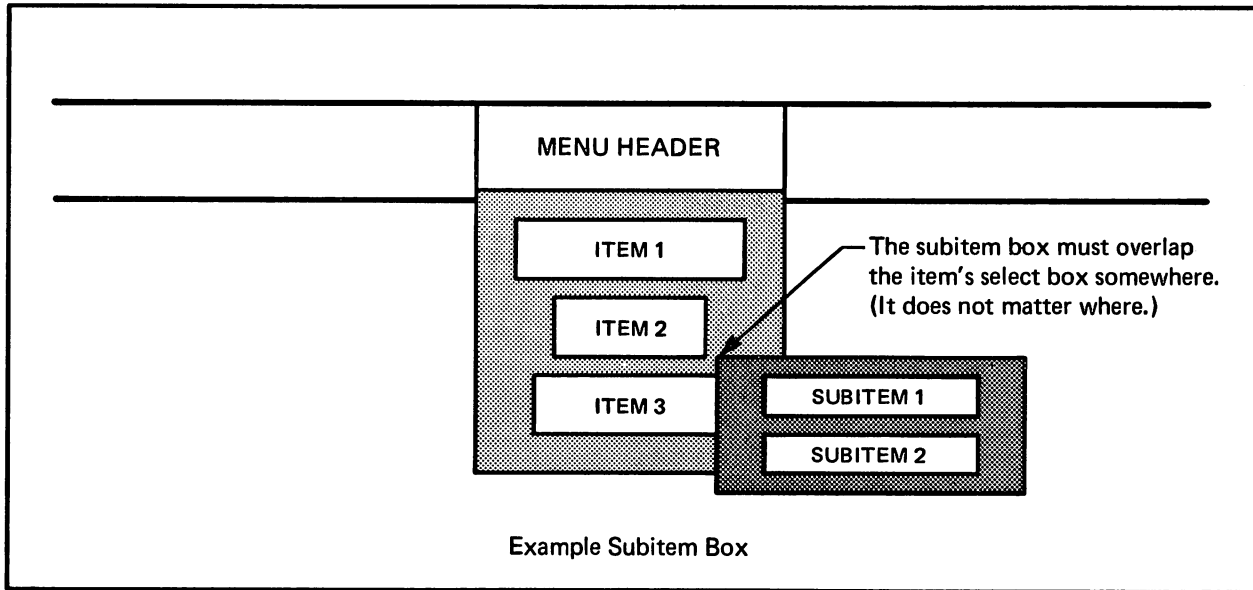
Figure 6-3: Example Sub-Item Box

ACTION/ATTRIBUTE ITEMS AND THE CHECKMARK

Menu action items are selected and acted upon immediately. Action items can be selected repeatedly. Every time the user selects an action item, the selection is transmitted to your program.

Menu attribute items, on the other hand, are selected and remain selected until the attribute is mutually excluded by the selection of some other attribute item. Menu attribute items, when selected, appear with a checkmark drawn along the left edge of the item's select box. A selected attribute item cannot be reselected until mutual exclusion causes it to become unselected. See the "Mutual Exclusion" section below for a description of how this works.

You specify that a particular menu item is an attribute item by setting the CHECKIT flag in the *Flags* variable of the item's MenuItem structure. If you set this flag, then this item will have a checkmark drawn next to it whenever it is selected..

You can initialize the state of an attribute item by presetting the item's CHECKED flag. If this flag is set when you submit your menu strip to Intuition, then the item is considered to be already selected and the checkmark will be drawn.

You can use the default Intuition checkmark (√), or you can design your own and set a pointer to it in the NewWindows structure when you open a window. See Chapter 4, "Windows", for details about supplying your own checkmark.

If your items are going to be checkmarked, you should leave sufficient blank space at the left edge of your select box for the checkmark imagery. If you are taking advantage of the default

checkmarks, you should leave CHECKWIDTH (pixel width) amount of blank space on high-resolution screens, and LOWCHECKWIDTH amount of blank pixels on low-resolution screens. These are defined constants describing the pixel-width in high- and low-resolution. They define the space required by the standard checkmarks (with a bit of space for aesthetic purposes). If you would normally place the LeftEdge of the image within the item's select box at 5, and you decide that you want a checkmark to appear with the item, then you should start the item at 5+CHECKWIDTH instead. You should also make your select box CHECKWIDTH wider than it would be without the checkmark.

MUTUAL EXCLUSION

You can choose to have some of your attribute items, when selected, to cause other items to become unselected. This is known as *mutual exclusion*. For example, if you have a list of menu items describing the available type sizes for a particular font, the selection of any type size would mutually exclude all other type sizes. You use the *MutualExclude* variable in the MenuItem structure to specify other menu items to be excluded when the user selects an item. Exclusion also depends upon the CHECKED and CHECKIT flags of the MenuItem as explained below.

- o If CHECKIT is set, then this item is an attribute item that can be selected and unselected. If CHECKED is *not* set, then this item is available to be selected. If the user selects this item, then the CHECKED flag is set and the user cannot then re-select this item. If the item is selected, the CHECKED flag will be set, and the checkmark will be drawn to the left of the item.

- o If CHECKIT is not set, then this is an action item—not an attribute item. The CHECKED flag is ignored and the checkmark will never be drawn. Mutual exclusion affects only attribute items.

- o If an item is selected that has bits set in the *MutualExclude* field, the CHECKIT and CHECKED flags are examined in the excluded items. If any item is currently CHECKED, its checkmark is erased.

- o Mutual exclusion is an active event. It pertains only to items that have the CHECKIT flag set. Attempting to exclude items that don't have the CHECKIT flag set has no effect.

It's up to you to note internally as needed that excluded items have been disabled and deselected.

In the *MutualExclude* field, bit 0 refers to the first item in the item list, bit 1 to the second, bit 2 to the third, and so on. In the adventure game example described earlier where carrying the heavy sword excludes carrying any other items, the *MutualExclude* fields of the four items would look like this:

Heavy sword	0xFFFE
Stiletto	0x0001
Rope	0x0001
Canteen	0x0001

"Heavy Sword" is the first item on the list. You can see that it excludes all items except the first one. All of the other items exclude only the first item, so that carrying the rope excludes

carrying the sword, but not the canteen.

COMMAND-KEY SEQUENCES AND RENDERING

A *command-key sequence* is an event generated by the user, where the user holds down one of the AMIGA keys (the ones with the fancy A) and presses one of the normal alphanumeric keys at the same time. You can associate a command-key sequence with a particular menu item. Menu command-key sequences are combinations of the right AMIGA key with any alphanumeric character. If the user presses a command-key sequence that's associated with one of your menu items, then Intuition will send you an event that will look like the user went through the entire process of selecting the menu item manually. This allows you to provide *shortcuts* to the user, since many people find it easy to memorize the command-key sequences for often-repeated menu selections. When accessing those often-repeated selections, most users would rather keep their hands on the keyboard than go to the mouse to make a menu selection.

You associate a command-key sequence with a menu item by:

o   setting the COMMSEQ flag in the *Flags* variable of the MenuItem structure, and

o   putting the ASCII character (upper or lower case) that you want associated with the sequence into the *Command* variable of the MenuItem structure.

When items have alternate key sequences, the menu boxes show:

o   a special AMIGA key icon rendered about one character span plus a few pixels from the right edge of the menu select box, and

o   the command-key used with the AMIGA key rendered immediately to the right of the AMIGA key image, at the rightmost edge of the menu select box.

If you want to show a command-key sequence for an item, you should make sure that you leave blank space at the right edge of your select box and imagery. You should leave COMMWIDTH amount of blank space on high-resolution screens, and LOWCOMMWIDTH amount of space on low-resolution screens.
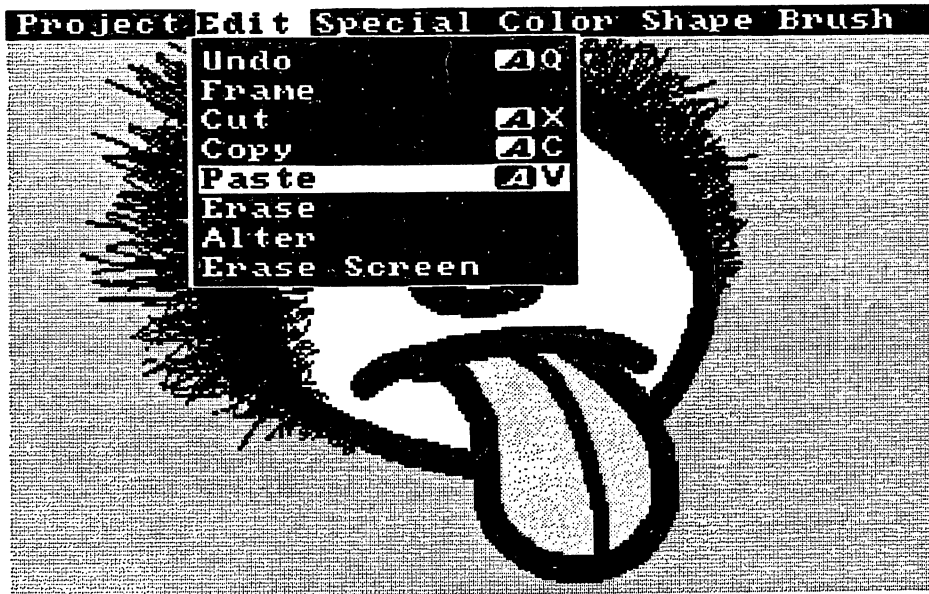
Figure 6-4: Menu Items with Command Key Shortcuts

Please be sure to see Chapter 12, "Style Notes", for suggested command key sequences.

ENABLING AND DISABLING MENUS AND MENU ITEMS

Disabling menu items makes them unavailable for selection by the user. Disabled menus and menu items are displayed in a "ghosted" fashion; that is, the imagery is overlaid with a faint pattern of dots, making it less distinct. Enabling or disabling a menu or menu item is always a safe procedure, whether or not the user is currently using the menus. A problem arises only if you disable a menu item that the user has already selected with extended select. You will receive a MENULIST message for that item, even though you think you have already disabled it. You will have to ignore items that you already know are disabled.

You use the routines *OnMenu()* and *OffMenu()* to enable and disable individual sub-items, items or whole menus. These routines check if the user is using the menus and whether the menus need to be redrawn to reflect your new states.

CHANGING MENU STRIPS

If you want to make changes to the menu strip you previously attached to your window, you must first call *ClearMenuStrip()*. You may alter the menu strip only after it has been removed from the window.

To add a new menu strip to your window, you must call *ClearMenuStrip()* before you call *Set-*

MENU NUMBERS AND MENU SELECTION MESSAGES

An input event is generated every time the user activates the menu system by pressing the mouse menu button (or entering an appropriate command-key sequence). Your program receives a message of type MENULIST telling which menu item has been selected. If one of your items has a sub-item list, the menu number you receive for that item includes some sub-item selection.

Even if the user presses and releases the menu button without selecting any of the menu items, an event is generated. If the user presses and releases the menu button without selecting one of the menu items, you receive a message with the menu number equal to MENUNULL. In this way, you can always find out when the user has simply clicked the menu button rather than making a menu selection.

The user can select multiple menu items with one of the extended selection procedures (pressing the mouse select button without releasing the menu button, or drag-selecting). Your program finds out whether or not multiple items have been chosen by examining the field called *NextSelect* in the menu item data structure. After you take the appropriate action for the item selected by the user, you should check the *NextSelect* field. If the number there is equal to the constant MENUNULL, there is no next selection. However, if it's not equal to MENUNULL, the user has selected another option after this one. You should process the next item as well, by checking its *NextSelect* field, until you find a *NextSelect* equal to MENUNULL.

The following code fragment shows the correct way to process a menu event:

```
while (MenuNumber != MENUNULL)
    {
    Item = ItemAddress(MenuStrip, MenuNumber);
    /* process this item */
    MenuNumber = Item->NextSelect;
    }
```

The number given in the MENULIST message describes the ordinal position of the Menu in your linked list, the ordinal position of the MenuItem beneath that Menu, and (if applicable) the ordinal position of the sub-item beneath that MenuItem. Ordinal means the successive number of the linked items, starting from 0. To discover the Menus and MenuItems that were selected, you should use the following macros:

> Use MENUNUM(num) to extract the ordinal menu number from the value.
> Use ITEMNUM(num) to extract the ordinal item number from the value.
> Use SUBNUM(num) to extract the ordinal sub-item number from the value.
> MENUNULL is the constant describing "no menu selection made".
> Likewise, NOMENU, NOITEM, and NOSUB are the null states of the parts.

For example:

```
if (number === MENUNULL) then no menu selection was made, else
MenuNumber = MENUNUM(number);
ItemNumber = ITEMNUM(number);
SubNumber = SUBNUM(number);
if there were no sub-items attached to that item, SubNumber will equal NOSUB.
```

When you get a menu number, it describes either MENUNULL or a valid menu selection. If it's a valid selection, it will always have at least a menu number and a menu item number. Users can never "select" the menu text itself, but they always select at least an item within a menu. Therefore, you always get one menu specifier and one menu item specifier. If a given menu item has a sub-item, you will receive a sub-item specifier as well. Just as it's not possible to select a menu, it's not possible to select a menu item that has a list of sub-items. The user must select one of the options in the sub-item before you ever hear about it as a valid selection.

If the user enters a command-key sequence, Intuition checks to see if the sequence is associated with a current menu item. If so, Intuition sends the menu item number to the program with the active window just as if the user had made the selection using the mouse buttons.

The function *ItemAddress()* translates a menu number into an item address.

HOW MENU NUMBERS REALLY WORK

Here is a description of how menu numbers really work. This should illuminate why there are certain numeric restrictions on the number of menu components Intuition allows. You should not use the information given here to access the menu number information directly. This discussion is included only for completeness. To assure upward compatibility, always use the macros supplied. For example, call ITEMNUM(MenuNumber) to extract the item number from the variable *MenuNumber*. See the previous section, "Menu Numbers and Menu Selection Messages", for a complete description of the menu number macros.

MENU NUMBERS are 16-bit numbers with 5 bits used for the menu number, 6 bits used for the menu item number, and 5 bits used for the sub-item number. Everything is specified by its ordinal position in a list of same-level pieces, as shown below.

```
c   c   c   c   c   b   b   b   b   b   b   a   a   a   a   a
|                   |                     |
|                   |                     > These bits are for the menu number.
|                   |
|                   > These bits are for the menu items within the menu.
|
> These bits are for the sub-menu items within the menu items.
```
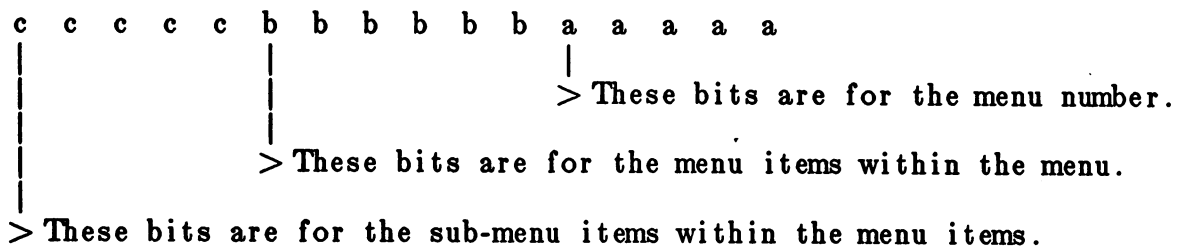
This means that for each level of menu item and sub-item, up to 31 pieces can be specified. And there are 63 item pieces that you can build under each menu. 63 items per menu is a lot, especially with 31 sub-items per item. You can have 31 menu choices across the menu bar (it would be a tight squeeze, but in 80-column mode you could do it), and each of those menus can exercise up to 1953 items. You shouldn't need any more choices than that.

The value "all bits on" means that no selection of this particular component was made. MENUNULL actually equals "no selection of any of the components was made" so MENUNULL always equals "all bits of all components on".

Here's an example. Say that your program gets back the menu number (in hexadecimal) 0x0CA0. In binary that equals:

```
0   0   0   0   1   1   0   0   1   0   1   0   0   0   0   0
|               |                       |
|               |                       > Menu number 0
|               |
|               > Menu item number 0x25 = 37
|
> Sub-item number 1
```

Again, it is never safe to examine these numbers directly. Use the macros described above if you want to design sanely and assure upward-compatibility.

INTERCEPTING NORMAL MENU OPERATIONS

You have two convenient ways to intercept the normal menu operations that take place when the user presses the right mouse button. The first, MENUVERIFY, gives you the opportunity to react before menu operations take place, and optionally to cancel menu operations. The second, RMBTRAP allows you to trap right mouse button events for your own use.

MenuVerify

MenuVerify is one of the Intuition verification functions. These functions allow you to make sure that you're prepared for some event before it takes place. Using MenuVerify, Intuition allows all windows in a screen to verify that they are prepared for menu operations before the operations begin. In general, you would want to use this if you are doing something special to the display of a custom screen, and you want to make sure it's completed before menus are rendered.

Any window can access the MenuVerify feature by setting the MENUVERIFY flag in the NewWindow structure when opening the window. When you get a message of class MENU-VERIFY, menu operations will not proceed until you reply to the message.

The active window gets special MenuVerify treatment. It's allowed to see the MenuVerify message before any other window and has the option of cancelling menu operations altogether. You could use this, for instance, to examine where the user has positioned the mouse when the right button was pressed. If the pointer is in the menu bar area, then you can let normal menu operations proceed. If the pointer is below the menu bar, then you can use the right button event for some non-menu purpose.

You can tell whether or not you are the active window by examining the code field of the MENUVERIFY message. If the code field is equal to MENUWAITING, you are not the active window and Intuition is simply waiting for you to verify that menu operations may continue.

However, if the code field is equal to MENUHOT, you <u>are</u> the active window and you get to decide whether or not menu operations should proceed. If you do <u>not</u> want them to proceed, you should change the code field of the message to MENUCANCEL before replying to the message. This will cause Intuition to cancel the menu operations.

No Menu Operations — Right Mouse Button Trap

By setting the RMBTRAP flag in the NewWindow structure when you open your window, you select that you don't want any menu operations at all for your window. Whenever the user presses the right button while your window is the active one, you will receive right button events as normal MOUSEBUTTON events.

REQUESTERS AS MENUS

You may, in some cases, want to use a requester instead of a menu. A requester can function as a "super-menu" because you can attach a requester to the double-click of the mouse menu button. This allows users to bring up the requester on demand. With a requester, however, the user must make some response before resuming input to the window. See Chapter 7, "Requesters and Alerts", for more information.

Using Menus

Follow these steps to design and use menus:

o   Design the menu structures and link them together into a menu strip.
o   Submit the menu strip to Intuition, which attaches the strip to a window
o   Arrange for your program to respond to Intuition's menu selection messages.

To create the menu structures, you need to decide on:

o   The menu names that appear in the screen title bar
o   The menu items that appear when the user selects a menu, including:

   o   Each menu item's position in the list
   o   Text or a graphic image for each menu item
   o   Highlighting method for this item when the user positions the pointer over it
   o   Any equivalent command-key sequence
   o   Menu items that have have sub-items. For the sub-menu items, you make the same decisions as for the menu items, except for this one.

Menu strips are constructed of three components: menus, menu items, and sub-items. They use two data types: Menu and MenuItem. Sub-items are of the MenuItem data type.

Menu is the data type that describes the basic unit of the menu strip. The menu strip is made up of a linked list of Menus. Each menu is the header or topic name for a list of MenuItems that can be selected by the user. The user never selects just a Menu, but rather a Menu *and* at least one of its MenuItems.

The Menu structure contains the following:

- o Menu bar text that appears across the screen's title bar when the menu button is pressed
- o The position for the menu bar text
- o A pointer to the next in the list of Menus
- o A pointer to the first in a linked list of MenuItems

The MenuItem structure contains the following:

- o The location of the item (with respect to the select box of its Menu)
- o A pointer to text or a graphics image
- o Highlighting method when the user positions the pointer over this item
- o Any equivalent command sequence
- o The "select box" for the item (used to detect selection and for some of the highlighting modes)
- o Other items mutually excluded by the selection of this one (if any)
- o A pointer to the first in a linked list of sub-items (if any)
- o The menu number of the next selected item (if any). When more than one item has been selected, this field provides the link.

The third menu component, the sub-item, uses the same data structure as the menu item. Sub-items are identical in most respects to menu items. The differences are:

- o The sub-item's location is relative to its menu item's select box.
- o The sub-item's sub-item link is ignored.

MENU STRUCTURES

This sub-section contains the specifications for the menu structures:

- o Menus - the headers that show in the menu bar
- o MenuItems - the items and sub-items that can be chosen by the user

Menu Structure

Here is the specification for a Menu structure:

```
struct Menu
{
        struct Menu *NextMenu;
        SHORT LeftEdge, TopEdge, Width, Height;
        USHORT Flags;
        BYTE *MenuName;
        struct MenuItem *FirstItem;
};
```

The variables in the Menu structure have the following meanings:

NextMenu
>   This variable points to the next Menu header in the list. The last Menu in the list should
>   have a NextMenu value of NULL.

LeftEdge, TopEdge, Width, Height
>   These fields describe the select box of the header. Currently, any values you may supply
>   for TopEdge and Height are ignored by Intuition, which uses instead the screen's Top-
>   Border for the TopEdge and the height of the screen's Title Bar for the Height. This will
>   change someday when menu headers are allowed to be either textual or graphical, and are
>   allowed to appear anywhere in the menu title bar. LeftEdge is relative to the LeftEdge of
>   the screen plus the screen's left border width, so if you say LeftEdge is 0, Intuition puts this
>   header at the leftmost allowable position.

Flags
>   The flag space is shared by your program and Intuition. The flags are:

>   MENUENABLED
>>   Whether or not this Menu is currently enabled. You set this flag before you submit the
>>   menu strip to Intuition. If this flag is not set, the menu header and all menu items
>>   below it will be disabled, and the user will not be able to select any of the items. After
>>   you submit the strip to Intuition, you can change whether your menu is enabled or dis-
>>   abled by calling *OnMenu()* or *OffMenu()*.

>   MIDRAWN
>>   Whether or not this Menu's items are currently displayed to the user.

MenuName
>   This is a pointer to a null-terminated character string which is printed on the screen title
>   Bar starting at the LeftEdge of this Menu's select box, and TopEdge just below the screen
>   title bar's top border

FirstItem
>   This points to the first item in the linked list of this Menu's items (MenuItem structures).

MenuItem Structure

Here is the specification for a MenuItem structure (used both for items and sub-items):

```
struct MenuItem
{
        struct MenuItem *NextItem;
        SHORT LeftEdge, TopEdge, Width, Height;
        USHORT Flags;
        LONG MutualExclude
        APTR ItemFill;
        BYTE Command;
        struct MenuItem *SubItem;
        USHORT NextSelect;
};
```

The fields have the following meanings:

NextItem
>   Pointer to the next item in the list. The last item in the list should have a NextItem value of NULL.

LeftEdge, TopEdge, Width, Height
>   These fields describe the select box of the MenuItem. The LeftEdge is relative to the LeftEdge of the Menu. The TopEdge is relative to topmost position Intuition allows. TopEdge is based on the way the user has the system configured, — which font, which resolution, and so on. Use 0 for the topmost position.

Flags
>   The flag space is shared by your program and Intuition. See "MenuItem Flags" below for a description of the flag bits.

MutualExclude
>   This LONG word refers to the items that may be on the same "plane" as this one (maximum of 32 items). You use these bits to describe which if any of the other items are mutually excluded by this one. This doesn't mean that you can't have more than 32 items in any given plane, just that only the first 32 can be mutually excluded.

ItemFill
>   This points to the data used in rendering this MenuItem. It can point to either an instance of an IntuiText structure with text for this MenuItem, or it can point to an instance of an Image structure with image data. You tell Intuition what sort of data is pointed to by this variable by either setting or clearing the MenuItem flag bit ITEMTEXT. See "MenuItem

Flags" below for more information about ITEMTEXT.

SelectFill
    If you select the MenuItem highlighting mode HIGHIMAGE (in the *Flags* variable), Intuition substitutes this alternate image for the original rendering described by *ItemFill*. *SelectFill* can point to either an Image or an IntuiText, and the flag ITEMTEXT describes which.

Command
    This variable is storage for a single alphanumeric character. If the *Flag* COMMSEQ is set, the user can hold down the right AMIGA key on the keyboard (to mimic using the right mouse menu button) and press the key for this character as a shortcut for using the mouse to select this item. If the user does this, Intuition transmits the menu number for this item to your program. It will look to your program exactly as if the user had selected a menu item using menus and the pointer.

SubItem
    If this item has a sub-item list, this variable should point to the first sub-item in the list. Note that if this item is a sub-item, this variable is ignored.

NextSelect
    This field is filled in by Intuition when this item is selected by the user. If this item is selected by the user, your program should process the request and then check the *NextSelect* field. If the *NextSelect* field is equal to MENUNULL, then no other items were selected, otherwise there's another item to process. See "Menu Numbers and Menu Selection Messages" above for more information about user selections.

MenuItem Flags

Here are the flags that you can set in the *Flags* field of the MenuItem structure:

CHECKIT
    You set this flag to inform Intuition that this item is an attribute item and you want item rendered with a preceding check mark if the flag CHECKED is set. See the section "Action/Attribute Items and the CheckMark" above for full details.

CHECKED
    Set the CHECKIT flag above if you want this item to be checked when the user selects it. When you first submit the menu strip to Intuition, set this bit to specify whether or not this item is currently a selected one. Thereafter, Intuition maintains this bit based on effects from the item list's mutual exclusions.

ITEMTEXT
    You set this flag if the representation of this item (pointed to by the *ItemFill* field and possibly by *SelectFill*) is text and points to an IntuiText, or clear it if the item is graphic and points to an Image.

COMMSEQ

> If this flag is set, this item has an equivalent command-key sequence (see the *Command* field above).

ITEMENABLED

> This flag describes whether or not this item is currently enabled. If an item is not enabled, its image will be ghosted and the user will not be able to select it. Set this flag before you submit the menu strip to Intuition. Once you have submitted your menu strip to Intuition, you enable or disable items only by using *OnMenu()* or *OffMenu()*. If this item has sub-items, all of the sub-items are disabled when you disable this item.

HIGHFLAGS

> An item can be highlighted when the user positions the pointer over the item. These bits describe what type of highlighting you want, if any.

> You must set one of the following bits according to the type of highlighting you want:

HIGHCOMP

> Complements all of the bits contained by this item's select box.

HIGHBOX

> Draws a box outside this item's select box.

HIGHIMAGE

> Displays the alternate imagery in *SelectFill* (textual or image).

HIGHNONE

> No highlighting.

The following two flags are used by Intuition:

ISDRAWN

> Intuition sets this flag when this item's sub-items are currently displayed to the user and clears it when they are not.

HIGHITEM

> Intuition sets this flag when this item is highlighted, and clears it when the item is not highlighted.

MENU FUNCTIONS

There are menu functions for attaching and clearing menu strips, for enabling and disabling

menus or menu items, and for finding a menu number.

Attaching and Removing a Menu Strip

The following functions attempt to attach a menu strip to a window or clear a menu strip from a window:

SetMenuStrip(Window,Menu)

Menu is a pointer to the first menu in the menu strip. This procedure sets the menu strip into the window.

ClearMenuStrip(Window)
This procedure clears any menu strip from the window.

Enabling and Disabling Menus and Items

You can use the following functions to enable and disable items after a menu strip has been attached to the window. If the item component referenced by *MenuNumber* equals NOITEM, the entire menu will be disabled or enabled. If the item component equates to an actual component number, then that item will be disabled or enabled.

You can enable or disable whole menus, just the menu items, or just single sub-items.

o   To enable or disable a *whole menu,* set the item component of the menu number to NOITEM. This will disable all items and any sub-items.

o   To enable or disable a *single item* and all sub-items attached to that item, set the item component of the menu number to your item's ordinal number. If your item has a sub-item list, set the sub-item component of the menu number to NOSUB. If your item has no sub-item list, the sub-item component of the menu number is ignored.

o   To enable or disable a *single sub-item,* set the item and sub-item components appropriately.

OnMenu(Window, MenuNumber)
Enables the given menu or menu item.

OffMenu(Window, MenuNumber)
Disables the given menu or menu item.

Getting an Item Address

This function finds the address of a menu item when given the item number:

ItemAddress(MenuStrip, MenuNumber)
> MenuStrip is a pointer to the first menu in the menu strip.

<center>Chapter 7</center>

<center># REQUESTERS AND ALERTS</center>

This chapter describes requesters and alerts. Requesters are menu-like information exchange boxes that can be displayed in windows by the system or by application programs. You can also have requesters that the user can bring up on demand. They're called requesters because the user has to "satisfy the request" before continuing input through the window. Alerts are similar to requesters, but are reserved for emergency messages.

The first section of the chapter gives you general information about the features of application requesters—how to design your own completely custom requesters or let Intuition render them for you, how to place requesters in constant locations or display them relative to the current pointer position, and how to use the special IDCMP requester features. The second section shows you the mechanics of setting up requesters, invoking them, and removing them from the display. The last section covers alerts.

About Requesters

Requesters are like menus since both menus and requesters offer options to the user. Requesters, however, go beyond menus. They become "super menus" because you can place them anywhere in the window, design them to look however you want, and bring them up in the window whenever your program needs to elicit a response from the user—and they come replete with any kind of gadgets you care to use. The most fundamental differences between requesters and menus is that requesters require a response from the user; and while the requester is in the window, the window locks out all user input. The requirement of a user response is just about the only restriction placed on your program's use of requesters.
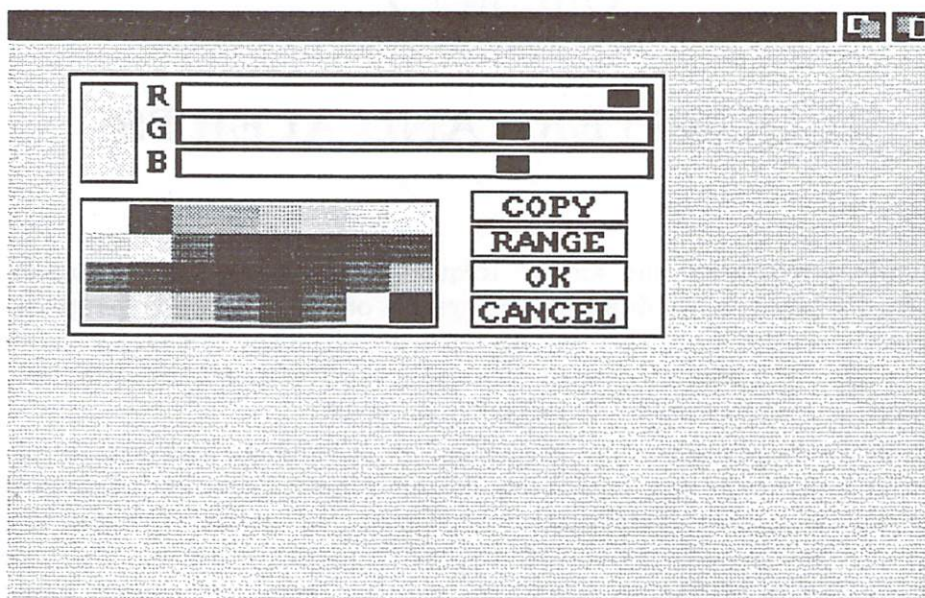
Figure 7-1: Requester Deluxe

Requester Display

Requesters can be brought up in a window in three different ways.

o   System requesters are invoked by the operating system; your program has no control over these. For example, someone using a text editor might try to save a file to disk when there's no disk in the drive. The system requester comes up and makes sure the user understands the situation by demanding a response from the user.

o   You can bring up the regular application requesters whenever your program needs input from the user.

o   You can attach a requester to a double-click of the mouse menu button. Users can bring up this "double-menu request" whenever they need the particular option supplied by the requester.

Once a requester is brought up in an window, all further input to the program from that window is blocked. This is true even if the user has brought up the requester. The requester remains in the window and input remains blocked until the user satisfies the request by choosing one of the requester gadgets. You decide which of your gadgets meets this criterion. While the requester is in the window, the only input the program receives from that window is broadcasts when the user selects a requester gadget. Even though the window containing the requester is locked for input, the user can work in another application or even in a different window of your application and respond to the requester later.

A window with an unsatisfied requester is *not* blocked for program output. Nothing prevents your program from writing to the window. You must, however, use caution since the requester

resides in the display memory of the window. If you should over-write the requester or cause the text window to scroll, you may render the window or even the entire display unusable. Fortunately, there are several ways to monitor the comings and goings of requesters, which you can use to ensure that you can safely bring up an application requester. (See "IDCMP Features" below.)

In displaying any kind of requester (except the super-simple yes or no kind created with *AutoRequest()*), you can specify the location in two ways. You can select either a constant location that is an offset from the top left corner of the window or a location relative to the current location of the pointer. Displaying the requester relative to the pointer can get the user's attention immediately and closely associates the requester with whatever the user was doing just before the requester came up in the window.

You can nest several application requesters in the same window, and the system may present requesters of its own that become nested with the application requesters. These are all satisfied in reverse sequence; the last requester to be displayed must be satisfied first.

Application Requesters

In adding requesters to your program, you have several options. You can supply a minimum of information and let Intuition do the work of rendering the requester, or you can design a completely custom requester, drawing the background, borders, and gadgets yourself and submit the requester to Intuition for display.

For a requester rendered by Intuition you have two choices. If the requester is complex and you want to attach gadgets and have some custom features, you initialize a requester for general usage. In the requester structure, you supply the gadget list, borders, text, and size of the rectangle that encloses the requester. Intuition will allocate the buffers, construct a bit-map that lasts for the duration of the display, and render the requester in the window on demand from your program or the user. If the requester requires only a simple yes or no answer from the user, you can use the special *AutoRequest()* function that builds the requester, displays it, and waits for the user's response.

On the other hand, you can design your own custom requester with your own hand-drawn image for the background, gadgets, borders, and text. You get your own bit-map with a custom requester, so you can design the imagery pixel-by-pixel if you wish, using any of the Amiga art creation tools. When you have completed the design, you submit it to Intuition for display as usual. Consistency and style are the only restrictions imposed on designing your own requester. The gadgets should look like gadgets and the gadget list should correspond to your images (particularly the gadget select boxes, to avoid confusing the user).

You should always provide a safe way for the user to back out of a requester without taking any action that affects the user's work. This is *very important*.

A user's action or response to a requester can be as simple as telling the requester to go away. Because the user's action consists of choosing a requester gadget, there must be one or more gadgets that terminate the requester.

Another Option

As an option to bringing up a requester, you can flash your screen in a complementary color (binary complement, that is—see the "Images, Line Drawing, and Text" chapter for an explanation). This is handy if you want to notify the user of an event that is not serious enough to warrant a requester and the user doesn't really need to indicate a choice. For instance, the user might be trying to choose an unavailable function from a menu or trying to use an incorrect command-key sequence. If the event is a little more serious, you can flash all the screens simultaneously. See the description of *DisplayBeep()* in Chapter 11, "Other Features".

REQUESTER RENDERING

There are two ways of rendering complex requesters—you can supply Intuition with enough information to do the rendering for you or you can supply your own completely custom bit-map image. You fill in the Requester structure differently according to which rendering method you have chosen.

If you want Intuition to render the requester for you, you need to supply regular gadgets, a "pen" color for filling the requester background, and one or more text structures and border structures.

For custom bit-map requesters, you draw the gadgets yourself, so you supply a valid list of gadgets, but the text and image information in the gadgets structures can be set to NULL because it will be ignored. Other gadget information—select box dimensions, highlighting, and gadget type—is still relevant. The select box information is especially important since the select box must have a well-defined correspondence with the gadget imagery that you supply. The basic idea here is to make sure that the user understands your requester imagery and gadgetry. The fields that define borders, text, and pen color are ignored and can be set to NULL.

For both Intuition-rendered and custom-design requesters, you declare the requester structure and then call *InitRequester()* to initialize the requester.

REQUESTER DISPLAY POSITION

You can have Intuition display the requester in a position relative to the position of the pointer or as an offset from the upper-left corner of the window.

To display the requester relative to the current pointer position, set the POINTREL flag and initialize the *RelLeft* and *RelTop* variables, which describe the offset of the upper left corner of the requester from the pointer position. The values in these variables can be negative or positive. Note that the values you supply are only advisory. If the pointer is in a location that would cause the requester to be rendered outside the window, it will be rendered as close as possible to the desired location but still within the window frame. The actual top and left position are stored in the *TopEdge* and *LeftEdge* variables.

To display the requester as an offset from the upper left corner of the window, initialize the *TopEdge* and *LeftEdge* variables. These should be positive values.

DOUBLE MENU REQUESTERS

A double-menu requester is just like other requesters with one exception. It is displayed only when the user double-clicks the mouse menu button. After the user brings up one of these requesters, window input is blocked as if your program or Intuition had brought up the requester. A message stating that a requester has been brought up in your window is entered into the input stream. If you want to stop the user from bringing up a double-menu requester, (for instance, if you want to modify it or simply just throw it away) you can unlink it from the window.

GADGETS IN REQUESTERS

Each requester gadget should have the REQGADGET flag set in its *GadgetType* variable.

Each requester must have at least one gadget that satisfies the request and allows input to begin again. For each gadget that ends the interaction and removes the requester, you set the ENDGADGET flag in the gadget *Flags*. Every time one of the requester gadgets is selected, Intuition examines the ENDGADGET flag; if the flag is set, the requester is erased from the screen and unlinked from the window's active-requester list.

Algorithmic (Intuition-rendered) and custom bit-map requesters differ in how their gadgets are rendered. In algorithmic requesters, you supply regular gadgets just like the application gadgets in windows or screens. In custom bit-map requesters, the gadgets are part of the bit-map that you supply for display. Even in custom bit-map requesters, however, you do supply a list of gadgets because you must still define the select box, highlighting, and gadget type for each gadget.

IDCMP REQUESTER FEATURES

If you are using the IDCMP for input, the following IDCMP flags add refinements to the use of requesters:

REQVERIFY
    With this flag set, you can make sure that your program is ready to allow a requester to appear in the window. When you receive a REQVERIFY message, the requester will not be rendered until you reply to the message.

REQSET
    With this flag set, you will receive a message when the first requester opens in your window.

REQCLEAR
    With this flag set, you will receive a message when the last requester is cleared from the window.

You set these flags when you create a NewWindow structure or call *ModifyIDCMP()*. See Chapter 8, "Input and Output Methods", for further information about these IDCMP flags.

A SIMPLE, AUTOMATIC REQUESTER

For a very simple requester that prompts the user for a positive or negative response, you can use the *AutoRequest()* function. You supply some explanatory text for the body of the requester, negative and positive text to prompt the user's response, the width and height of the requester, and some optional flags for the IDCMP. The positive text is the text you want associated with the user choice of "Yes", "True", "Retry", and similar responses. Likewise, the negative text is associated with the user choice of "No", "False", "Cancel" and so on. The positive text is automatically rendered in a gadget in the lower left of the requester, and the negative text is rendered in a gadget in the lower right of the requester. The positive text pointer can be set to NULL, specifying that there is no positive choice for the user to make. The IDCMP flags allow either positive or negative external events to satisfy the request. For instance, the positive external event of the user putting a disk in the drive could satisfy the request.
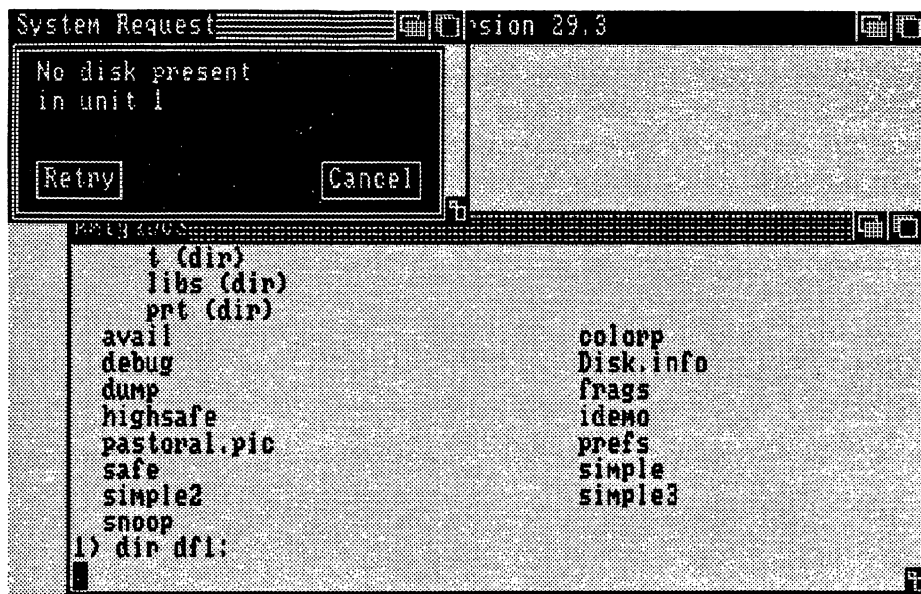
Figure 7-2: A Simple Requester Made With AutoRequest()

When you call the function, Intuition will build the requester, display it, and wait for a response from the user. If possible, the requester is displayed in the window supplied as an argument to the routine. If not, Intuition opens a window to display the requester.

The *AutoRequest()* function calls *BuildSysRequest()* to construct the simple requester. You can call *BuildSysRequest()* directly if you want the simple requester and if you want to monitor the requester yourself. All gadgets created by *BuildSysRequest()* have the following gadget flags set:

BOOLGADGET
> It's a Boolean TRUE or FALSE gadget.

RELVERIFY
> You receive a broadcast if this gadget is activated.

REQGADGET
> Specifies that this is a requester gadget.

TOGGLESELECT
> Specifies that this is a toggle-select type of gadget.

Using Requesters

To create and use a requester, you follow these steps:

1. Declare or allocate a Requester structure.
2. Initialize the structure with a call to *InitRequester()*.
3. Fill out the Requester with your specifications for gadgets, text, borders, and imagery.
4. If you are using the IDCMP for input, decide whether to use the special functions provided.
5. Display the requester by calling either:
   * *Request()*, or
   * *SetDMRequest()* so the user can bring up the requester.

REQUESTER STRUCTURE

To create a requester structure, follow these steps:

1. Fill in the values you need in the structure and leave the unused variables set to the values initialized by the call to *InitRequester()*.
2. Set up a gadget list.
3. Supply a *BitMap* structure if this is a custom requester.

Here is the specification for a requester structure:

```
struct Requester
{
    struct Requester *OlderRequest;
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    SHORT RelLeft, RelTop;
    struct Gadget *ReqGadget;
    struct Border *ReqBorder;
    struct IntuiText *ReqText;
    USHORT Flags;
    UBYTE BackFill;
    struct ClipRect ReqCRect;
    struct BitMap *ImageBMap;
    struct BitMap ReqBMap;
};
```

Here are the meanings of the fields in the requester structure:

NOTE: See "Intuition Rendering" and "Custom Bit-Map Rendering" below for information about how the initialization of the structure differs according to how the requester is rendered.

OlderRequest
> This is a link maintained by Intuition, which points to requesters that were rendered before this one.

LeftEdge, TopEdge
> Initialize these if the requester is to appear relative to the upper left corner of the window (as contrasted to the POINTREL method, where the requester is rendered relative to the pointer).

Width, Height
> Describe the size of the entire requester rectangle, containing all the text and gadgets.

RelLeft, RelTop
> Initialize these if the requester is to appear relative to the current position of the pointer. Also, set the POINTREL flag.

ReqGadget
> A pointer to the first in a linked list of gadget structures.
>
> There must be at least one gadget with the ENDGADGET flag set to terminate the requester.

ReqBorder
> A pointer to an optional Border structure for the drawing lines around and within your requester.

ReqText
> A pointer to an IntuiText structure containing text for the requester.

Flags
> You can set these flags:

> POINTREL
>> Set this flag to specify that you want the requester to appear relative to the pointer (rather than offset from the upper-left corner of your window).

> PREDRAWN
>> Set this flag if you are supplying a custom BitMap structure for the requester and *ImageBMap* points to the structure.

> Intuition uses these flags:

> REQOFFWINDOW
>> Set by Intuition if the requester is currently active and some part of the gadgets was rendered off-window.

> REQACTIVE
>> Set or cleared by Intuition based on whether or not this requester is currently being used.

> SYSREQUEST
>> Set by Intuition if this is a system-generated requester.

BackFill
> Pen number for filling the requester rectangle before anything is drawn into the rectangle.

ReqCRect, ReqBMap
> These are used by Intuition to create the requester image.

ImageBMap
> Pointer to the custom bit-map for this requester. If you are not supplying a custom bit-map for this requester, then Intuition ignores this variable.

> If you are supplying a custom bit-map, the PREDRAWN flag must be set.

The following sections describe the differences in the Requester structure between requesters rendered by Intuition and custom bit-map requesters.

Intuition Rendering

The following notes apply to requesters rendered by Intuition.

o   *ReqGadget* is a pointer to the first in a list of regular gadgets to be rendered in the requester box. Take care not to specify gadgets that extend beyond the requester-rectangle that you describe in the *Width* and *Height* fields, for Intuition does no boundary checking.

o   *ReqBorder* is a pointer to a Border structure for your requester. The lines specified in this structure can go anywhere in the requester; they are not confined to the perimeter of the requester.

o   *ReqText* is a pointer to an IntuiText structure. This is for general text in the requester.

o   *BackFill* is the pen number to be used to fill the rectangle of your requester before any rendering takes place.

For example, here is a Requester structure that allows Intuition to do the rendering:

```
struct Requester MyRequest =
    {
    NULL,                   /* OlderRequester maintained by Intuition */
    20, 20, 200, 100,       /* LeftEdge, TopEdge, Width, Height */
    0, 0,                   /* RelLeft, RelTop */
    &BoolGadget,            /* First gadget */
    NULL,                   /* ReqBorder */
    &MyText                 /* ReqText */
    NULL,                   /* Flags */
    2,                      /* BackFill */
    NULL,                   /* ReqCRect */
    NULL,                   /* BitMap */
    NULL,                   /* Other BitMap */
    };
```

Custom Bit-Map Rendering

These notes apply to custom bit-map requesters.

o   *ReqGadget* points to a valid list of gadgets, which are real gadgets in every way except that the gadget text and imagery information are ignored (and can be NULL). The select box, highlighting, and gadget type data is still pertinent. You must make sure there is an extremely well-defined correspondence between the gadgets' select boxes and the requester imagery that you supply.

o   The *ReqBorder, ReqText,* and *BackFill* variables are ignored, and can be set to NULL.

o   The *ImageBMap* pointer points to your own BitMap of imagery for this requester.

o   You should set the flag PREDRAWN.

THE VERY EASY REQUESTER

Here are the arguments you supply to *AutoRequest()* for the automatic, simple Boolean requester that Intuition will build for you:

Window
   Pointer to the window where the requester is to appear.

BodyText
   Pointer to an IntuiText structure that explains the purpose of the requester.

PositiveText
   Pointer to the IntuiText structure containing the positive response text.
   This field can be NULL if there is no positive response.

NegativeText
   Pointer to the IntuiText structure containing the negative response text.

PositiveFlags
   Flags for the IDCMP for positive external events that will satisfy the request.

NegativeFlags
   Flags for the IDCMP for negative external events that will satisfy the request.

Width, Height
   Size of the rectangle enclosing the requester.

REQUESTER FUNCTIONS

This section gives a brief rundown of the requester functions.

Initializing a Requester

The following function initializes a requester for general use, for both algorithmic and custom bit-map rendering:

InitRequester(Requester)

  Requester points to the requester structure.

Submitting a Requester for Display

The following function submits regular requesters to Intuition for display:

  Request(Requester, Window)

    Displays a requester in the specified window.

Double Menu Requesters

The following functions affect double-menu requesters:

  SetDMRequest(Window, Requester)

    Attaches a requester to the double-click of the mouse menu button.

  ClearDMRequest(Window, Requester)

    Unlinks the requester from the window, and stops the user from bringing it up.

Removing a Requester from the Display

  EndRequest(Requester, Window)

    Erases a requester invoked by the user or application and resets the window. Doesn't remove all requesters, just the one named.

The Easy Yes or No Requester

This function automatically builds, displays, and gets a negative or positive response from a requester:

AutoRequest (Window, BodyText, PositiveText, NegativeText
PositiveFlags, NegativeFlags, Width, Height)

Builds a requester from the arguments supplied, displays the requester, and returns TRUE or FALSE.

Alerts

Alerts are for emergency messages. There are two types: system alerts and application alerts.

System and application alerts display absolutely essential messages and should be reserved for critical communications where the user must take some immediate action; for instance, where the application has experienced a fatal error, or the system has or is about to crash. System alerts are managed entirely by Intuition.
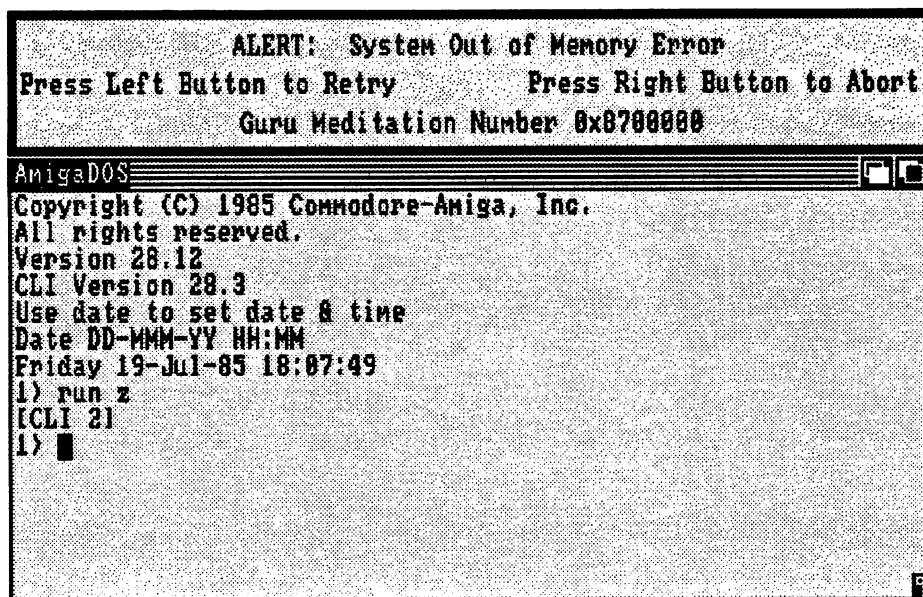
Figure 7-3: The "Out Of Memory" Alert

The sudden display of an alert is a jarring experience for the user, and the system stops and holds its breath while the alert is displayed. For these reasons, you should use alerts only when there is no other recourse. If you can, use requesters with warning messages instead.

The alert display has a black background and red border, a 640 pixel resolution, and can be as tall as needed to display your text. The alert appears at the top of the video display. If the rest of the display is still healthy, it's pushed down low enough to show the alert. If this is a fatal alert and the system is going down, the alert takes over the entire display.

There are two types of alerts: RECOVERY_ALERT, and DEADEND_ALERT.

o   RECOVERY_ALERT displays your text and flashes the alert's border outline while waiting for the user to respond. This alert is optimistic and presumes that the system can continue operations after the alert is satisfied. It returns TRUE if the user presses the left mouse button in response to your message. Otherwise it returns FALSE.

o   DEADEND_ALERT prints your text and returns FALSE to you immediately.

The Boolean function *DisplayAlert()* creates and displays an alert message. Your message will most likely get out to the screen regardless of the current state of the machine (with the exception of catastrophic hardware failures). If the user presses one of the mouse buttons, the display returns to its original state, if possible. *DisplayAlert()* also displays the Amiga system alert messages. *DisplayAlert()* needs three arguments: an AlertNumber, a pointer to a string, and a number describing the required display height.

o   *AlertNumber* is a LONG value. Here you set bits specifying whether this is a RECOVERY_ALERT or a DEADEND_ALERT.

o   The *String* argument points to an AlertMessage string which is made up of one or more sub-strings. Each sub-string contains the following:

*   The first component is a 16-bit x-coordinate and an 8-bit y-coordinate describing where on the alert display you want the string to appear. The y-coordinate describes the location of the text baseline.

*   The second component is the text itself. The string must be null-terminated (it ends with a zero byte).

*   The last component is the continuation byte. If this byte is zero, this is the last sub-string in the message. If this byte is non-zero, there is another sub-string in this alert message.

o   The last argument, *Height*, tells Intuition how many display lines are required for your alert display.

Chapter 8

INPUT AND OUTPUT METHODS

This chapter describes the methods of getting input from the user and sending output to the user. The first section contains an overview of the input/output functions. The next two sections explain the Intuition Direct Communication Message Ports and show how to set up a custom monitor task and user port. The last section introduces the Console Device.

An Overview of Input and Output

Figure 8-1: Watching the Stream

From the Intuition point of view, information flows through the system in the following steps:

o Information originates from somewhere in the user's cranial area.

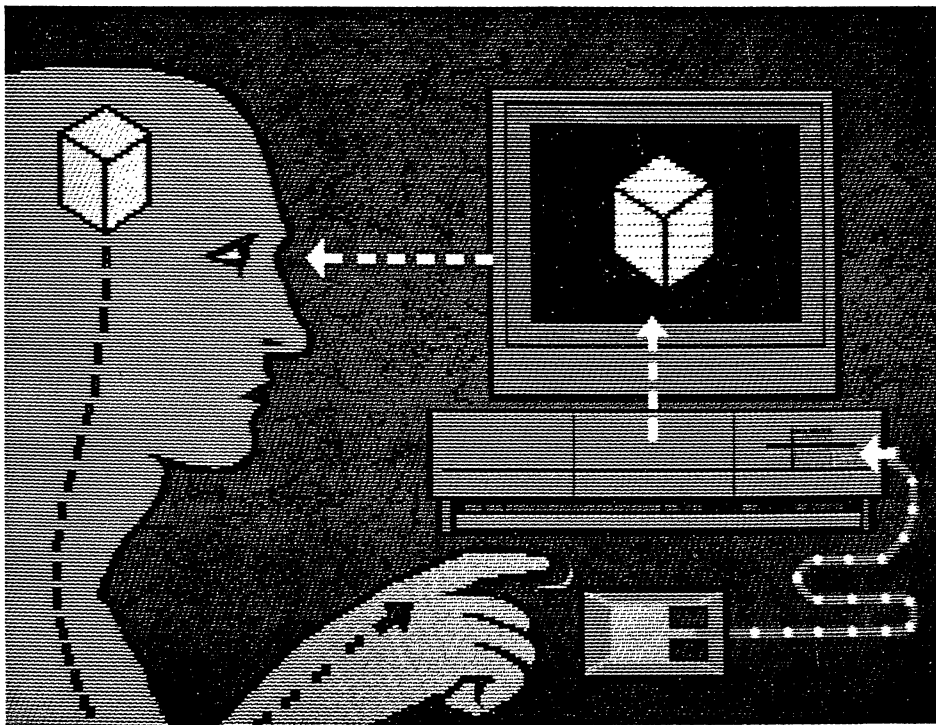o From there, it flows through biological output devices such as fingers and into electro-mechanical input devices like keyboards, mice, graphics tablets, light pens. These input devices create input signals which enter the Amiga through several different ports.

o Inside, these input signals are merged together into a coherent stream of input events.

o This input stream is examined and manipulated by several entities, including Intuition. Intuition gazes deeply into the essence of every event it sees. Sometimes it consumes events, other times it adds to the stream, and often it sits lazily by, watching the stream flow through its fourth dimension.

o Finally, application programs receive the input stream and take action based on the data contained therein. The result of the action is often to create output, which is presented to the user via a video monitor.

o The user's eye input devices detect the information being displayed on the video output device. The eyes, and some still-mysterious merge mechanism, translate the data into signals which are transmitted to the brain, thus completing the cycle.

About Input and Output

The Amiga has an *Input Device* to monitor all input activity, which nominally includes keyboard and mouse activity., but which can be extended to include many different types of input signals. Whenever the user moves the mouse, presses the one of the mouse buttons, or types on the keyboard, the Input Device detects it and constructs an *input event* (a message describing what just occurred). Other devices and programs can ask the Input Device to construct an input message using their own data (for instance, the AmigaDOS is able to generate an input event whenever a disk is inserted or removed, and an application-installed music-keyboard device can add note events to the stream as well). All of these events are merged together into the *input stream*. The Input Device then broadcasts this input event stream through special message ports so that any interested party can monitor the events, intercept some of the events, and even add new ones to the stream. Intuition is one of the interested parties.

Some of the events, like "mouse-button pressed", may have great meaning to Intuition. If they do, Intuition *consumes* them, which is to say that Intuition extracts those events from the input stream. Other events, like the "disk inserted" event, may be of interest to more than one user of Intuition, so Intuition translates these into a separate message for each application. Still other events, like most of the keyboard events, mean nothing to Intuition, and Intuition merely passes them along.

A typical application decides what to do from moment to moment by responding to the events in the input stream. Although many applications may be waiting for input simultaneously, only the application that Intuition regards as active for input will receive these input stream events. Usually, as described in Chapter 4, "Windows", the user selects which application is active for input by using the Intuition pointer to select that application's window. If your program is the active one, then you get to see the input stream events after Intuition has examined them.

Your program receives the input stream either directly from Intuition or via another mechanism known as the *Console Device*. If you don't have any use for the messages either, then the next consumer gets a chance to examine the stream, and so on.

Intuition provides two paths for your program to receive messages from the input stream. One is immediate and involves no pre-processing of the data. The other can supply you with standard terminal input functions, buffers, and data representations. The paths are:

o   Intuition's Direct Communications Message Ports system (IDCMP), which is standard Amiga Exec message communications made easy for you, and which gives you input data in its most raw (untranslated) form. This also supplies the only mechanism you have for communicating *to* Intuition.

o   The Console Device, which gives you "cooked" input data, including key-code conversions to ASCII, and conversions to ANSI escape sequences (Intuition-generated events, like CLOSEWINDOW, will be translated into escape sequences).

When your program wants to present visual information to the user via your window or screen, you can choose from three methods. The one you choose depends on your particular needs. These three methods are:

o   Creating imagery by sending your output directly to the graphics, text and animation primitives of the Amiga ROM kernel. You can use these for rendering functions like line drawing, area fill, specialized animation, and output of unformatted text. This is the most elementary method.

o   Using the Intuition-supplied support functions for rendering text, graphical imagery, and line drawing. These provide many of the same functions as the deeper ROM routines, but these routines do the clerical work of saving, initializing and restoring states for you. Also, the Image functions provide a new method of object-oriented rendering for you.

o   Outputting text via the Console Device, which formats text with special text primitives like ClearEndOfLine() and text functions like auto line-wrapping and scrolling. For string output, if you want to do anything more than the simplest text rendering, you should use the Console Device. This gives you nicely formatted text with little fuss.

Note that the Console Device is mentioned as both a source for input and a mechanism for output. It's very convenient to do both input and output via the Console Device only. In particular, text-only programs can open the Console and do all their I/O there without ever learning anything about Windows, BitMaps, RastPorts, or Message Ports. Use of the Console Device for most text-only applications is encouraged, since it requires less work on your part and simplifies the I/O logic of your programs.

On the other hand, opening a Console Device consumes a fair amount of RAM (currently about 1.5K). If you don't need the Console Device or are willing to forego its features, it may be better for you to open the IDCMP for input and do your graphics rendering directly through the Intuition and graphics primitives. Under some conditions (for instance, when you have a complex program doing lots of different things), you might want to open both the Console Device and the IDCMP for input. There is no rule for deciding which mechanism you should use. After you read this chapter, you'll be able to decide for yourself.

The rest of this section describes in more detail how I/O flow works with (and around) your program. Please refer to the illustrations while you read this. This is actually a super-

simplified model of how system-wide I/O really works, but is a true representation of I/O at the microcosmic level of your program.

In the illustrations that follow, you will find the Input Device at the top of the diagram. This is where mouse, keyboard, and other input events are merged together into a single stream of *InputEvents*. These InputEvents are then submitted to Intuition for further processing.

Figure 8-2 shows an example of a program after it has opened the IDCMP. This will be the typical configuration for games or other applications that are willing to process input data themselves. The IDCMP allows you to configure the events that are important to you. You can, for instance, learn about gadget events and get notification that your application should stop writing to its window (the IDCMP flags SIZEVERIFY and REQVERIFY), but you may not want to learn about other mouse or keyboard events. If you ask to learn about keyboard events through the IDCMP, note that the key codes you receive come straight from the keyboard to you. These keycodes are as raw as they get, although the IDCMP also provides the special Qualifier field to assist your translations. Messages sent via the IDCMP are instances of the structure IntuiMessage. When you open the IDCMP, you must monitor the Message Port supplied by Intuition.
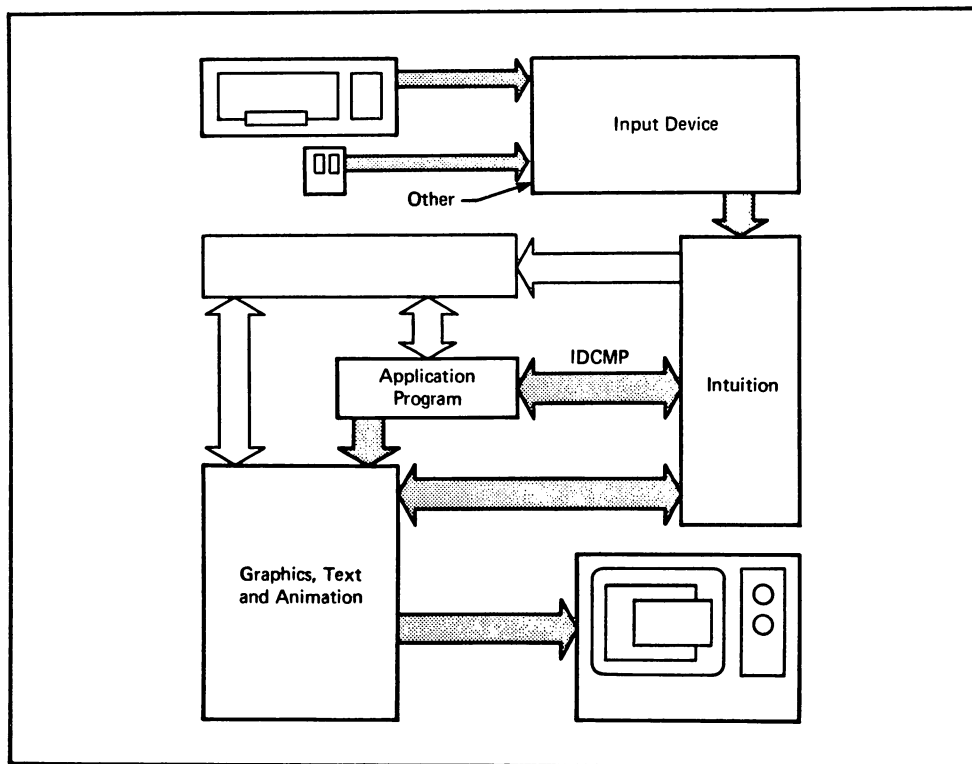
Figure 8-2: Input from the IDCMP, Output through the Graphics Primitives

Figure 8-3 illustrates the flow of information when the only the Console is opened. This will be the typical configuration for text-only applications, and applications that want the simplest I/O possible. Refer to the *Amiga ROM Kernel Manual* for details on opening a Console Device and

performing I/O through it.
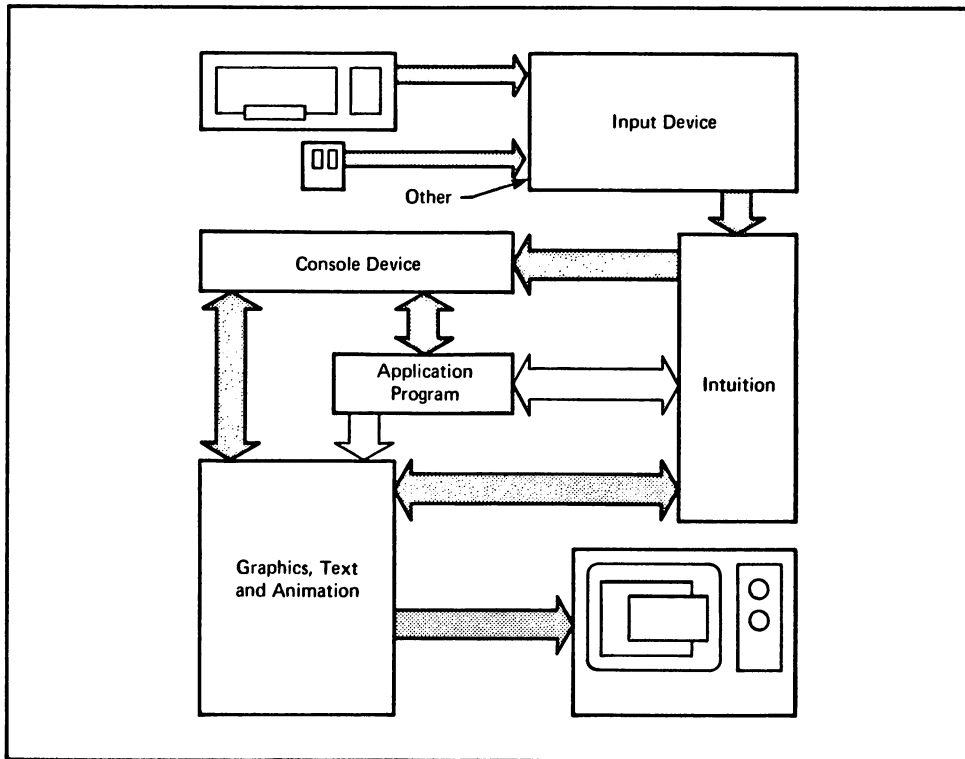
Figure 8-3: Input and Output through the Console Device

Figure 8-4 shows a complex program that needs the features of both the Console Device and the IDCMP. For instance, a program that needs ASCII input and formatted output and the IDCMP verification functions (for example, to verify that it has finished writing to the window before the user can bring up a requester).
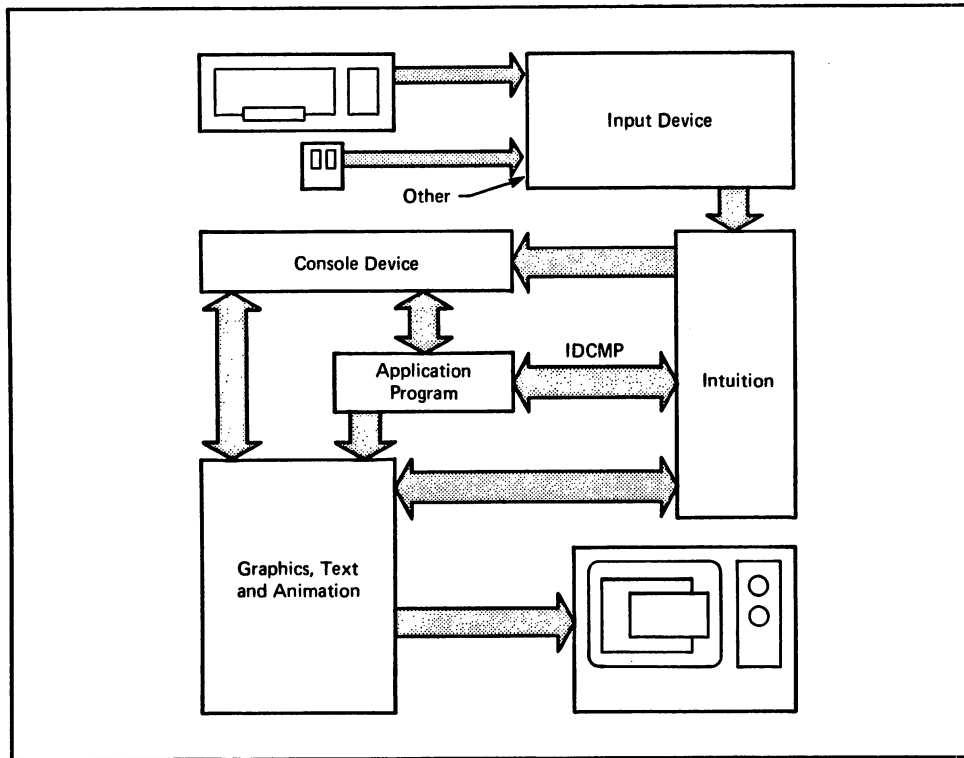
Figure 8-4: Full-System Input and Output (A Busy Program)

Figure 8-5 shows an application that has opened a window with neither a Console nor an IDCMP. This window gets no input, and the application can write to the window only via the graphics primitives. You might want to do this if your program has opened other windows that do I/O and you want special graphics-only windows (for instance, to monitor RAM usage or watch the clock) which you will close later. If the user selects a window that has no Console or IDCMP, further input is discarded until a different window is selected.
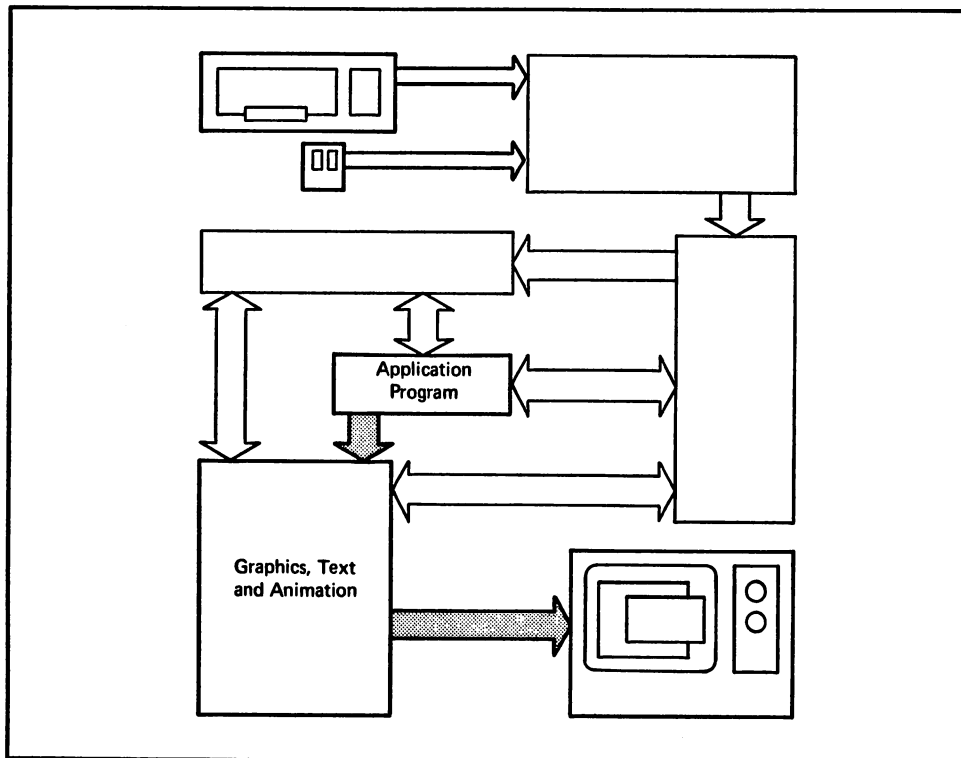
Figure 8-5: Output Only

Using the IDCMP

The IDCMP ports allow your application and Intuition to talk directly to each other. You can use the IDCMP to learn about mouse, keyboard, and Intuition events without going through the Console Device. Intuition also uses the IDCMP, for example, to control the menu display or manage gadget lists. Also, there are certain useful Intuition features, most notably the verification functions (described under *IDCMP Flags* below), which require that the IDCMP be opened as this is the only mechanism available for communicating *to* Intuition.

The IDCMP consists of a pair of *message ports*, which are allocated and initialized by Intuition on your request: one port for you and one port for Intuition. These are standard Exec message ports, used to allow inter-process communications in the Amiga multi-tasking environment. To open these ports automatically, you request IDCMP functions when you define window structures. To open or close them later, you call *ModifyIDCMP()*. As with much of Intuition, all of the "grunt work" with message ports is done for you, leaving you free to concentrate on more

global issues.

Alternatively, if you have a message port that you've already created, you can have Intuition use that port to communicate with you. This is described below.

If you set IDCMP flags in the NewWindow structure when you open the window, Intuition allocates and initializes both message ports. You can call *ModifyIDCMP()* to allocate or deallocate message ports or to change which events will be broadcast to your program through the IDCMP. Once the IDCMP is opened, you can receive many different flavors of information directly from Intuition, based on which flags you have set.

CAUTION: If you attempt to close the IDCMP, either by calling *ModifyIDCMP()* or by closing the window, without first having *Reply()*'d to all of the messages sent out by Intuition, Intuition will reclaim and deallocate those messages without waiting for a *Reply()* from you. If you attempt to *Reply()* after the close, you will get to watch the Amiga FIREWORKS_DISPLAY mode.

To learn more about message ports and message passing, please refer to the *Amiga ROM Kernel Manual.*

INTUIMESSAGES

The IntuiMessage data type is an Exec Message that's been extended to include Intuition-specific information. The Exec Message part of the IntuiMessage is used by the Executive to manage the transmission of the message. The Intuition extensions of the IntuiMessage are used to transmit all sorts of information to you.

Here's what the IntuiMessage looks like:

```
struct IntuiMessage
{
    struct Message ExecMessage;
    ULONG Class;
    USHORT Code;
    USHORT Qualifier;
    APTR IAddress;
    SHORT MouseX, MouseY;
    ULONG Seconds, Micros;
    struct Window *IDCMPWindow;
    struct IntuiMessage *SpecialLink;
};
```

IntuiMessages contain the following components:

ExecMessage
> The ExecMessage data is maintained by Exec. It's used for linking the message into the system and broadcasting it to a message port.

Class
> A ULONG variable, whose bits correspond directly with the IDCMP flags

Code

A USHORT whose bits contain special values like menu numbers or special code values set by Intuition. The meaning of this field is directly tied to the Class (above) of this message. Often, there's no special meaning for the code field, and it's merely a copy of the code of the InputEvent initially sent to Intuition by the Input Device. In the case where this message is of class RAWKEY, this has the raw key code generated by the keyboard device.

Qualifier

Contains a copy of the ie_Qualifier field that's transmitted to Intuition by the Input Device. This field is useful if you are handling raw key codes since the Qualifier tells you, for instance, whether or not the *SHIFT* key or *CTRL* key is currently pressed.

MouseX and MouseY

Every IntuiMessage you receive will have the mouse coordinates in these variables. The coordinates are relative to the upper-left corner of your window.

Seconds and Micros

These ULONG values are copies of the current system clock time in seconds and microseconds. Microseconds range from zero up to one million minus one. 32 bits for the Seconds variable means that the Amiga clock can run for 139 years before wrapping around to zero again.

IAddress

Has the address of some Intuition object, such as a gadget or a screen, when the message concerns, for example, a gadget selection or screen operation.

IDCMPWindow

Contains the address of the window to which this message pertains.

SpecialLink

For system use only.

IDCMP FLAGS

You specify the information you want Intuition to send you via the IDCMP by setting the IDCMP flags. You can set them either in the NewWindow structure when you open a window, or when calling *ModifyIDCMP()* to change the IDCMP specifications. The following is a specification of the IDCMP functions and flags.

Mouse flags:

MOUSEBUTTONS

Reports about mouse button up and down events are sent to you, if these transitions don't mean something to Intuition.

When you receive a MOUSEBUTTONS class of event, you can examine the code field to discover which button was pressed or released. The code field will be equal to

SELECTDOWN, SELECTUP, MENUDOWN or MENUUP.

NOTE: If the user clicks the mouse button over a gadget, Intuition deals with it and you don't hear about it. Also, the only way you can learn about menu button events in this way is by setting the RMBTRAP flag in the window. See Chapter 4, "Windows", for more information.

MOUSEMOVE

Reports about mouse movements are sent in the form of X and Y coordinates. This can mean a lot of messages, so you should reply to them swiftly. See the section called "An Example of the IDCMP" below.

NOTE: This works only if the REPORTMOUSE flag is set in the NewWindow structure, or some gadget is selected with the FOLOWMOUSE flag set.

By setting both RAWKEY and MOUSEMOVE, you don't need a Console Device to get mouse and keyboard input.

DELTAMOVE

When you set this flag, you get mouse movement reports as deltas (amount of change from the last position) rather than as absolute positions. This flag works in conjunction with the MOUSEMOVE flag.

Gadget flags:

GADGETDOWN

When the user selects a gadget you have created with the flag GADGIMMEDIATE set, you will receive a message of this class.

GADGETUP

When the user releases a gadget that you have created with the flag RELVERIFY set, you will receive a message of this class.

CLOSEWINDOW

If the user has selected your window's close gadget, the message telling you about it will be of this class.

Menu flags:

MENUPICK

The user has pressed the menu button. If a menu item was selected, the menu number of the menu item can be found in the *Code* field of the IntuiMessage. If no item was selected, the code field will be equal to MENUNULL.

MENUVERIFY

This is a special verification mode which, like the others, allows you to verify that you're finished drawing to your window before Intuition allows the users to start menu operations.

This is a special kind of verification, however, in that *any* window in the entire screen that has this flag set will have to respond that menu operations may proceed. Also, the active window of the screen is allowed to cancel the menu operation. This is unique to

MENUVERIFY. Please refer to Chapter 6, "Menus", for a complete description.

See the "Verification Functions" section below for some things to consider when using this flag.

Requester flags:

REQSET
Set this flag to receive a message when the first requester opens in a window

REQCLEAR
Set this flag to receive a message when the last requester is cleared from the window.

REQVERIFY
Set this flag if your application wants to make sure that rendering to its window has ceased before a requester is rendered into the window. This includes requiring the system to get your approval before opening a system requester in your window. With this flag set, Intuition sends the application a message that a requester is pending, and and then *Wait()*'s for the application to *Reply()* before drawing the requester into the window.

If several requesters open in the window, Intuition asks the application to verify only the first one. After that, Intuition assumes that all output is being held off until all the requesters are gone. You can set the REQCLEAR flag to find out when *all* requesters are removed from the window. Once the application receives a message of the type REQCLEAR, it is safe to write to the window until another REQVERIFY is received. You can also check the INREQUEST flag of the window, although this is not as safe a method because of the asynchronous nature of any multi-tasking environment.

See the "Verification Functions" section below for some things to consider when using this flag.

Window flags:

NEWSIZE
Intuition sends you a message after the user has re-sized the window. After receiving this, you can examine the size variable in the window structure to discover the new size of the window.

REFRESHWINDOW
A message is sent to the application whenever your window needs refreshing. This flag makes sense only with windows where the SIMPLE_REFRESH or SMART_REFRESH type of refresh has been selected.

SIZEVERIFY
You set this flag if you are drawing to the window in such a way that you need to finish before the user sizes the window. If the user tries to size the window, a message is sent to the application and Intuition will *Wait()* until you reply.

See the "Verification Functions" section below for some things to consider when using this flag.

ACTIVEWINDOW and INACTIVEWINDOW
You set these flags to discover when your window becomes activated or inactivated.

Other flags:

RAWKEY
Keycodes from the keyboard are sent in the *Code* field. They are raw keycodes, so you may want to process them.

NOTE: By combining this and the two mouse flags, you don't need a Console Device to get mouse and keyboard inputs.

NEWPREFS
When the user changes the system Preferences by using the Preferences tool, or when some other routine causes the system Preferences to change, you can find out about it by setting this flag.

When you get a message of class NEWPREFS, you can call the procedure *GetPrefs()* to get the new Preferences.

DISKINSERTED and DISKREMOVED
When the user inserts or ejects any disk with any drive, you can ask to be told about the event by setting either or both of these flags.

Note that everyone who sets these flags will learn about these events, not just the active window.

Verification Functions

SIZEVERIFY, REQVERIFY, and MENUVERIFY are exceptional in that Intuition sends an IntuiMessage and then waits, by calling the Exec message port function *Wait()*, for the application to reply that it's OK to proceed. The application replies by calling the Exec message passing function *ReplyMsg()*.

The implication is that the user requested some operation, but it won't happen immediately and, in fact, won't happen at all until you say it's safe. Because this delay can be frustrating and intimidating, you should strive to make the delay as short as possible. You should always reply to a verification message as immediately as you can.

You can overcome these problems by setting up a separate task to monitor the IDCMP and respond to incoming IntuiMessages immediately. This is recommended whenever you are planning heavy traffic through the IDCMP, which occurs when you have set many IDCMP flags.

SETTING UP YOUR OWN IDCMP MONITOR TASK AND USER PORT

To set up your own IDCMP monitor task, you supply your own port. The addresses of the IDCMP message ports can be found in two variables, UserPort (your application's input port) and WindowPort (Intuition's input port).

In the simplest case, Intuition allocates (and deallocates) both of these ports for you when you define a window with IDCMP flags or call *ModifyIDCMP()*. If the WindowPort isn't already opened when you call one of these functions, it will be allocated and initialized. The UserPort is checked separately to see whether it is already opened. Intuition will send messages to you via the UserPort, and will receive replies from you via the WindowPort. The port variables point to a valid message port if they are opened, and are NULL if not opened.

When Intuition initializes the UserPort for you, Intuition calls *AllocSignal()* to get a signal bit for you. Since your task called *OpenWindow()*, this allocation of a signal is valid for your task. The address of your task is saved into the SigTask variable of the Message Port.

You can choose to supply your own port, if you want. You might do this in an environment where your program is going to open several windows, and you want to monitor input from all of the windows using only one message port. To supply your own port, do the following:

1. Define the window with the variable IDCMPFlags set to NULL, which means no ports will be opened.

2. Set the UserPort variable of the window to any valid port of your own choosing.

3. Call *ModifyIDCMP()* with the flags set as you wish. When Intuition sees that the User-Port variable is non-null, it will assume that the variable points to a valid message port. When Intuition sees that the WindowPort variable is still NULL, a message port will be created.

4. Later, before calling *CloseWindow()*, set UserPort equal to NULL. Intuition will delete the WindowPort, and will detect that the UserPort is not there to be deleted.

An Example of the IDCMP

This section shows a short example of working with the IDCMP. Let's say that we've opened a window that has the following IDCMP flags set:

RAWKEY
MOUSEMOVE
MOUSEBUTTONS
CLOSEWINDOW
GADGETUP
REQSET
REQCLEAR

You could receive and respond to these events using a loop like this:

```
FOREVER
    {
    /* Wait until some message arrives at the port */
    Wait(1 << MyWindow->UserPort->mp_SigBit);

    /* Now, one or more messages have arrived.  Respond to all of them
     * First, set up to accumulate mouse moves (rather than responding
     * to each one as it comes in)
     */
    MouseMoved = FALSE;

    while (message = GetMsg(MyWindow->UserPort))
        {
        /* First, gather some relevant information and then reply right away! */
        class = message->Class;
        code = message->Code;
        address = message->IAddress;
        x = message->MouseX;
        y = message->MouseY;
        ReplyMsg(message);

        if (class === MOUSEMOVE) MouseMoved = TRUE;
        else (ProcessMessage(class, code, address, x, y));
        }

    /* If the mouse moved during the loop, respond to it now */
    if (MouseMoved) ProcessMove(x, y);
    }
```

Using the Console Device

This is an extremely brief description of how you open and use the Console Device. For full
details, refer to the *Amiga ROM Kernel Manual* and the *AmigaDOS Technical Reference
Manual*.

There are two ways to open the Console Device. You can use the one that gives you the power
and flexibility you want and suits the environment in which you are working. You can either
open the Console Device as a normal AmigaDOS file, or you open it directly via a call to *Open-
Device()*. There are advantages and disadvantages to both approaches.

o Opening the Console as an AmigaDOS file

   -   Doing Console input and output via AmigaDOS file-handling is very simple and con-
       venient. Also, there are special line-edit capabilities that you get when opening an Ami-
       gaDOS Console.

- There are, however, two limitations. File I/O requires more processing overhead than going straight to the Console Device. Also, you must be in an AmigaDOS environment (AmigaDOS must be active), which won't be the case for those of you who want to take over the machine.

o Opening the Console Device directly

- When you open the Console Device directly, you have direct control over the parameters and use of the console input and output.

- Opening the Console Device directly is more involved than opening a file; you have to open the device and then send packets of information using a special data structure. Also, you don't have the special line-edit capabilities.

USING THE AMIGADOS CONSOLE

There are two sorts of input that you can get with an AmigaDOS console: unprocessed input through a "RAW:" file type, or processed input either through the DOS's window or through a window of your own choosing.

Getting input from the AmigaDOS console merely involves opening a file with the DOS command *Open()*, and reading from that file with the DOS command *Read()*. These files are simple character-oriented files (also known as byte-stream files). The characters are read into a buffer of your choosing.

To write characters to a window via the AmigaDOS console, you should use the AmigaDOS command *Write()*. When you've finished with console I/O, you should call *Close()* to close the file.

USING THE CONSOLE DEVICE DIRECTLY

To use the Console Device directly, you create an IOStdReq data structure, in which you initialize only one field — the *io_Data* field. You initialize this field with a pointer to your window. Then you call *OpenDevice()*, which opens the Console Device and attaches it to your window. The call to *OpenDevice()* also initializes your IOStdReq structure for subsequent calls to Console Device routines. You can then get input from the Console and send text output to the Console using the functions sketched out below.

Reading from the Console Device

When you want to read from or write to the Console Device, you use the same IOStdReq data structure information that was created by the call to *OpenDevice()*, with the following extra initializations:

o Set the *io_Data* field to point to your *buffer*. A buffer is a block of memory that will be used to to receive the characters from the Console Device.

o   Set the *io_Length* field of the IOStdReq equal to the number of bytes in your buffer. The Console Device will not write more bytes than this into the buffer.

o   Set the *io_Command* field to the constant CMD_READ.

After you initialize the IOStdReq structure with your buffer information, you call either the *SendIO()* or *DoIO()* function to read in any characters that are waiting to be read. The difference between *SendIO()* and *DoIO()* is that *SendIO()* is *asynchronous*, which means that while the Console Device monitors the keyboard you go away and do other processing and check later to see whether or not the user has typed something. *DoIO()*, on the other hand, is *synchronous*, which means that when you call *DoIO()* control doesn't return to you until the user has typed something.

After the call to one of the input routines, you can examine the *io_Actual* field to discover how many characters were actually written into your buffer.

Writing Text to Your Window Via the Console Device

You can write characters to your window or do special formatting by writing *control escape sequences* to the Console Device. Control escape sequences are special sequences of characters that start with the "Escape" character, which is a character with the byte value of 155 (that's 0x9B in hex). This character is also known as the *control sequence introducer*, or CSI.

When you want to read from or write to the Console Device, you use the same IOStdReq data structure information that was created by the call to *OpenDevice()*, with the following extra initializations:

o   Put the characters (and control escape sequences) you want written to your window into a buffer, and put the address of the buffer into the *io_Data* field of the IOStdReq structure.

o   Initialize the field *io_Length* with the number of characters that are found in the *io_Data* buffer. Or, if your text is null-terminated, you can specify a length of -1 and let the Console Device figure out the length for you.

o   Set the *io_Command* field to CMD_WRITE.

Text is written entirely within the non-border area of a window (it doesn't matter what sort of refresh mode the window has). When writing text with the Console Device, you never have to worry about the text being written over the gadget imagery in the borders of the window.

Character-wrap is supported at edge-of-window boundaries. Character wrap is a special feature of all of the console devices. It allows the console devices to behave like virtual terminals. Character-wrap means that if a character to be written won't fit in the remaining space of the current line, the Console Device will write the character in the first position of the next line instead. Compare this with writing text directly into a window using the text primitives: if your character string reaches the boundary of the window, it will be written out in the invisible space beyond the window.

The control escape sequences can be used for special text operations like LINE FEED, CLEAR_END_OF_LINE, and cursor movements. The complete list of control functions available from the Console Device is quite long. For a complete list of these and other special control functions of the Console Device, please refer to the *Amiga ROM Kernel Manual.*

SETTING THE KEYMAP

The *keymap* is the translation table that the Console Device uses when translating the raw key-codes that come from the Keyboard Device into normal characters (usually ASCII) for your program to use. If you never bother with the keymap of your virtual terminal, then you will get plain ASCII translations of the characters typed at the keyboard. These are equivalent to the characters that are printed on the keys of the Amiga keyboard.

The keymap also describes higher-level functions such as which keys repeat, which keys combine with the control keys to result in special control-key sequences, and more. The default Console Device keymap configures these functions to look like a generic terminal.

You can supply your own keymapping translation tables if you like. For example, if you are supporting something like a Dvorak keyboard, you can map the input signals to your own choice of alphanumerics.

You can see the current keymap table by using the *CDAskKeyMap()* routine, which returns a copy of the table to you. You can set your own keymap by calling the *CDSetKeyMap()* routine with your own table.

Chapter 9

IMAGES, LINE DRAWING, AND TEXT

Intuition provides two approaches to rendering graphics images, lines, and text into displays:

o   For quick and easy rendering, Intuition provides its own high-level data structures and functions.

o   You are also free to use all of the lower-level Amiga graphics, animation, and text primitives.

This chapter shows you how to you use the Intuition structures and functions, but the Amiga primitives are a large topic in themselves and we can only point the way. You will find instructions for using the primitives in the *Amiga ROM Kernel Manual.*

The first section of this chapter describes the three Intuition structures for graphics, lines, and text. In the section for each structure, there is initially some general information, then the complete specification for the formal data structure, and an example or two.

The last part of this section shows how to use Intuition functions to display your graphics, line, and text data structures in windows and screens. The mechanics of displaying the structures in windows and screens differ slightly from the way you display them in requesters, gadgets, and menus.

The last section shows how to get the address of display memory for windows and screens. You need this information to use both the graphics primitives and the Intuition functions.

Using Intuition Graphics

Images, Borders, and IntuiText are the general-purpose Intuition structures for rendering graphics and text into your display. They are called *illustration data types.*

o   Images are graphic objects of any size and complexity.

o   Borders are connected lines of any length and number, drawn at any angle, and defining any arbitrary shape.

o   IntuiText strings can be written in the default font or in a custom font of your own design.

The illustration data types are easy to design and economical to use. They are easy to design because their definitions are brief and flexible. Even though each structure defines a different data type, the data types share a consistency of features and capabilities, so once you've learned

one you've pretty much learned them all. This decreases the amount of energy spent in learning new things, and you can reuse the same structures in many places. It also buys an economy of Intuition-internal routines, so we all win.

Each of these illustration data types is located with respect to a *display element*, or *containing element*, which can be any of the primary Intuition components: a window, screen, menu, gadget, or requester. The starting location of an image, border, or text string is defined as an offset relative to some particular pixel, usually the top-left corner of the element. Any of the illustration data types can be rendered in any of the display elements. In fact, you can display the same structure in more than one of the elements at the same time.

There are two methods of rendering images, borders, and text into display elements:

> o In menus, gadgets, and requesters, you use a pointer field provided in the menu, gadget or requester structure. Then, as Intuition handles those structures, the illustrations are rendered for you.
>
> o In windows or screens, you render the illustration types directly into the display element by using one of the functions *DrawImage()*, *DrawBorder()*, or *PrintIText()*.

In the definitions of all three of these general-purpose structures, you supply a top-left location that is a relative offset from the top-left of the display element that will contain the illustration. These relative offsets allow you to use the underlying data arrays across limitless instances of Image, Border, or IntuiText structures. For example, if you have numerous gadgets of the same size, you can use the same Border coordinate pairs to draw a line around each gadget.

An important fact about the illustration elements is that each can point to another of its own kind. You can link many of them together and have them all rendered with just one procedure call.

DISPLAYING BORDERS, INTUITEXT, AND IMAGES

Requester, gadget and menu structures contain a field for rendering borders, text and images. This field contains a pointer to an instance of a Border, IntuiText, or Image structure. For rendering the illustration types directly into screens and windows, however, you use the Intuition functions *DrawBorder()*, *DrawImage()*, and *PrintIText()*. You supply a Border, Image, or IntuiText structure as an argument to the function.

Note that the offsets you specify as arguments to these functions are added to the offsets in the graphics structures. Sometimes this extra level of offset can come in handy, especially when positioning as a group a linked list of illustration structures.

For rendering into screens and windows, you also need a pointer into the window or screen RastPort. See the "Using the Graphics Primitives" section below.

CREATING BORDERS

Although this data structure is called a Border, it's actually a general purpose structure for drawing connected lines at any angles and rendering any arbitrary shape made up of groups of connected lines. It's called a Border because that's how it started out.

To define a Border, you specify the following:

- o   A set of x and y offsets to the beginning point of the line
- o   A set of coordinate pairs for each vertex
- o   Two colors and a drawing mode:
    - *   A color for the lines
    - *   A color that can be used for background areas enclosed within lines
    - *   One of several drawing modes
- o   Optional pointer to another instance of Border

Border Coordinates

Intuition draws lines between points that you specify as as sets of x,y coordinates. The Border variables *LeftEdge* and *TopEdge* contain offsets to the first pair of coordinates. The *XY* field contains a pointer to an array of coordinate pairs. All of these coordinates are offsets from the top-left corner of the element that contains the line. Thus, you can define one line and use it in different display elements or use it many times in the same element. The first coordinate pair describes the starting point of the first line. Every coordinate pair after the first describes the ending point of the current line and, if there's another coordinate pair, the starting point of the next line.

Here's an example for you. Consider a gadget whose select box is 140 pixels wide and 80 pixels high. The top-left corner of the gadget's select box is located in a window at position (10,5). If the border's (*LeftEdge, TopEdge*) coordinates are (10,10), this results in an absolute base position of (10+10,5+10), or (20,15), as shown in Figure 9-1.
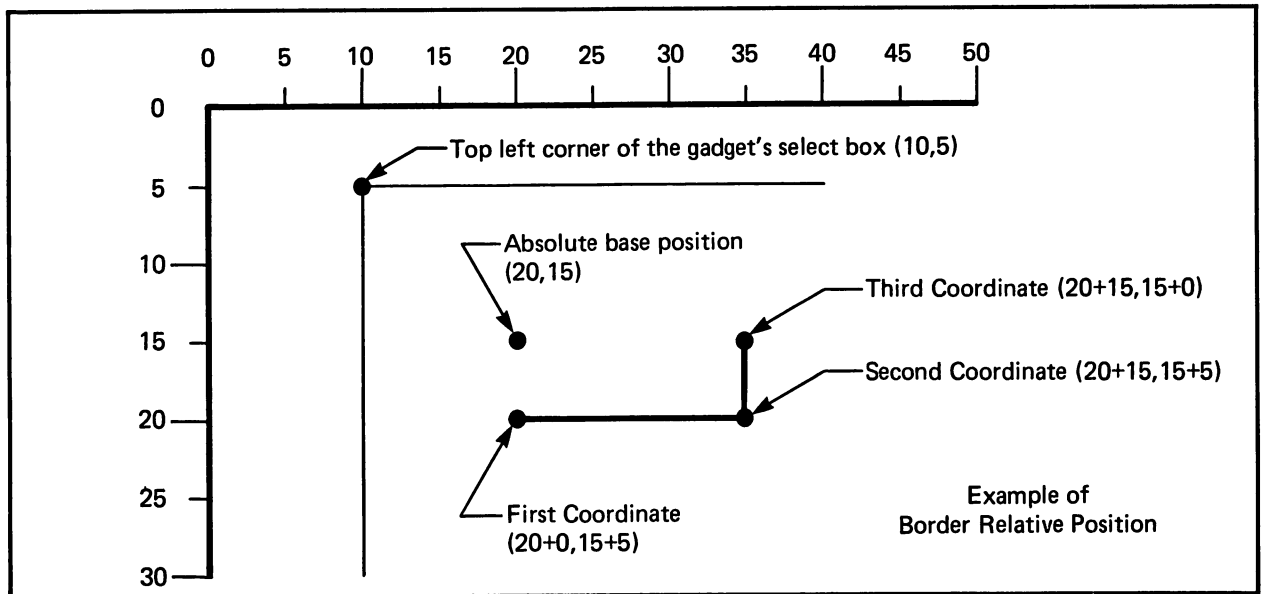
Figure 9-1: Example of Border Relative Position

The (LeftEdge, TopEdge) coordinate pair define the absolute base pixel for this border. All coordinate pairs of the border are relative to this point. If the first set of coordinates in the array of coordinates is (0,5), the starting point of the first line will be at (20+0,15+5), or (20,20). If the next coordinate pair is (15,5), the end point of the first line will be at (20+15,15+5), or (35,20). A line will be drawn from absolute position (20,20) to absolute position (35,20). If there is one last coordinate pair, (15,0), then the next point is at (20+15,15+0), or (35,15). A second line segment is drawn from (35,20) to (35,15).

For a border that is outside the select box of a gadget, you can specify negative offsets. For example, starting position (-1,-1) for a gadget border is just outside the gadget select box.

Border Colors and Drawing Modes

Intuition uses the current set of colors in the color register to draw the border and, optionally, to draw its background. As usual, the available colors depend upon the number of bit-planes used in the screen. For instance, if the screen has 5 bit-planes, then you can select from the colors in color registers 0 through 31. The lines are always drawn in the color in the *FrontPen* field.

Two drawing modes pertain to border lines: JAM1, and XOR. To draw the line in your choice of color, use JAM1. You can choose to have the line "invert" the color of the pixels over which it is drawn by selecting the XOR drawing mode. If you use XOR mode, then for every pixel the line is drawn over, the data bits of the pixel are changed to their binary complement. The complement is formed by reversing the all the 0-bits and 1-bits in the binary representation of the

color register number. In a 3-bit-plane display, for example, color 6 is 110 in binary. If a pixel is color 6, it will be changed to the complement of 001 (binary), which is color 1.

Linking Borders Together

The *NextBorder* field can point to another instance of a Border structure. This allows you to link borders together to describe complex line-draw shapes. Having multiple borders allows you to draw multiple, distinct groups of lines, each with its own set of line segments, and its own color and draw mode. For example, you may want a double border to make a requester stand out more from the surrounding display. You can define the inner border in a second Border structure and link it to the first structure by using this field.

Border Structure Definition

Here is the specification for a Border structure:

```
struct Border
{
        SHORT LeftEdge, TopEdge;
        SHORT FrontPen, BackPen, DrawMode;
        SHORT Count;
        SHORT *XY;
        struct Border *NextBorder
};
```

The meanings of the fields in the Border structure are:

LeftEdge, TopEdge

> Starting origin for the border as an offset from the top-left of the containing element. LeftEdge is the x coordinate and TopEdge is the y coordinate for the top-left bit of the image. This field can contain integers or constants.

LeftEdge
> Number of pixels from left edge of containing element.

TopEdge
> Number of lines from top line of containing element.

FrontPen, BackPen, DrawMode

> *FrontPen* is the color used to draw the line. The pen color fields contain integers or constants that correspond to color registers. *BackPen* is currently unused.

You set the *DrawMode* field to one of the following:

JAM1
> Uses *FrontPen* to draw the line and makes no change in the background.

XOR
> Changes the background beneath the line to its binary complement.

NextBorder
> Pointer to another instance of a Border structure.
>
> Set this field to NULL if there is no other Border structure or if this is the last Border structure in the linked list.

XY
> Pointer to an array of coordinate pairs, one pair for each line.

Count
> The number of pairs in the array of coordinate pairs.
>
> This field contains an integer or constant.

CREATING TEXT

The IntuiText structure provides a simple way of writing text strings anywhere in your display. For example, an array of IntuiText strings is very handy in creating menus.

To define and display IntuiText, you specify the following:

- o Colors for the text and, optionally, for the text's background.
- o One of three drawing modes
- o Starting location for the text
- o Default font or your own special font
- o Pointer to another instance of IntuiText (if any).

Text Colors and Drawing Modes

As with border colors, Intuition uses the current set of colors in the color register to write the text and, optionally, to draw its background. As usual, the available colors depend upon the number of bit-planes used in the screen. For instance, if the screen has 5 bit-planes, then you can select from the colors in color registers 0 through 31. The text is usually drawn in the color in the *FrontPen* field.

Text characters in general are made of two areas: the character image itself, and the background area surrounding the character image.

In addition to the two drawing modes for borders, JAM1 and XOR, you also have JAM2. These modes are described in the following paragraphs.

If you select JAM1 drawing mode, the text character images will be drawn but the character background areas won't be drawn. The character image is drawn in *FrontPen* color. Because the background of a character is not drawn, the pixels of the destination memory around the character image are not disturbed. This is called *overstrike*.

If you select JAM2 drawing mode, the character image is drawn in *FrontPen* and the character background is drawn in the color in the *BackPen* field. Using this mode, you completely cover any graphics that previously appeared beneath the letters.

If drawing mode is XOR, the character is drawn in the binary complement of the colors at its destination. The destination is the display memory where the text is rendered. *FrontPen* and *BackPen* are ignored. To form the complement, you reverse the all the 0-bits and 1-bits in the binary representation of the color register number. In a 3-bit-plane display, for example, color 6 is 110 in binary. The complement is 001 (binary), which is color 1.

Linking Text Strings

The *NextText* field can point to another instance of an IntuiText structure. Using this field, you can have several distinct groups of characters rendered with one stroke, each with its own color, font, location, and drawing mode.

Starting Location

The starting *TopEdge* for a text string is the topmost pixels of the tallest characters. Note that this is different from the baseline of the text. The baseline is the horizontal line on which the characters and punctuation marks rest. The system default fonts are designed to be both above and below the baseline. The descenders of letters (the part of certain letters that is usually below the writing line, like the tail on the lower-case "y") are rendered below the base line. Therefore, you need to allow for this in rendering text in the display. For more information about text imagery, refer to the *Amiga ROM Kernel Manual.*

Fonts

You can use the default font, as set by Preferences, or you can have your own custom font in a FontDesc structure and use the *TextAttr* field to point to the custom font. For more information about custom fonts, see the *Amiga ROM Kernel Manual.*

IntuiText Structure

Here is the specification for an IntuiText structure:

```
struct IntuiText
{
    UBYTE FrontPen, BackPen;
    UBYTE DrawMode;
    SHORT LeftEdge;
    SHORT TopEdge;
    struct TextAttr *ITextFont;
    UBYTE *IText;
    struct IntuiText *NextText;
}
```

The meanings of the fields in the IntuiText structure are as follows.

FrontPen, BackPen

    FrontPen is the color used to draw the text. *BackPen* is the color used to draw the background for the text, if JAM2 drawing mode is specified.

    These fields contain integers or constants that correspond to color register numbers.

DrawMode

    One of three drawing modes:

    JAM1
        FrontPen is used to draw the text; background color is unchanged.

    JAM2
        FrontPen is used to draw the text; background color is changed to the color in *BackPen*.

    XOR
        The characters are drawn in the complement of the background.

LeftEdge

    Starting position for the text as an offset from the left corner of the containing element.

    This field contains an integer or constant, which is the number of pixels from left edge of containing element.

TopEdge

    Starting position for the text as an offset from the top line of the display element.

    This field contains an integer or constant, which is the number of lines from top line of containing element.

TextAttr

    Pointer to a TextAttr structure containing your own font description. Set to NULL if you want the default font.

IText
> Pointer to null-terminated text.

NextText
> Pointer to another instance of IntuiText, if this is part of a linked list of text strings.
>
> Set this field to NULL if this is not part of a list or if this is the last structure in the list.

CREATING IMAGES

With an Image structure you can create graphics objects quickly and easily and display them almost anywhere. Images have an additional attribute that makes them even more economical—with one minor change in the structure, you can display the same image in different colors within the same display element.

To define and display an Image, you specify the following:

o   Location of the Image within the display element

o   Image data — width and height of the Image and the data to create the Image

o   Depth of the Image; that is, how many bit-planes are used to define it

o   Bit-planes in the display element that are used to display the Image. This determines the colors in the image.

Image Location

You specify a location for the Image that places its top-left corner as an offset from the top-left corner of the element that contains the Image.

Defining Image Data

To create the data for your image, you write 1's and 0's into a block of 16-bit memory words, which are located at sequentially increasing addresses. When the image is displayed, this sequential series of memory words is organized into a rectangular area, called a bit-plane. You can have up to 6 bit-planes in an image; they are drawn together when the image is displayed.

The color of each pixel in the image is directly related to the value in one or more memory bits depending upon:

o   how many bit-planes there are in the image data, and

o   in which bit-planes of the screen or window you choose to display your image.

The color of a given pixel is determined by one or more data bits Each bit in the pixel is taken from the same position in each of the bit-planes used to define the image. For each pixel, the

system combines all the bits in the same position to create a binary value that corresponds to one of the system color registers. This method of determining pixel color is called color indirection because the actual color value is not in the display memory. Instead, it is in color registers which are located somewhere else in memory.

If an image consists of only one bit-plane and is displayed in a one-bit-plane display, then:

o   Wherever there is a 0 bit in the image data, the color in color register 0 is displayed.

o   Wherever there is a 1-bit, the color in color register 1 is displayed.

In an image composed of two bit-planes, the color of each pixel is obtained from a binary number formed by the values in two bits, one from bit-plane 0 and one from bit-plane 1. If bit-plane 0 contains all 1's and bit-plane 1 contains 0's and 1's, the pixels derive their colors from register 1 (binary 01) and register 3 (binary 11).

You create your image data by giving Intuition a series of data words. Before specifying these numbers, you may find it helpful to lay out your image on graph paper, or to use one of the Amiga art tools to assist you. For example, the following illustration shows the layout for the system sizing gadget, which is a one-bit-plane image.

Image Data		Hexidecimal Representation		
	F	F	F	F
	C	0	F	F
	C	C	F	F
	C	0	0	3
	F	C	F	3
	F	C	F	3
	F	C	F	3
	F	C	0	3
	F	F	F	F
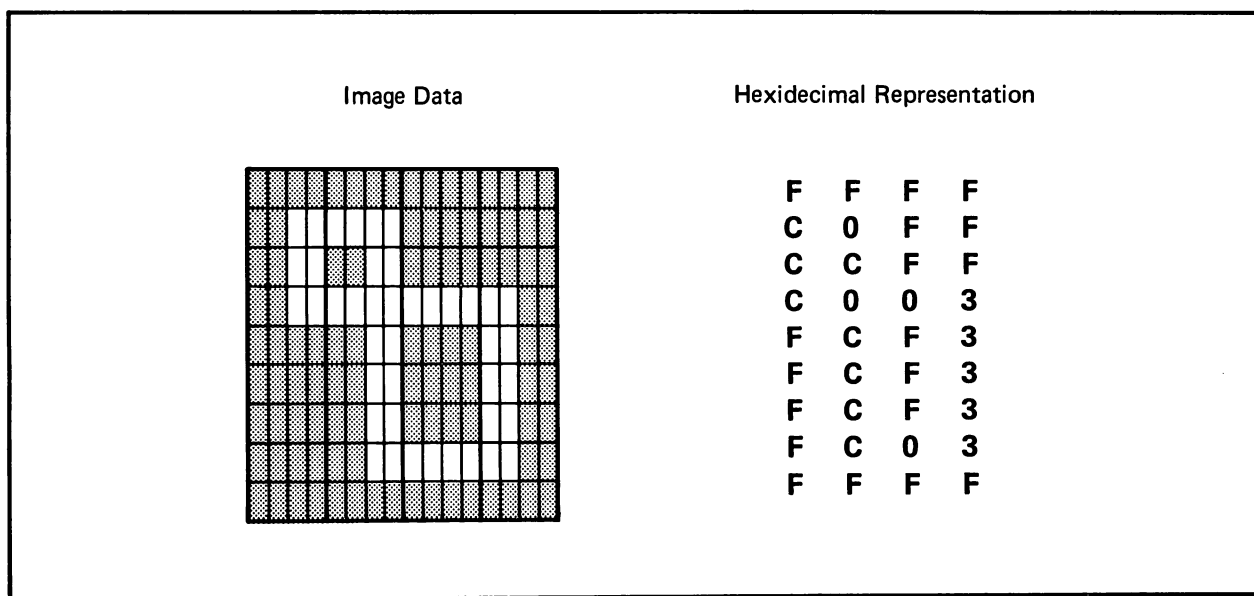
Figure 9-2: Intuition's High-Resolution Sizing Gadget Image

In hex notation, the data words of the sizing gadget image are defined as follows:

```
USHORT SizeData[] =
    {
    0xFFFF,
    0xC0FF,
    0xCCFF,
    0xC003,
    0xFCF3,
    0xFCF3,
    0xFCF3,
    0xFC03,
    0xFFFF,
    };
```

In the image data, you need to specify enough whole words to contain the Image width. For example, an image 7 bits wide requires one word per line while an image 17 bits wide requires two words per line. In the *Width* field of the Image structure, you specify the actual width in pixels of the widest part of the image not how many pixels are contained in the words that define the image. The *Height* field contains the height of the image in pixels.

Here is the actual Image structure of the system sizing gadget. The last two fields in the structure, *PlanePick* and *PlaneOnOff* are explained in the next section.

```
struct Image SizeImage =
    {
    0, 0,           /* left top */
    16, 9, 1,       /* width, height, depth */
    &SizeData[0],   /* Address
    0x1, 0x0,       /* PlanePick, PlaneOnOff */
    NULL,           /* NextImage */
    };
```


Picking Bit-Planes for Image Display

You use the *PlanePick* and *PlaneOnOff* fields in the Image structure to specify which bit-planes of the containing window or screen are used to display the image. This gives you great flexibility in using Image structures. You can:

o   draw an Image into a screen or window of any depth (if you've designed it right)

o   make one Image and display it in different colors

o   minimize the amount of memory needed to define a simple Image that is destined for a display of multiple bit-planes

PlanePick "picks" the bit-planes of the containing window or screen RastPort that will receive the bit-planes of the Image. *PlaneOnOff* specifies what to do with the window or screen bit-planes that are not picked to receive image data. For each display element plane that is "picked" to receive data, the next successive plane of image data is drawn there. For every

bit-plane not picked to receive image data, you tell Intuition to fill the plane with 0's or 1's. For both variables, the binary form of the number you supply has a direct correspondence to the bit-planes of the window or screen containing the Image. The lowest bit position corresponds to the lowest-numbered bit-plane. For example, for a window or screen with 3 bit-planes (consisting of Planes 0, 1, and 2), all the possible values for *PlanePick* or *PlaneonOff* and the planes picked are as follows.

PLANEPICK OR PLANEONOFF	PLANES PICKED
000	No planes
001	Plane 0
010	Plane 1
011	Planes 0 and 1
100	Plane 2
101	Planes 0 and 2
110	Planes 1 and 2
111	Planes 0, 1 and 2

The system sizing gadget shown above has only one bit-plane of data. To display this gadget in Plane 0 of a 4-bit-plane window using Color 1 for the Image and Color 0 for its background you set *PlanePick* to 0001 (binary) and *PlaneOnOff* to 0000 (binary). These settings give Intuition the following instructions:

o   Display the data that describes the image in plane 0 of the destination RastPort

o   For all of the other planes in the RastPort, set the bits in the area where the Image is displayed to 0.

Figure 9-3 illustrates the discussion in the preceding paragraphs.

Bit Map Planes			

	Plane 3	Plane 2	Plane 1	Plane 0
Planepick:	0	0	0	1
Planeonoff:	0	0	0	don't care

Image Data
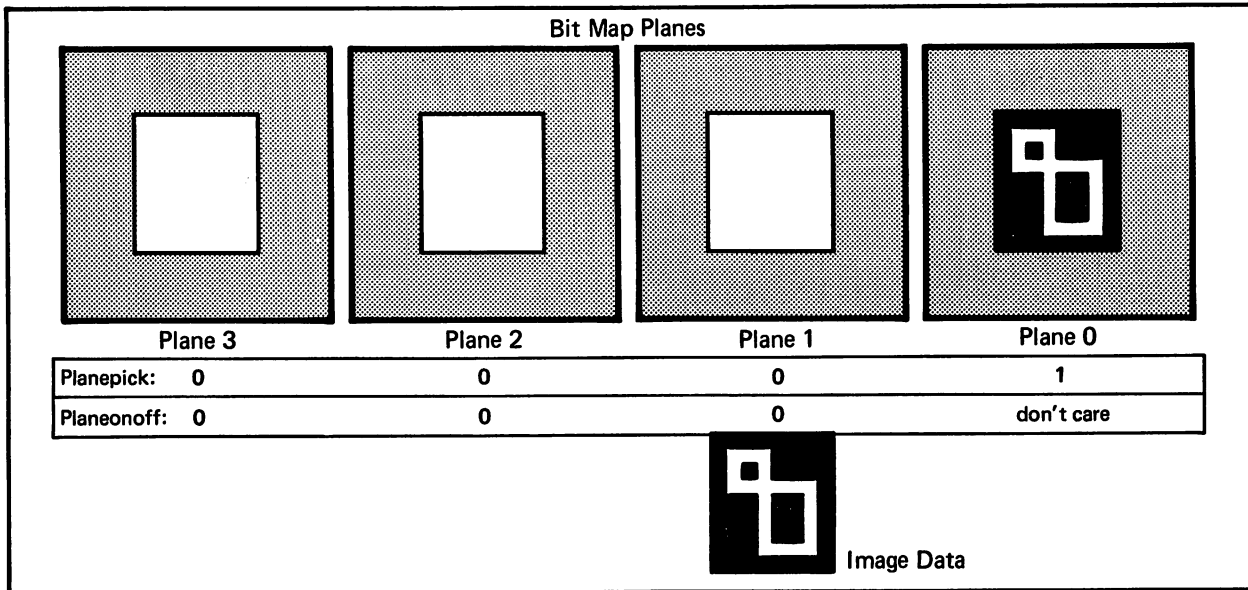
Figure 9-3: Example of PlanePick and PlaneOnOff

If you want the sizing gadget to be rendered in Color 2 and its background in Color 0, you need to define pixels whose values are 0010 and 0000. To do this, simply change *PlanePick* to 0010.

If you want Color 3 for the sizing gadget and color 1 for its background, you need to define pixels with values 0011 and 0001. Therefore, plane 1 defines the image and plane 0 has to be all 1's. You can achieve this by setting *PlanePick* to 0010 and *PlaneOnOff* to 0001.

If you want an Image that is simply a filled rectangle, you don't have to supply any image data at all! You specify a *Depth* of zero, set *Width* and *Height* to any size you like, and set *PlanePick* to 0000 since there are no planes of image data to pick. Then, set *PlaneOnOff* to the color you want for the rectangle. To see how a gadget like this looks, refer to the "Requester Deluxe" illustration, Figure 7-1, in Chapter 7, "Requesters and Alerts".

Image Structure

Here is the specification for an Image structure:

```
struct Image
{
        SHORT LeftEdge, TopEdge;
        SHORT Width, Height, Depth;
        SHORT *ImageData;
        UBYTE PlanePick, PlaneOnOff;
        struct Image *NextImage;
};
```

The meanings of the fields in the Image structure are:

LeftEdge, TopEdge

> Offsets from the top-left of the display element.
>
> These fields contain integers or constants:

LeftEdge
> Number of pixels from left edge of display element.

TopEdge
> Number of lines from top line of display element.

Width

> Width in pixels of the actual image.
>
> This field contains an integer or constant.

Height, Depth

> Height of the image in pixels and number of bit-planes needed to define the image.
>
> These fields contain integers or constants.

ImageData
> Pointer to the actual bits defining the image.

PlanePick, PlaneOnOff

> *PlanePick* tells which planes of the containing element you "pick" to receive planes of image data. *PlaneOnOff* tells what to do about the planes that are not "picked".
>
> These fields represent bit-plane numbers.

Image Example

Here is a more complex example of an Image. The Image shown in Figure 9-4 below belongs to one of the system depth-arrangement gadgets (the front gadget, which brings a window or screen to the front of the display):

The 3-Color Front Gadget   Plane 0, Works even in   Plane 1, for Highlight
One-plane Screens
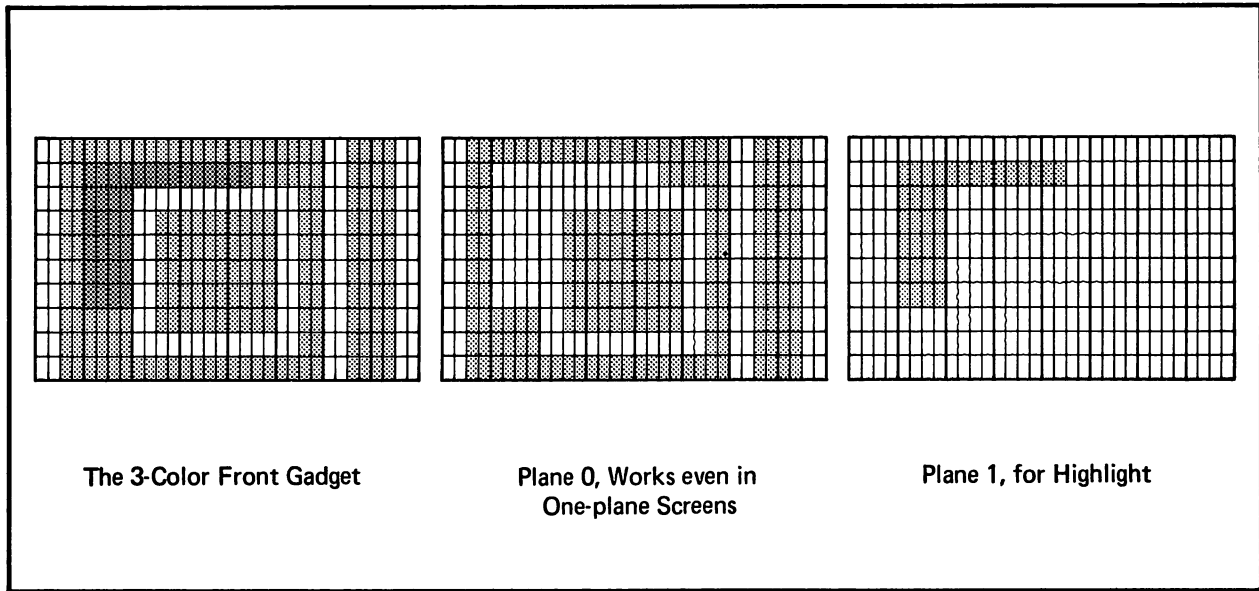
Figure 9-4: Example Image — the Front Gadget

Its data structure and data definition look like this:

```
USHORT UpFrontData[] =
    {
    0x3FFF, 0xFF3C,
    0x3000, 0x3F3C,
    0x3000, 0x033C,
    0x303F, 0xF33C,
    0x303F, 0xF33C,
    0x303F, 0xF33C,
    0x303F, 0xF33C,
    0x3F3F, 0xF33C,
    0x3F00, 0x033C,
    0x3FFF, 0xFF3C,
    /**/
    0x0000, 0x0000,
    0x0FFF, 0xC000,
    0x0F00, 0x0000,
    0x0F00, 0x0000,
    0x0F00, 0x0000,
    0x0F00, 0x0000,
    0x0F00, 0x0000,
    0x0000, 0x0000,
    0x0000, 0x0000,
    0x0000, 0x0000,
    };

struct Image UpFImage =
    {
    0, 0,           /* left top */
    29, 10, 2,          /* width, height, depth */
    &UpFrontData[0],  /* image data */
    0x3, 0x0,           /* PlanePick, PlaneOnOff */
    NULL,         /* NextImage */
    };
```

This gadget was designed to look good in a window or screen of any depth. *PlanePick* 0x3 (000011) picks Planes 0 and 1 of the destination RastPort for Planes 0 and 1 of the gadget, respectively. If this gadget is displayed in a window or screen of depth 1, only Plane 0 of its data is displayed. Color 0 is used for the background and Color 1 for the imagery.

If this gadget is displayed in a window or screen of depth 2 or more, both planes are displayed. The resulting colors are 0 for the background and 1 and 2 for the imagery.

INTUITION GRAPHICS FUNCTIONS

Following are brief descriptions of the Intuition functions that relate to the use of the Intuition illustration data types and the Amiga graphics primitives.

Rendering Images, Lines, or Text into a Window or Screen

DrawImage (RPort, Image, LeftOffset, TopOffset)

Moves the Image data into the RastPort of the screen or window.

RPort = pointer to the RastPort
Image = pointer to an Image structure
LeftOffset = offset added to the Image's x coordinate.
TopOffset = offset added to the Image's y coordinate.

DrawBorder (RPort, Border, LeftOffset, TopOffset)

Draws the vectors of the Border into the window or screen RastPort.

RPort = pointer to the RastPort
Border = pointer to a Border structure
LeftOffset = offset added to each vector's x coordinate.
TopOffset = offset added to each vector's y coordinate.

PrintIText (RPort, IText, LeftOffset, TopOffset)

Prints IntuiText into the window or screen RastPort.

RPort = pointer to the RastPort to receive the text
IText = pointer to an IntuiText structure
LeftOffset = offset added to IntuiText x coordinate.
TopOffset = offset added to IntuiText y coordinate.

Obtaining the Width of a Text String

IntuiTextLength (IText)

Returns the width in pixels of an IntuiText. *IText* is a pointer to an instance of an IntuiText structure.

Obtaining the Address of a View or ViewPort

ViewAddress ()

> This function returns the address of the Intuition View structure for any graphics, text or animation primitive that requires a pointer to a View.

ViewPortAddress (window)

> This functions returns the address of the screen ViewPort associated with the specified window, for any graphics, text or animation primitive that requires a pointer to a ViewPort.

Using the Amiga Graphics Primitives

This section shows how to get pointers into display memory. You need these pointers for rendering into windows and custom screens with the general-purpose Amiga graphics routines and for rendering borders, images and text into windows and screens with the Intuition routines. This section also has some cautionary advice about using rendering routines in Intuition displays. Unfortunately, this book does not have the space to provide a primer for using the graphics routines. To learn how to use them, you will need to refer to the *Amiga ROM Kernel Manual.*

You can use all of the Amiga graphics routines in your Intuition windows and custom screens. All of the routines require a pointer to some writable display area, either a RastPort, ViewPort, or View. Intuition creates a RastPort and ViewPort for each of your windows and custom screens. A RastPort defines some general parameters of a complete display and provides an area where you can safely write. A ViewPort specifies some portion of a RastPort.

You can obtain a pointer to any window or screen RastPort or ViewPort by using instructions like these:

POINTERS TO WINDOW RASTPORT AND VIEWPORT:

```
struct Window *MyWindow;
struct RastPort *MyRPort;
struct ViewPort *MyVPort;
struct View *BigView;

MyWindow = OpenWindow(...);
MyRPort = MyWindow->RPort;
MyVPort = ViewPortAddress(MyWindow);
BigView = ViewAddress();
```

POINTERS TO SCREEN RASTPORT AND VIEWPORT:

```
struct Screen *MyScreen;
struct RastPort *MyRPort;
struct ViewPort *MyVPort;
struct View *BigView;

MyScreen = OpenScreen(...);
MyRPort = &MyScreen->RastPort;
MyVPort = &MyScreen->ViewPort;
BigView = ViewAddress();
```

The Intuition function *ViewPortAddress()* returns the address of a window's ViewPort.

A View structure is a linked list of one or more ViewPorts. Intuition's View is a linked list of all the display structures that you use in your Intuition-based program. The function *ViewAddress()* returns the address of the Intuition View structure.

When you use graphics primitives to render directly into a window RastPort, and you allow the user to size or move the window, the underlying screen display is destroyed. A blank background is displayed in the areas uncovered when the screen is sized or moved. If this is a problem for your program, you can overcome it by opening windows that cannot be moved or sized.

If a graphics routine requires the allocation and initialization of other graphics mechanisms— TmpRas structure, GelsInfo, AreaFill buffers, UserCopperList or the like—you set these up as usual as described in the *Amiga ROM Kernel Manual.*

Chapter 10

MOUSE AND KEYBOARD

In the Intuition system, the mouse is the normal method of making selections. This chapter describes how users employ the mouse to interact with the system and your programs and how you can arrange for your program to use the mouse in other ways. It also describes the use of the keyboard as an alternate means of input.

About the Mouse

A mouse is a small, hand-held input device connected to the Amiga by a flexible cable. By rolling the mouse around on a smooth surface, the user can input horizontal and vertical position coordinates to the computer. The mouse also provides a pair of input keys, called *mouse buttons*, for the user to input further information to the computer.

Most of the things the user does with the mouse are meaningful to Intuition. Because of this, Intuition monitors mouse activity very closely. As the user moves the mouse, Intuition follows the motion by changing the position of the Intuition *pointer*. The Intuition pointer is an image (using hardware sprite zero) that can move around the entire video display, mimicking the user's movement of the mouse. The user can use the mouse and pointer to point at some object and then have some action performed on that object. Typically, users specify an action by manipulating either or both mouse buttons. Users can also position the mouse while the buttons are activated.

The basic mouse activities are shown in Table 10-1.

Table 10-1: Mouse Activities

ACTION	EXPLANATION
Pressing	Positioning the pointer while holding down a button. The action specified by the position of the pointer can continue to occur until the button is released, or alternatively may not occur at all until the button is released.
Clicking	Positioning the pointer and quickly pressing and releasing one of the mouse buttons.
Double-clicking	Positioning the pointer and pressing and releasing a mouse button twice.
Dragging	Positioning the pointer over some object, pressing a button, moving the mouse to a new location, and releasing the button.

The left mouse button is most often used for *selection*. The right mouse button is most often used for *information transfer*. The terms selection and information are intentionally left open to some interpretation, as it's impossible to imagine all the uses you'll find for the mouse buttons. The selection/information paradigm can be crafted to cover most interaction between the user and your program. You are encouraged, when designing mouse usage, to emphasize this model. It will help the user to understand and remember the elements of everyone's design.

When the user presses the left button, Intuition examines the state of the system and the position of the pointer. Intuition uses this information to decide whether or not the user is trying to select some object, operation, or option. For example, the user positions the pointer over a gadget and then presses the left button to select that gadget. Or the user may position the pointer over a window and press the select button to activate the window. If the user moves the mouse while holding down the select button, this sometimes means that the user wants to select everything that the pointer moves over while the button is still pressed.

The right mouse button is used to initiate and control information-gathering processes. Intuition uses this button most often for menu operations. Pressing the right button usually displays the active window's menu bar over the screen title bar. Moving the mouse while holding down the right button sometimes means that the user wishes to browse through all available information; for example, browsing through the menus. Double-clicking the right mouse button can bring up a special requester for extended exchange of information. This requester is called the double-menu requester, because of the double-click of the menu button required to reveal it, and because this requester is like a super menu where a complex exchange of information can take place. Because the requester is used for the transfer of information, it's appropriate that this mechanism is called up by using the right button.

Your program can receive mouse button and mouse movement events directly. If you are planning to handle mouse button events yourself, you should continue the selection/information model used by Intuition.

You can combine mouse button activations and mouse movement to create compound instructions. Here's an example of how Intuition combines multiple mouse events. While the right button is pressed to reveal the menu items of the active window, the user can press the left

button several times to select more than one option from the menus. Also, you can allow the user to move objects or select multiple objects by moving the mouse while holding down the buttons. As another example, consider the Workbench tool. There, to move an object, the user places the pointer within the object's icon and then presses the left button and moves the pointer. When the icon is in the desired location, the user releases the button.

Dragging can have different effects, depending on the object being dragged. To move a window to another area of the screen, the user positions the pointer within the window's drag gadget and drags the window to a new position. To change the size of a window, the user positions the pointer within the size gadget and drags the window to some smaller or larger size. In drag selection, the user can hold down both buttons while in menu mode and move the pointer across the menu display, making multiple selections with one stroke.

Mouse Messages

Mouse events are broadcast to your program via the IDCMP or the Console Device. See Chapter 9, "Input and Output Methods", for information on how to receive communications.

Simple mouse button activity not associated with any Intuition function will be reported as the event class MOUSEBUTTONS, with the codes SELECTDOWN, SELECTUP, MENUDOWN and MENUUP to specify changes in the state of the left and right buttons respectively. Mouse button activity over your gadgets is reported with a class of GADGETDOWN or GADGETUP, and the IAddress field (or EventAddress field of InputEvents) has the address of the selected gadget. Menu selections appear with a class of MENUPICK, with the menu number in the code field.

Your program receives mouse position changes in the event class MOUSEMOVE. The MouseX and MouseY position coordinates describe the position of the mouse relative to the upper-left corner of your window. These coordinates are always in the resolution of the screen you are using, and may represent of any pixel position in your screen, even though the hardware sprites can be positioned only on the even-numbered pixels of a high-resolution screen and on the even-numbered rows of an interlace screen.

For mouse movement reports as deltas (amount of change from the last position) instead of absolute positions, you can use the IDCMP flag, DELTAMOVE.

About the Keyboard

The keyboard is used mainly for entering data. However, there are several special ways to use the keyboard events as alternate methods for the user to enter commands. In particular, the Amiga keyboard has several special command keys. Each is uniquely identifiable when pressed along with one of the regular alphanumeric keys. The user can hold down one of these command keys and type an alphanumeric key at the same time. This generates a keyboard event that is recognizably different from a normal keystroke. These special keyboard events are known as

command-key sequences. Intuition responds to certain of the sequences. Your program can respond to them, too. When you receive a RAWKEY event through the IDCMP, you can tell if the user pressed any of the special command keys at the same time by examining the input message's Qualifier field for the special flags designating the special keys.

These special command keys (and their flags) are shown in Table 10-2.

Table 10-2: Special Command Keys

KEY	LABEL	EXPLANATION
control	CTRL	The associated Qualifier flag is the CONTROL flag.
alternate	ALT	Please note that there are two separate ALT keys, one on each side of the space bar. These can be treated distinctly. You can detect which one was pressed by examing the LALT and RALT commands for the Left ALT and Right ALT keys respectively
escape	ESC	When this key is struck, its keycode is entered into the input stream as an actual keystroke.
function	F1 to F10	Shortcut methods for entering command-key sequences starting with the ESC key.
AMIGA	Fancy A	Like the ALT keys, there are two Amiga keys, one on each side of the space bar. These are distinctly identifiable as well. The Left AMIGA key is recognized by the Qualifier flag LCOMMAND, and the Right AMIGA key by RCOMMAND.

Certain command-key sequences starting with one of the AMIGA keys have special meaning to Intuition. Most notably, these involve shortcuts and alternatives to using the mouse, as described in the following sections.

Using the Keyboard as an Alternate to the Mouse

All Intuition mouse activities can be emulated using the keyboard, by combining the Amiga command keys with other keystrokes.

The pointer can be moved by pressing down either AMIGA key along with one of the four cursor keys (the ones with the arrows). The longer these keys are held down, the faster the mouse will move. Also, you can hold down either SHIFT key to make the pointer leap greater

distances.

To emulate the left mouse button, users can press the left ALT key and the left AMIGA key simultaneously. To emulate the right mouse button, users can press the right ALT key and the right AMIGA key simultaneously. These key combinations permit users to make gadget selections and perform menu operations using the keyboard alone. This will be a boon for mouse-haters.

The following special shortcut functions are supported by Intuition:

o "Bring Workbench to the Front" (Left AMIGA and the "N" key)
o "Send Workbench to the Back" (Left AMIGA and the "M" key)

Note that these functions emulate left mouse button and mouse movement operations. Also note that Intuition always consumes these two command-key sequences for its own use. That is, it always detects these events and removes them from the input stream.

You can pair up menu items with command-key sequences to associate certain letters with specific menu item selects. This gives the user a shortcut method to select oft-used menu operations, such as UNDO, CUT, and PASTE. Whenever the user presses the Right AMIGA key with some alphanumeric key, the menu strip of the active window is scanned to see if there are any command-key sequences in the list that match the sequence entered by the user. If there is a match, Intuition translates the key combination into the appropriate menu item number and transmits the menu number to the application program. It looks to the application as if the user had selected a given menu item with the mouse. For more information on menu item selection, see Chapter 6, "Menus".

If Intuition sees a command key sequence that means nothing to it, the key sequence is broadcast to your program as usual. See Chapter 8, "Input and Output Methods", for how this works.

The final chapter of this book, "Style", contains a complete list of the suggested standard command key usage. We recommend that you take advantage of this standard so that the Amiga user can grow accustomed to a common set of functions across applications.

Chapter 11

OTHER FEATURES

The topics in this chapter have effects on the entire program or the entire Intuition display. This chapter:

o   introduces you to Intuition's easy way to allocate and free memory in an orderly fashion.

o   shows you how to get the user's Preferences settings or the default Preferences settings. Preferences is the program that allows the user to set numerous system-wide parameters.

o   describes the two functions that affect the entire Intuition display—one function remakes the ViewPort for each screen and the other reconstructs the entire display.

o   describes the functions for flashing the screen, and getting the current time values.

o   discusses using sprites in Intuition displays.

o   gives information about register allocation and variable names for assembly language programmers.

Easy Memory Allocation and Deallocation

Intuition has a pair of routines that make it easy for you to do easy and easily abortable memory allocations and deallocations. The routines are *AllocRemember()* and *FreeRemember()*. They use a data type called Remember.
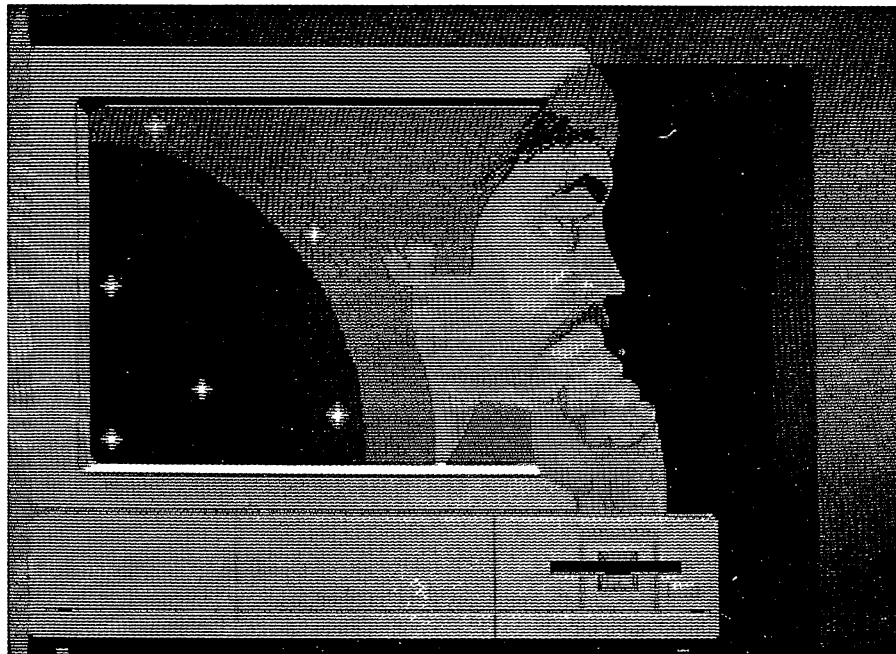
Figure 11-1: Intuition Remembering

INTUITION HELPS YOU REMEMBER

The *AllocRemember()* routine calls the Exec AllocMem() function for you, but also allocates a *link node* and uses it to save the parameters of the allocation into a master linked list for you. Then you can simply call *FreeRemember()* at a later time to deallocate all allocated memory without being required to remember the details of the memory you've allocated.

The *FreeRemember()* function gives you the option of freeing memory one of two ways:

    o   You can free both the memory blocks you've allocated and the link nodes that Intuition allocates, or

    o   After you've successfully allocated all the memory blocks you need, you can free up only the link nodes and keep the memory blocks for yourself.

These routines have two primary uses:

    o   The most general use of these routines is to do all of a program's memory allocations using *AllocRemember()*. The advantage of this is that a linked list of all your memory allocations is created for you, so that when you want to free up all the memory, a single call to *FreeRemember()* does the job for you.

    o   The other use is to do a series of memory allocations and abandon it in midstream easily, if you must. Let's say that you're doing a long series of allocations in a procedure (for example, the Intuition *OpenWindow()* procedure), and you detect some error

condition, like "out of memory". When aborting, you should free up any memory that you've already managed to allocate. These procedures allow you to free up that memory easily, without being required to keep track of how many allocations you've already done, the sizes of the allocations, and where the memory was allocated.

HOW TO REMEMBER

You create the "anchor" for the allocation master list by creating a variable that's a pointer to the data structure Remember, and initializing that pointer to NULL. This is called the *RememberKey*. Whenever you call *AllocRemember()*, the routine actually does two memory allocations, one for the memory you want and the other for a copy of a Remember structure. The Remember structure is filled in with data describing your memory allocation, and it's linked into the master list pointed to by your RememberKey. Then, to free up any memory that's been allocated, all you have to do is call *FreeRemember()* with your RememberKey.

See the *Amiga ROM Kernel Manual* for a description of the *AllocMem()* call and the values you should use for the Size and Flags variables.

THE REMEMBER STRUCTURE

Here's the Remember structure:

```
struct Remember
{
    struct Remember *NextRemember;
    ULONG RememberSize;
    UBYTE *Memory;
};
```

Here's what the Remember variables mean:

NextRemember
    This is the link to the next Remember node.

RememberSize
    This is the size of the memory remembered by this node.

Memory
    This is a pointer to the memory remembered by this node.

AN EXAMPLE OF REMEMBERING

```
struct Remember *RememberKey;
UBYTE *MemAPointer, *MemBPointer;

RememberKey = NULL;
MemAPointer = AllocRemember(&RememberKey, BUFSIZE, MEMF_CHIP);
MemBPointer = AllocRemember(&RememberKey, BUFSIZE, MEMF_CHIP);

/* Use the memory for various things ... */

/* and finally, give up the memory ... */
FreeRemember(&RememberKey, TRUE);
```

Preferences

Preferences is a program that lets the user see and change many system-wide parameters on the Amiga. Users can also edit the standard Intuition pointer image and colors. You have access to the Command Line Interpreter (CLI) through Preferences, by setting a flag that allows the CLI icon to be visible on the Workbench display. (See the AmigaDOS manuals for more information about the CLI.)

The user invokes Preferences to make settings and you can call *GetPrefs()* to find out what settings the user has made. In a system where the user does not use Preferences, you can call *GetDefPrefs()* to find out the Intuition default Preference settings. If you are using the IDCMP for input, you can set the IDCMP flag NEWPREFS. With this flag set, you will receive an IntuiMessage telling you that there is a new set of Preferences for you to examine. To get the new settings, you then call *GetPrefs()*.

Developers of printer driver programs should always call *GetPrefs()* just before every print job. The user may change to a different printer and run Preferences to modify the printer settings.

When Intuition is initialized (when the system is reset), you can call *GetDefPrefs()* to find the default Preferences settings that Intuition uses when it is first opened. Then, under AmigaDOS, Intuition is configured according to the set of Preferences that are saved on the startup disk.

Upon invoking the Preferences tool, the user is shown a screen full of gadgets and can change settings by selecting and playing with the gadgets. In some cases, a requester appears after the user selects a gadget. Figure 11-2 shows the main Preferences display.
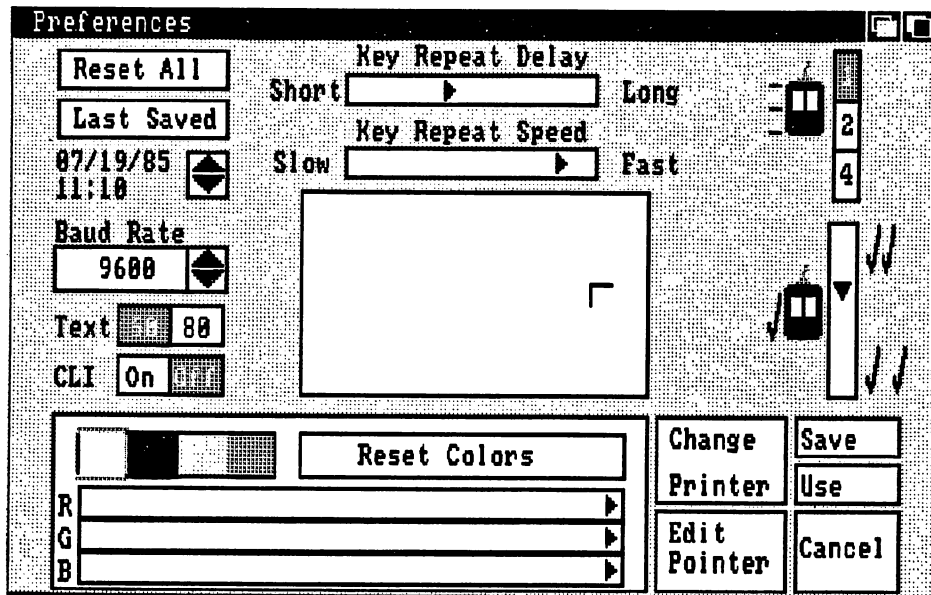
Figure 11-2: Preferences Display

One of the arguments to *GetPrefs()* and *GetDefPrefs()* is the size of the buffer you are supplying to receive the Preferences data. If you are interested only in the first few bits of data, you don't have to allocate a buffer large enough to hold the entire Preferences structure. For this reason, the most commonly used data has been grouped near the beginning of the structure.

Preferences allows the user to change the following:

o   Date and time of day.

o   Key repeat speed — the speed at which a key repeats when held down.

o   Key repeat delay — the amount of delay before the key begins repeating.

o   Mouse speed — how far the pointer moves when the user moves the mouse.

o   Double-click delay — maximum time allowed between the two clicks of a mouse double-click. For information about how to test for double-click timeout, see the description of the *DoubleClick()* function in Appendix A.

o   Text size — size of the default font characters. The user can choose 60-column mode (60 characters on a line in high resolution mode and 30 characters in low resolution mode) or 80-column mode (80 characters on a line in high resolution mode and 40 characters in low resolution mode). The first variable in the Preferences structure is *FontHeight*, which is the height of the characters in display lines. If this this is equal to the constant TOPAZ_EIGHTY, the user has chosen the 80-column version. If it is equal to TOPAZ_SIXTY, the user has chosen the 60-column version. The Preferences Display in Figure 11-2 shows TOPAZ_SIXTY.

o   CLI — allows access to the Command Line Interpreter for developers.

o   Display centering — allows the user to center the image on the video display.

o   Baud rate — the user can change the rate of data transmission to accommodate whatever device is attached to the serial connector.

o   Workbench colors — the user can change any of the four colors in the Workbench display by adjusting the amounts of red, green, and blue in each color.

o   Printer — the user can select from a number of printers supported by Amiga or can type in another printer name, depending upon which printers are supported by any application. The user can also indicate whether the printer is connected to the serial connector or the parallel connector.

o   Print characteristics — the user can select paper size, right and left margin, continuous feed or single sheets, draft or letter quality, pitch, and line spacing. If the user chooses the "Graphic Select" gadget, a requester appears from which the user can select shade (gray-scale printing), aspect (normal or sideways), positive or reverse image, and threshold (for black and white printing, which colors are printed as white and which as black).

The Preferences settings can be written to a Workbench disk so the user can save the settings for the next work session. The manual called *Introduction to Amiga* contains more information about Preferences from the user's standpoint.

PREFERENCES STRUCTURE

Here is the Preferences data structure:

```
struct Preferences
{
    BYTE FontHeight;
    UBYTE PrinterPort;
    USHORT BaudRate;
    struct timeval KeyRptSpeed, KeyRptDelay;
    struct timeval DoubleClick;
    USHORT PointerMatrix[POINTERSIZE];
    BYTE XOffset, YOffset;
    USHORT color17, color18, color19;
    USHORT PointerTicks;
    USHORT color0, color1, color2, color3;
    BYTE ViewXOffset, ViewYOffset;
    WORD ViewInitX, ViewInitY;
    BOOL EnableCLI;
    USHORT PrinterType;
    UBYTE PrinterFilename[FILENAME_SIZE];
    USHORT PrintPitch;
    USHORT PrintQuality;
    USHORT PrintSpacing;
    UWORD PrintLeftMargin, PrintRightMargin;
    USHORT PrintImage;
    USHORT PrintAspect;
    USHORT PrintShade;
    WORD PrintThreshold;
    USHORT PaperSize;
    UWORD PaperLength;
    USHORT PaperType;
};
```

The meanings of the fields in the Preferences structure are as follows:

FontHeight
> This variable will contain one of two constants: TOPAZ_SIXTY or TOPAZ_EIGHTY. These are the font heights required to cause the default Topaz font to be rendered in either 60- or 80-column mode wherever the default font is requested.

PrinterPort
> This is set to either PARALLEL_PRINTER or SERIAL_PRINTER, to describe which type of printer is attached to the printer port.

BaudRate
> This can be set to any of these default baud rates. See Appendix B for a complete list of the definitions you might find in this variable.

KeyRptSpeed, KeyRptDelay
> These are timeval structures, which have two components, seconds and microseconds. *KeyRptDelay* describes how long the system hesitates before the Input Device starts

repeating the keys. *KeyRptSpeed* describes how the time between repeats of the key.

DoubleClick

> This is a timeval structure, which describes the maximum time allowable between clicks of the mouse button for the operation to be considered a double-click operation. See Chapter 10, "Keyboard and Mouse", for details about double-clicking.

PointerMatrix[POINTERSIZE]

> This contains the sprite data for the Intuition pointer.

XOffset, YOffset

> This describes the offsets from the upper-left corner of the pointer image to the pointer's active spot.

color17, color18, color19

> These are the colors of the Intuition pointer.

PointerTicks

> This describes how many ticks are required for the mouse to move one increment. This should always be a power of two. The Preferences tool allows it to be set to 1, 2, or 4. Setting it to greater than 4 is not a great thing to do. For instance, if *PointerTicks* was set to 32768, then to move the pointer from the bottom to the top of the screen the user would have to move the mouse more than a mile.

color0, color1, color2, color3

> These are the Workbench colors.

ViewXOffset, ViewYOffset

> These describe the offset of the View from its initial startup position. This configurable offset allows the user to position the display on his monitor.

ViewInitX, ViewInitY

> These have copies of the initial View values, as created by the graphics library.

EnableCLI

> This Boolean value describes whether or not the Workbench should display the CLI icon when the CLI tool is available.

PrinterType

> These are the definitions of the available printer types. See Appendix B for a complete list of the definitions you might find in this variable.

PrinterFilename[FILENAME_SIZE]

> The default name for the disk-based printer configuration file is kept in this buffer.

PrintPitch, PrintQuality, PrintSpacing

> These describe the pitch, print quality, and page spacing for printer drivers.

PrintLeftMargin, PrintRightMargin
> The character spacing of the print margins are described by these variables.

PrintImage, PrintAspect, PrintShade
> The values of these variables tell printer drivers about the desired type of page imagery.

PrintThreshold
> For simple black/white printer dumps, this describes the intensity threshold required to trigger a print of a pixel.

PaperSize, PaperLength, PaperType
> These describe the user's choice of printer paper.

PREFERENCES FUNCTIONS

You can use the following functions to check the current Preferences settings.

> *GetPrefs(PrefBuffer, Size);* Gets a copy of the current Preferences data.
> > *PrefBuffer* - pointer to the memory buffer to receive the Preferences data
> > *Size* - number of bytes to copy to the buffer

> *GetDefPrefs(PrefBuffer, Size);* Gets a copy of the default Preferences data.
> > *PrefBuffer* - pointer to the memory buffer to receive the Preferences data
> > *Size* - number of bytes to copy to the buffer

Remaking the ViewPorts

This section is for advanced programmers who are interested in controlling their custom screens directly and want to control the entire Intuition display.

There are two functions that operate on the entire display. These are *RethinkDisplay()* and *RemakeDisplay()*. The *MakeScreen()* function works only with the Copper lists of your custom screen.

RethinkDisplay()

RethinkDisplay() reworks Intuition's internal state data, rethinks the relationship of all of the screen ViewPorts to each other and reconstructs the entire Intuition display by calling the graphics primitives *MrgCop()* and *LoadView()*. This includes all the screens in the display, not just the ones controlled by your program. It is especially handy if you are creating custom screens and want to make up your own lists of Copper instructions for handling the display. For more information about the Copper, see the *Amiga ROM Kernel Manual* and *Amiga*

RethinkDisplay() makes calls to the graphics primitives *MrgCop()* and *LoadView()*, which causes the display of Intuition's screens to be reconstructed. *MrgCop()* merges all the various Copper instructions for different ViewPorts of the display into a single instruction stream. This creates a complete set of instructions for each *display field* (complete scanning of the video beam from top to bottom of the video display). *LoadView()* uses this merged Copper instruction list to create the display. Before calling *RethinkDisplay()*, you may wish to call *MakeScreen()* to create the Copper instruction list for your own custom screens.

Note that *RethinkDisplay()* can take several milliseconds to run, and it locks out all other tasks while it runs. This can seriously degrade the performance of the multi-tasking Executive, so don't use this routine lightly.

RemakeDisplay()

The function *RemakeDisplay* reconstructs the entire Intuition display. It calls *MakeScreen()* for every screen in the system and then calls *RethinkDisplay()*. As with *RethinkDisplay()*, *RemakeDisplay()* can take several milliseconds to run, and it locks out all other tasks while it runs. This can seriously degrade the performance of the multi-tasking Executive, so do not use this routine lightly.

MakeScreen()

To remake the Copper lists of your custom screen, call *MakeScreen()*. Then only difference between *MakeScreen()* and the graphics library routine *MakeVPort()* is that Intuition synchronizes your call to *MakeVPort()* with any that it needs to make.

Current Time Values

The function *CurrentTime()* gets the current time values. To use this function, you first declare the variables *Seconds* and *Micros*. Then, when you call the function, the current time is copied into the argument pointers. The synopsis of this function is:

```
ULONG Seconds, Micros;
CurrentTime(&Seconds,&Micros);
```

Flashing the Display

Because the Amiga has no internal bell or beeper, this function is supplied to notify the user of some event that is not serious enough to require a requester. For example, Intuition uses this function when the user types an invalid character into an integer gadget. This function flashes the background color of the screen. If the argument to the function is NULL, every screen in the display is flashed. The synopsis of this function is:

DisplayBeep(Screen);

Using Sprites in Intuition Windows and Screens

Sprites do not behave well under Intuition, except in somewhat limited cases. The hardware and graphics library sprite systems manage sprites independent of the Intuition display. In particular:

o   sprites can't be "attached" to any particular screen. Instead, they always appear in front of every screen.

o   when a screen is moved, the sprites do not automatically move with it. The sprites move to their correct locations only when the appropriate function is called (either *DrawGList()* or *MoveSprite()*).

Hardware sprites are of limited use under the Intuition paradigm. They travel out of windows and out of screens, unlike all other Intuition mechanisms (except the Intuition pointer, which is meant to be global).

Assembly Language Conventions

In all Intuition routines, the arguments always follow the same order: addresses first, data second. The registers are allocated in ascending order from register 0. Always. So you can look at any routine, start from register A0 if the routine's arguments start with an address, and start from D0 when the routine's arguments become data values. As an added mnemonic, even the register names are in alphabetical order—A0 precedes D0. Good enough?

Unfortunately for assembly programmers, many of you will have to use assemblers that don't give you macros to declare and reference structure elements. If this is the case for you, then

you should use the include file called "intuition.i", where every Intuition structure variable has a unique name, found in assembler format.

Chapter 12

STYLE

This chapter describes some important aspects of Intuition style. If you adhere to these style notes, you will help to ensure that Intuition applications present a consistent interface to the user. Try to exercise all of the suggestions in this chapter.

Menu Style

Always make sure that you use *OffMenu()* when a item becomes meaningless or non-functional. Don't ever let the user select something and then have the application do nothing in response. Always take away the user's ability to select that item.

The pens you set when you open a window are used to render the menu bar and the items. If you are opening multiple windows, you might consider color-coding the window frames and menus.

PROJECT MENUS

If you are going to be allowing the user to select which project to work with, we suggest that you create a "Project" menu. For consistency, we suggest that everyone create their menu strips with the Project menu as the leftmost menu. It should contain the items shown in Table 12-1. If possible, the items should be in the order shown.

Table 12-1: Project Menus

MENU ITEM	FUNCTION
NEW	Creates a project
OPEN	Gets back a project previously saved
SAVE	Saves the current project to the disk
SAVE AS	Saves the current project using a different name
PRINT	Prints the entire project
PRINT AS	Prints part of a project or selects other than the default printer settings
QUIT	Stops the program (if the project was modified, ask if the user wants to save the work)

EDIT MENUS

If your application can perform edit-like functions, we suggest that you create an "Edit" menu, which should appear to the right of the Project menu. It should contain the items shown in Table 12-2. If possible, the items should be in the order shown in the table.

Table 12-2: Edit Menus

MENU ITEM	FUNCTION
UNDO	Undoes the previous operation (if possible. If not, disable this option!)
CUT	Removes the selected portion of the project and puts it in the ClipBoard
COPY	Puts a copy of the selected bit of the project in the ClipBoard
PASTE	Puts a copy of the ClipBoard into the project
ERASE	Removes the selected bit without putting it into the ClipBoard

Gadget Style

When creating a list of gadgets, in a requester or perhaps a window, be sure to design bolder, more eye-catching imagery for the obvious or safe choice. For example, note how the CANCEL choice is highlighted in Figure 12-1.

Overlapping the select boxes of gadgets is in general not a good thing to do. This is especially true when it's not obvious to users which gadget they are selecting. Unless you're very careful, all sorts of weird things can happen and the gadgets will behave in unusual ways.

As with menus, use *OffGadget()* to remove a gadget when it becomes meaningless or nonfunctional.
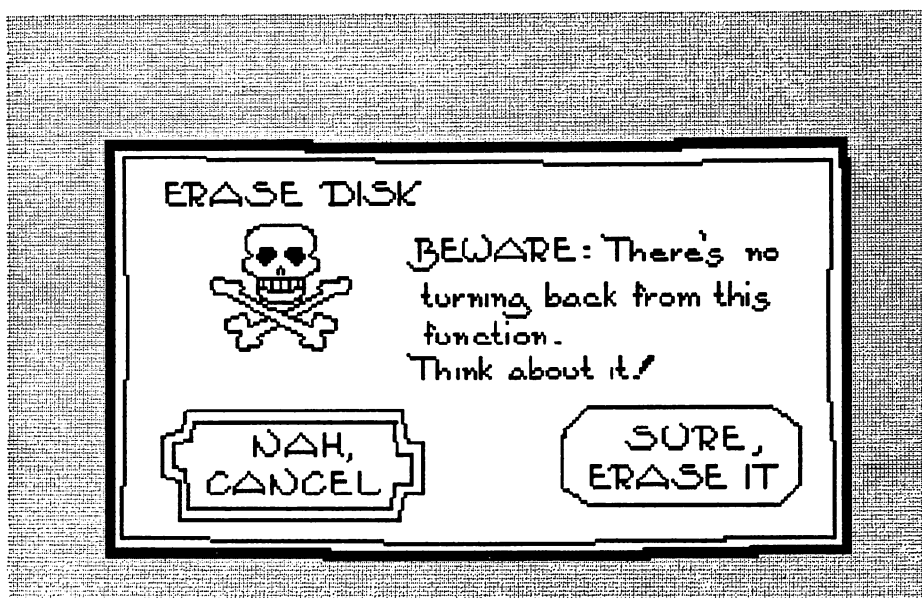
Figure 12-1: The Dreaded Erase-Disk Requester

Requester Style

This is easily the *most important* rule about requesters: Always make sure that there's a safe way to exit from any requester. As in Figure 12-2, notice that the dire "ERASE DISK" requester can be cancelled and, in fact, the "run away" option is rendered in bolder imagery. If, for instance, the user accidentally selected the ERASE DISK option from a menu, the CANCEL option saves the day. This is an extremely important note. We can't emphasize this point

strongly enough.

When you design a requester with your own BitMap imagery, make sure that the imagery works well with the select boxes of the gadget list that you supply.

Command Key Style

Treat the AMIGA keys like SHIFT keys. To enter a shortcut, users should be able to hold down the AMIGA key with the little finger of one hand, and press one of the keys they'd normally press with the other hand. This will help touch typists as well as prevent that clumsy feeling that we all experience.

Table 12-3 shows our recommendations for standard selection shortcuts (using the left AMIGA key to emulate usage of the left button of the mouse):

Table 12-3: Selection Shortcuts

Press Left AMIGA with:	*To:*
I	"Select a small piece to the right of the cursor" like the next word
O	"Select a bigger piece to the right of the cursor" like the next sentence
P	"Select an even bigger piece to the right of the cursor" like the next paragraph
J	"Select a small piece to the left of the cursor" like the previous word
K	"Select a bigger piece to the left of the cursor" like the previous sentence
L	"Select an even bigger piece to the left of the cursor" like the previous paragraph
N	Bring the Workbench to the front (this is automatically trapped by Intuition)
M	Send the Workbench to the back (this is automatically trapped by Intuition)

Table 12-4 shows our recommendations for standard information (menu) shortcuts (using the right AMIGA key to emulate usage of the right button of the mouse):

Table 12-4: Information (Menu) Shortcuts

Press Right AMIGA
with: *To:*

X	Cut
C	Copy
P	Paste
I	Change font type to italic
B	Change font type to bold
U	Change font mode to underline
P	Reset font characteristics to plain defaults
Q	Undo (cancel)
S	Save

Miscellaneous Style Notes

Remember, exiting programs should always make a call to *OpenWorkBench()*, even if you didn't call *CloseWorkBench()*. We want Workbench to be open as much as possible. If Workbench was closed and your departure has freed up enough memory for Workbench to reopen, we want it to happen. *OpenWorkBench()* won't necessarily work (if there's no memory for the display, it won't open). But if everyone calls *OpenWorkBench()* then Workbench will open if it can. By using this mechanism, you can help give the user a consistent environment. Intuition always checks to see if Workbench *must* open whenever any screen is closed.

As much as possible, allow the user to configure the parameters of your program. For instance, if you've opened a custom screen, let the user change the colors. If your program makes sound, give the user the ability to adjust the tone and volume. Don't make the configuration a requirement, however, and always give the user an avenue for restoring the defaults.

The Intuition default pointer is designed with the light source coming from the top-right. If you design your own pointer, consider mimicking this. Most importantly, here are the color assignments used for the Intuition pointer sprite data:

o   Color 0 is transparent

o   Color 1 of the sprite (hardware color register 17) is the color with medium intensity

o   Color 2 of the sprite (hardware color register 18) is low intensity

o   Color 3 of the sprite (hardware color register 19) is high intensity

Your pointer should be framed by either Color 1 or Color 3.

Since the Intuition pointer is always hardware sprite zero, you can set the colors of the pointer by calling the SetRGB() function on the ViewPort of any screen. An example of this is:

```
struct Screen *MyScreen;

SetRGB(&MyScreen->ViewPort, 17, Red17, Green17, Blue17);
SetRGB(&MyScreen->ViewPort, 18, Red18, Green18, Blue18);
SetRGB(&MyScreen->ViewPort, 19, Red19, Green19, Blue19);
```

A Final Note on Style

Design beautiful Gadgets, Menus, Requesters. Think simplicity and elegance. And always remember the 4th grader, the sophisticated user, and the poor soul who's terrified of breaking the machine.

Dare to be gorgeous and unique! But don't ever be cryptic or otherwise unfathomable. Make it unforgettably great.

Appendix A

INTUITION FUNCTION CALLS

In this appendix, all of the Intuition functions are presented in alphabetical order. The description of each function follows the format shown below:

NAME
   The name of the function and a one-line description of what it does.

SYNOPSIS
   The correct form of the function call.

FUNCTION
   Everything the function does.

RESULT
   The results, if any, returned by the function.

BUGS
   All known bugs, limitations, and deficiencies.

SEE ALSO
   References to related functions in this and other books, references to the text of this book.

APPENDIX A

TABLE OF CONTENTS

ModifyIDCMP	Modifies the state of the Window's IDCMP
ModifyProp	Modifies the current parameters of a Proportional Gadget
MoveScreen	Attempts to move the Screen by the delta amounts
MoveWindow	Asks Intuition to move a Window
OffGadget	Disables the specified Gadget
OffMenu	Disables the given menu or menu item
OnGadget	Enables the specified Gadget
OnMenu	Enables the given menu or menu item
OpenScreen	Opens an Intuition Screen
OpenWorkBench	Opens the WorkBench Screen
PrintIText	Prints the text according to the IntuiText argument
RefreshGadgets	Refreshes (redraws) the Gadget display
RemakeDisplay	Remakes the entire Intuition display
RemoveGadget	Removes a Gadget from a Window or a Screen
ReportMouse	Tells Intuition whether or not to report mouse movement
Request	Activates a Requester
RethinkDisplay	The grand manipulator of the entire Intuition display
ScreenToBack	Sends the specified Screen to the back of the display
ScreenToFront	Brings the specified Screen to the front of the display
SetDMRequest	Sets the DMRequest of the Window
SetMenuStrip	Attaches the Menu strip to the Window
SetPointer	Sets a Window with its own Pointer
SetWindowTitles	Sets the Window's titles for both Window and Screen
ShowTitle	Sets the Screen title bar display mode
SizeWindow	Asks Intuition to size a Window
ViewAddress	Returns the address of the Intuition View structure
ViewPortAddress	Returns the address of a Window's ViewPort structure
WBenchToBack	Sends the WorkBench Screen in back of all Screens
WBenchToFront	Brings the WorkBench Screen in front of all Screens
WindowLimits	Sets the minimum and maximum limits of the Window
WindowToBack	Asks Intuition to send this Window to the back
WindowToFront	Asks Intuition to bring this Window to the front

NAME
>   AddGadget -- add a Gadget to the Gadget list of the Window or Screen

SYNOPSIS
>   SHORT AddGadget(Pointer, Gadget, Position);

FUNCTION
>   Adds the specified Gadget to the Gadget list of the given Window or Screen, linked in at
>   the position in the list specified by the Position argument (that is, if Pos === 0, the
>   Gadget will be inserted at the head of the list, and if Position === 1 then the Gadget
>   will be inserted after the first Gadget and before the second). If the Position you specify
>   is greater than the number of Gadgets in the list, your Gadget will be added to the end
>   of the list. The SCRGADGET Flag of the Gadget specifies whether the Pointer argu-
>   ment points to a Window (SCRGADGET not set) or a Screen (SCRGADGET is set).
>   This procedure returns the position at which your Gadget was added.
>
>   NOTE: A relatively safe way to add the Gadget to the end of the list is to specify a Po-
>   sition of -1. That way, only the 65536th (and multiples of it) will be inserted at the
>   wrong position. The return value of the procedure will tell you where it was actually in-
>   serted.
>
>   NOTE: The System Window and Screen Gadgets are initially added to the front of the
>   Gadget List. The reason for this is: if you position your own Gadgets in some way that
>   interferes with the graphical representation of the system Gadgets, the system's ones will
>   be "hit" first by User. If you then start adding Gadgets to the front of the list, you will
>   disturb this plan, so beware. On the other hand, if you don't violate the design rule of
>   never overlapping your Gadgets, there's no problem.

INPUTS
>   Pointer =
>>   pointer to the Window or Screen to get your Gadget
>
>   Gadget =
>>   pointer to the new Gadget
>
>   Position =
>>   integer position in the list for the new Gadget (starting from zero as the first po-
>>   sition in the list)

RESULT
>   Returns the position of where the Gadget was actually added.

BUGS
>   None

SEE ALSO
>   RemoveGadget()

NAME
> AllocRemember -- AllocMem and create a link node to make FreeMem easy

SYNOPSIS
> AllocRemember(RememberKey, Size, Flags);

FUNCTION
> This routine calls the EXEC AllocMem() function for you, but also links the parameters of the allocation into a master list, so that you can simply call the Intuition routine FreeRemember() at a later time to deallocate all allocated memory without being required to remember the details of the memory you've allocated.
>
> This routine will have two primary uses:
> - Let's say that you're doing a long series of allocations in a procedure (such as the Intuition OpenWindow() procedure). If any one of the allocations fails for lack of memory, you need to abort the procedure. Abandoning ship correctly involves freeing up what memory you've already allocated. This procedure allows you to free up that memory easily, without being required to keep track of how many allocations you already done, what the sizes of the allocations were, where the memory was allocated.
> - Also, in the more general case, you may do all of the allocations in your entire program using this routine. Then, when your program is exiting, you can free it all up at once with a simple call to FreeRemember().
>
> You create the "anchor" for the allocation master list by creating a variable that's a pointer to struct Remember, and initializing that pointer to NULL. This is called the RememberKey. Whenever you call AllocRemember(), the routine actually does two memory allocations, one for the memory you want and the other for a copy of a Remember structure. The Remember structure is filled in with data describing your memory allocation, and it's linked into the master list pointed to by your RememberKey. Then, to free up any memory that's been allocated, all you have to do is call FreeRemember() with your RememberKey.
>
> Please read the FreeRemember() header too. As you will see, you can select to either free just the link nodes and keep all the allocated memory for yourself, or you can elect to free both the nodes and your memory buffers.
>
> See the "Amiga ROM Kernel Manual" for a description of the AllocMem() call and the values you should use for the Size and Flags variables.

INPUTS
> RememberKey =
>> the address of a pointer to struct Remember. Before the very first call to AllocRemember, initialize this pointer to NULL. For instance:
>>
>>> struct Remember *RememberKey;
>>> RememberKey = NULL;
>>> AllocRemember(&RememberKey, BUFSIZE, MEMF_CHIP);
>>> FreeRemember(&RememberKey, TRUE);

Size =
>   the size in bytes of the memory allocation. Please refer to the EXEC AllocMem()
>   function in the "Amiga ROM Kernel Manual" for details.

Flags =
>   the specifications for the memory allocation. Please refer to the EXEC Alloc-
>   Mem() function in the "Amiga ROM Kernel Manual" for details.

RESULT
If the memory allocation is successful, this routine returns the byte address of your re-
quested memory block. Also, the node to your block will be linked into the list pointed
to by your RememberKey variable. If the allocation fails, this routine returns NULL
and the list pointed to by RememberKey, if any, will be undisturbed.

BUGS
None

SEE ALSO
FreeRemember()
The EXEC AllocMem() function

NAME

AutoRequest -- Automatically build and get response from a Requester

SYNOPSIS

AutoRequest(Window, BodyText, PositiveText, NegativeText,
        PositiveFlags, NegativeFlags, Width, Height);

FUNCTION

This procedure automatically builds a Requester for you and then waits for a response from the user or the system to satisfy your request. If the response is Positive, this procedure returns TRUE. If the response is negative, this procedure returns FALSE.

This procedure first preserves the state of the IDCMP values of the Window argument. Then it creates an IDCMPFlag specification by merging together your PositiveFlags, NegativeFlags, and the IDCMP class GADGETUP. You may choose to specify no flags for either the PositiveFlags or NegativeFlags arguments.

The IntuiText arguments, and the Width and Height values, are passed directly to the BuildSysRequest() procedure along with your Window pointer and the IDCMP flags. Please refer to BuildSysRequest() for a description of the IntuiText that you are expected to supply when calling this routine. It's an important but long-winded description that need not be duplicated here.

If the BuildSysRequest() procedure does not return a pointer to a Window, it will return TRUE or FALSE (not valid structure pointers) instead, and these BOOL values will be returned to you immediately.

On the other hand, if a valid Window pointer is returned, that Window will have had its IDCMP Ports and flags initialized according to your specifications. AutoRequest() then waits for an IDCMP message on the UserPort, which message will satisfy one of three requirements:

- either the message is of a class that matches one of your PositiveFlags arguments (if you've supplied any), in which case this routine returns TRUE. Or

- the message class matches one of your NegativeFlags arguments (if you've supplied any), in which case this routine returns FALSE. Or

- the only other possibility is that the IDCMP message is of class GADGETUP, which means that one of the two Gadgets, as specified by the PositiveText and NegativeText arguments, was selected by the user. If the TRUE Gadget was selected, TRUE is returned. If the FALSE Gadget was selected, FALSE is returned.

When the dust has settled, this routine calls FreeSysRequest() if necessary to clean up the Requester and any other allocated memory.

INPUTS

Window =
        pointer to a Window structure

BodyText =

        pointer to an IntuiText structure

PositiveText =
        pointer to an IntuiText structure

NegativeText =
        pointer to an IntuiText structure

PositiveFlags =
        flag for the IDCMP

NegativeFlags =
        flags for the IDCMP

Width, Height =
        the sizes required for the rendering of the Requester

RESULT
    The return value is either TRUE or FALSE. See the text above for a complete description of the chain of events that might lead to either of these values being returned.

BUGS
    None

SEE ALSO
    BuildSysRequest()

NAME
>   BeginRefresh -- Sets up a Window for optimized refreshing

SYNOPSIS
>   BeginRefresh(Window);

FUNCTION
>   This routine sets up your Window for optimized refreshing. It sets Intuition internal
>   states and then sets up the layer underlying your Window for a call to the layer library.
>   There, the "clip rectangles" of the layer are reorganized in a fashion where any rendering
>   performed in your Window (until you call to EndRefresh()) will occur only in the regions
>   which need to be refreshed. The phrase "clip rectangles" refers to the division of your
>   Window into visible and concealed rectangles. For more information about clipping rec-
>   tangles and the layer library, refer to the "Amiga ROM Kernel Manual".
>
>   For instance, if you have a SIMPLE_REFRESH Window which is partially concealed
>   and the user brings it to the front, you will receive a message asking you to refresh your
>   display. If you call BeginRefresh() before doing any of the rendering, then the layer that
>   underlies your Window will be arranged such that the only rendering that will actually
>   take place will be that which goes to the newly-revealed areas. This is very
>   performance-efficient.
>
>   After you have performed your refresh of the display, you should call EndRefresh() to
>   reset the state of the layer and the Window. Then you may proceed with rendering to
>   the Window as usual.
>
>   You learn that your Window needs refreshing by receiving either a message of class RE-
>   FRESHWINDOW   through   the   IDCMP,   or   an   input   event   of   class
>   IECLASS_REFRESHWINDOW through the Console Device. Whenever you are told
>   that your Window needs refreshing, you should call BeginRefresh() and EndRefresh() to
>   clear the refresh-needed state, even if you don't plan on doing any rendering.

INPUTS
>   Window = pointer to the Window structure which needs refreshing

RESULT
>   None

BUGS
>   None

SEE ALSO
>   EndRefresh()
>   The "Windows" chapter in this book

NAME
BuildSysRequest -- Build and display a system Requester

SYNOPSIS
BuildSysRequest(Window, BodyText, PositiveText, NegativeText,
    IDCMPFlags, Width, Height);

FUNCTION
This procedure builds a Requester based on the supplied information. If all goes well and the Requester is constructed, this procedure returns a pointer to the Window in which the Requester appears. That Window will have the IDCMP UserPort and WindowPort initialized to reflect the flags found in the IDCMPFlags argument. You may then Wait() on those ports to detect the user's response to your Requester, which response may include either selecting one of the Gadgets or causing some other event to be noticed by Intuition (like DISKINSERTED, for instance). After the Requester is satisfied, you should call the FreeSysRequest() procedure to remove the Requester and free up any allocated memory.

If it isn't possible to construct the Requester for any reason, this procedure will instead use the text arguments to construct a text string for a call to the DisplayAlert() procedure, and then will return either a TRUE or FALSE depending on whether DisplayAlert() returned a FALSE or TRUE respectively.

If the Window argument you supply is equal to NULL, a new Window will be created for you in the Workbench Screen. If you want the Requester created by this routine to be bound to a particular Window, you should not supply a Window argument of NULL.

The text arguments are used to construct the display. They are pointers to instances of the struct IntuiText.

The BodyText argument should be used to describe the nature of the Requester. As usual with IntuiText data, you may link several lines of text together, and the text may be placed in various locations in the Requester. This IntuiText pointer will be stored in the ReqText variable of the new Requester.

The PositiveText argument describes the text that you want associated with the user choice of "Yes. TRUE. Retry. Good." If the Requester is successfully opened, this text will be rendered in a Gadget in the lower-left of the Requester, which Gadget will have the GadgetID field set to TRUE. If the Requester cannot be opened and the DisplayAlert() mechanism is used, this text will be rendered in the lower-left corner of the Alert display with additional text specifying that the left mouse button will select this choice. This pointer can be set to NULL, which specifies that there is no TRUE choice that can be made.

The NegativeText argument describes the text that you want associated with the user choice of "No. FALSE. Cancel. Bad." If the Requester is successfully opened, this text will be rendered in a Gadget in the lower-right of the Requester, which Gadget will have the GadgetID field set to FALSE. If the Requester cannot be opened and the DisplayAlert() mechanism is used, this text will be rendered in the lower-right corner of the Alert display with additional text specifying that the right mouse button will select this

choice. This pointer cannot be set to NULL. There must always be a way for the user to cancel this Requester.

The Positive and Negative Gadgets created by this routine have the following features:
- BOOLGADGET
- RELVERIFY
- REQGADGET
- TOGGLESELECT

When defining the text for your Gadgets, you may find it convenient to use the special defines used by Intuition for the construction of the Gadgets. These include defines like AUTODRAWMODE, AUTOLEFTEDGE, AUTOTOPEDGE and AUTOFRONTPEN. You can find these in your local intuition.h (or intuition.i) file.

The Width and Height values describe the size of the Requester. All of your BodyText must fit within the Width and Height of your Requester. The Gadgets will be created to conform to your sizes.

VERY IMPORTANT NOTE: for the preliminary release of this procedure, a new Window is opened in the same Screen as the one containing your Window. However, with a forthcoming update of Intuition, this will change such that the Requester will be opened in the Window supplied as an argument to this routine, if possible. The primary implication of this will be that the IDCMPFlags and Ports will be disturbed by a call to this routine. To assure upward-compatibility, it's your responsibility to make sure that the Ports and IDCMPFlags of the Window passed to the routine are protected before the call.

INPUTS
Window =
    pointer to a Window structure

BodyText =
    pointer to an IntuiText structure

PositiveText =
    pointer to an IntuiText structure

NegativeText =
    pointer to an IntuiText structure

IDCMPFlags =
    the IDCMP flags you want used for the initialization of the
          IDCMP of the Window containing this Requester

Width, Height =
    the size required to render your Requester

RESULT
If the Requester was successfully rendered in a Window, the value returned by this procedure is a pointer to the Window in which the Requester was rendered. If, however, the Requester cannot be rendered in the Window, this routine will have called Display-Alert() before returning and will pass back TRUE if the user pressed the left mouse but-

ton and FALSE if the user pressed the right mouse button.

BUGS

This procedure currently opens a Window as wide as the Screen in which it was rendered, and then opens the Requester within that Window. Also, if DisplayAlert() is called, the PositiveText and NegativeText are not rendered in the lower corners of the Alert.

SEE ALSO

FreeSysRequest()
DisplayAlert()
ModifyIDCMP()
The Executive's Wait() instruction
AutoRequest()

NAME

ClearDMRequest -- clears the DMRequest of the Window

SYNOPSIS

ClearDMRequest(Window);

FUNCTION

Attempts to clear the DMRequester from the specified window. The DMRequester is the special Requester that you attach to the double-click of the menu button which the user can then bring up on demand. This routine WILL NOT clear the DMRequester if it's active (in use by the user). After having called SetDMRequest(), if you want to change the DMRequester, the correct way to start is by calling ClearDMRequest() until it returns a value of TRUE; then you can call SetDMRequest() with the new DMRequester.

INPUTS

Window =
> pointer to the window from which the DMRequest is to be cleared

RESULT

If the DMRequest was not currently in use, zeroes out the DMRequest   pointer in the Window and returns TRUE.
If the DMRequest was currently in use, doesn't change the pointer   and returns FALSE.

BUGS

None

SEE ALSO

SetDMRequest()
Request()

NAME
    ClearMenuStrip -- Clears the Menu strip from the Window

SYNOPSIS
    ClearMenuStrip(Window);

FUNCTION
    Clears the menu strip from the Window.

INPUTS
    Window = pointer to a Window structure

RESULT
    None

BUGS
    None

SEE ALSO
    SetMenuStrip()

NAME
>   ClearPointer -- clears the Pointer definition from a Window

SYNOPSIS
>   ClearPointer(Window);

FUNCTION
>   Clears the Window of its own definition of the Intuition pointer. After calling Clear-
>   Pointer(), every time this Window is the active one the default Intuition pointer will be
>   the pointer displayed to the user. If your Window is the active one when this routine is
>   called, the change will take place immediately.

INPUTS
>   Window = pointer to the Window to be cleared of its Pointer definition

RESULT
>   None

BUGS
>   None

SEE ALSO
>   SetPointer()

NAME
>   CloseScreen -- Closes an Intuition Screen

SYNOPSIS
>   CloseScreen(Screen);

FUNCTION
>   Unlinks the Screen, unlinks the ViewPort, deallocates everything. Doesn't care whether
>   or not there are still any Windows attached to the Screen. Doesn't try to close any at-
>   tached Windows; in fact, ignores them altogether. If this is the last Screen to go, at-
>   tempts to reopen WorkBench

INPUTS
>   Screen = pointer to the Screen to be deallocated

RESULT
>   None

BUGS
>   Don't think so

SEE ALSO
>   OpenScreen

NAME
> CloseWindow -- Closes an Intuition Window

SYNOPSIS
> CloseWindow(Window);

FUNCTION
> Closes an Intuition Window. Unlinks it from the system, unallocates its memory, and if its Screen is a system one that would be empty without the Window, closes the system Screen too
>
> A Grim, Foreboding Note: if you are ever rude enough to CloseWindow() on a Window that has an IDCMP without first having Reply()'d to all of my messages to the IDCMP port, Intuition in turn will so rude as to reclaim and deallocate its messages without waiting for your permission.
>
> Another grim note: if you have added a Menu strip to this Window (via a call to Set-MenuStrip()) you must be sure to remove that Menu strip (via a call to ClearMenuStrip()) before closing your Window. CloseWindow() doesn't check whether or not the menus of your Window are currently being used when the Window is closed. If this in fact happens to be the case, then as soon as the user releases the Menu button the system will crash with pyrotechnics that are usually quite lovely.

INPUTS
> Window = a pointer to a Window structure

RESULT
> None

BUGS
> Don't think so. What do you think?

SEE ALSO
> OpenWindow(), CloseScreen()

NAME
> CloseWorkBench -- Closes the WorkBench Screen

SYNOPSIS
> BOOL CloseWorkBench();

FUNCTION
> This routine attempts to close the WorkBench. The actions taken are:
> - Test whether or not any applications have opened Windows on the
> - WorkBench, and return FALSE if so. Otherwise ...
> - Clean up all special buffers
> - Close the WorkBench Screen
> - Make the WorkBench program mostly inactive (it will still monitor disk activity)
> - Return TRUE

INPUTS
> None

RESULT
> TRUE if the WorkBench Screen closed successfully.
> FALSE if anything went wrong and the WorkBench Screen is still out there.

BUGS
> None

SEE ALSO
> None

NAME

CurrentTime -- Get the current time values

SYNOPSIS

ULONG Seconds, Micros;
CurrentTime(&Seconds, &Micros);

FUNCTION

Puts copies of the current time into the supplied argument pointers.

This time value is not extremely accurate, nor is it of a very fine resolution. This time will be updated no more than sixty times a a second, and will typically be updated far fewer times a second.

INPUTS

Seconds = pointer to a LONG variable to receive the current seconds value Micros = pointer to a LONG variable for the current microseconds value

RESULT

Puts the time values into the memory locations specified by the arguments

BUGS

None

SEE ALSO

None

NAME
DisplayAlert -- Create a display of an Alert message

SYNOPSIS
DisplayAlert(AlertNumber, String, Height);

FUNCTION
Creates an Alert display with the specified message.

If the system can recover from this Alert, its a RECOVERY_ALERT and this routine waits until the user presses one of the mouse buttons, after which the display is restored to its original state and a BOOL value is returned by this routine to specify whether or not the User pressed the LEFT mouse button.

If the system cannot recover from this Alert, it's a DEADEND_ALERT and this routine returns immediately upon creating the Alert display. The return value is FALSE.

The AlertNumber is a LONG value, related to the value sent to the Alert() routine. But the only bits that are pertinent to this routine are the ALERT_TYPE bits. These bits must be set to either RECOVERY_ALERT for Alerts from which the system may safely recover, or DEADEND_ALERT for those fatal Alerts. These states are described in the paragraph above. There is a third type of Alert, the DAISY_ALERT, which is used only by the Executive.

The String argument points to an AlertMessage string. The AlertMessage string is comprised of one or more substrings, each of which is comprised of the following components:
- first, a 16-bit x-coordinate and an 8-bit y-coordinate, describing where on the Alert display you want this string to appear. The y-coordinate describes the offset to the baseline of the text.
- then, the bytes of the string itself, which must be null-terminated (end with a byte of zero)
- lastly, the continuation byte, which specifies whether or not there's another substring following this one. If the continuation byte is non-zero, there IS another substring to be processed in this Alert Message. If the continuation byte is zero, this is the last substring in the message.

The last argument, Height, describes how many video lines tall you want the Alert display to be.

INPUTS
"AlertNumber =" the number of this Alert Message. The only pertinent bits of this number are the ALERT_TYPE bits. The rest of the number is ignored by this routine

"String =" pointer to the Alert message string, as described above

"Height =" minimum display lines required for your message

RESULT

A BOOL value of TRUE or FALSE. If this is a DEADEND_ALERT, FALSE is always the return value. If this is a RECOVERY_ALERT. The return value will be TRUE if the User presses the left mouse button in response to your message, and FALSE if the User presses the right hand button is response to your text.

BUGS

If the system is worse off than you think, the level of your Alert may become DEADEND_ALERT without you ever knowing about it.

SEE ALSO

None

NAME
DisplayBeep -- "beeps" the video display

SYNOPSIS
DisplayBeep(Screen);

FUNCTION
"Beeps" the video display by flashing the background color of the specified Screen. If the Screen argument is NULL, every Screen in the display will be beeped. Flashing everyone's Screen is not a polite thing to do, so this should be reserved for dire circumstances.

The reason such a routine is supported is because the Amiga has no internal bell or speaker. When the user needs to know of an event that is not serious enough to require the use of a Requester, the DisplayBeep() function should be called.

INPUTS
Screen =
    pointer to a Screen. If NULL, every Screen in the display will be flashed

RESULT
None

BUGS
None

SEE ALSO
None

NAME
DoubleClick -- Test two time values for double-click timing

SYNOPSIS
DoubleClick(StartSeconds, StartMicros, CurrentSeconds, CurrentMicros);

FUNCTION
Compares the difference in the time values with the double-click timeout range that the user (using the "Preferences" tool) or some other source has configured into the system. If the difference between the specified time values is within the current double-click time range, this function returns TRUE, else it returns FALSE.

These time values can be found in InputEvents and IDCMP Messages. The time values are not perfect; however, they are precise enough for nearly all applications.

INPUTS
StartSeconds, StartMicros = the timestamp value describing the start of
the double-click time period you are considering
CurrentSeconds, CurrentMicros = the timestamp value describing
the end of the double-click time period you are considering

RESULT
If the difference between the supplied timestamp values is within the double-click time range in the current set of Preferences, this function returns TRUE, else it returns FALSE

BUGS
None

SEE ALSO
CurrentTime();

NAME
> DrawBorder -- draws the specified border into the RastPort

SYNOPSIS
> DrawBorder(RastPort, Border, LeftOffset, TopOffset);

FUNCTION
> First, sets up the DrawMode and Pens in the RastPort according to the arguments of the Border structure. Then, draws the vectors of the Border argument into the Rast-Port, offset by the Left and Top Offsets. This routine does Intuition window clipping as appropriate -- if you draw a line outside of your Window, your imagery will be clipped at the Window's edge.
>
> If the NextBorder field of the Border argument is non-zero, the next Border is rendered as well (return to the top of this .SH FUNCTION section for details).

INPUTS
> RastPort =
>> pointer to the RastPort to receive the border crossing
>
> Border =
>> pointer to a Border structure
>
> LeftOffset =
>> the offset which will be added to each vector's x coordinate
>
> TopOffset =
>> the offset which will be added to each vector's y coordinate

RESULT
> None

BUGS
> None

SEE ALSO
> None

NAME

DrawImage -- draws the specified Image into the RastPort

SYNOPSIS

DrawImage(RastPort, Image, LeftOffset, TopOffset);

FUNCTION

First, sets up the DrawMode and Pens in the RastPort according to the arguments of the Image structure. Then, moves the image data of the Image argument into the Rast-Port, offset by the Left and Top Offsets. This routine does Intuition window clipping as appropriate -- if you draw an image outside of your Window, your imagery will be clipped at the Window's edge.

If the NextImage field of the Image argument is non-zero, the next Image is rendered as well (return to the top of this .SH FUNCTION section for details).

INPUTS

RastPort =
> pointer to the RastPort to receive the border crossing

Image =
> pointer to an Image structure

LeftOffset =
> the offset which will be added to the Image's x coordinate

TopOffset =
> the offset which will be added to the Image's y coordinate

RESULT

None

BUGS

None

SEE ALSO

None

NAME
    EndRefresh -- Ends the optimized refresh state of the Window

SYNOPSIS
    EndRefresh(Window, Complete);

FUNCTION
    This function gets you out of the special refresh state of your Window. It is called fol-
    lowing a call to BeginRefresh(), which routine puts you into the special refresh state.
    While your Window is in the refresh state, the only rendering that will be wrought in
    your Window will be to those areas which were recently revealed and need to be re-
    freshed.

    After you've done all the refreshing you want to do for this Window, you should call this
    routine to restore the Window to its non-refreshing state. Then all rendering will go to
    the entire Window, as usual.

    The Complete argument is a boolean TRUE or FALSE value used to describe whether or
    not the refreshing you've done was all the refreshing that needs to be done at this time.
    Most often, this argument will be TRUE. But if, for instance, you have multiple tasks
    or multiple procedure calls which must run to completely refresh the Window, then each
    can call its own Begin/EndRefresh() pair with a Complete argument of FALSE, and only
    the last calls with a Complete argument of TRUE.

INPUTS
    Window =
        pointer to the Window currently in optimized-refresh mode
    Complete =
        Boolean TRUE or FALSE describing whether or not this Window is completely
        refreshed

RESULT
    None

BUGS
    None

SEE ALSO
    BeginRefresh()
    The "Screens" chapter in this book

NAME
    EndRequest -- Ends the Request and resets the Window

SYNOPSIS
    EndRequest(Requester, Window);

FUNCTION
    Ends the Request by erasing the Requester and resetting the Window. Note that this
    doesn't necessarily clear all Requesters from the Window, only the specified one. If the
    Window labors under other Requesters, they will remain in the Window.

INPUTS
    Requester =
        pointer to the Requester to be removed

    Window =
        pointer to the Window structure with which this Requester is associated

RESULT
    None

BUGS
    None

SEE ALSO
    None

NAME
   FreeRemember -- Free memory allocated by calls to AllocRemember()

SYNOPSIS
   FreeRemember(RememberKey, ReallyForget);

FUNCTION
   This function frees up memory allocated by the AllocRemember() function. It will either
   free up just the Remember structures, which supply the link nodes that tie your alloca-
   tions together, or it will deallocate both the link nodes AND your memory buffers too.

   If you want to deallocate just the Remember structure link nodes, you should set the
   ReallyForget argument to FALSE. However, if you want FreeRemember to really forget
   about all the memory, including both the Remember structure link nodes and the buffers
   you requested via earlier calls to AllocRemember() then you should set the ReallyForget
   argument to TRUE.

INPUTS
   RememberKey =
      the address of a pointer to struct Remember. This pointer should either be
      NULL or set to some value (possibly NULL) by a call to AllocRemember(). For
      example:

      struct Remember *RememberKey;
      RememberKey = NULL;
      AllocRemember(&RememberKey, BUFSIZE, MEMF_CHIP);
      FreeRemember(&RememberKey, TRUE);

   ReallyForget =
      a BOOL FALSE or TRUE describing, respectively, whether you want to free up
      only the Remember nodes or if you want this procedure to really forget about all
      of the memory, including both the nodes and the memory buffers pointed to by
      the nodes.

RESULT
   None

BUGS
   None

SEE ALSO
   AllocRemember()

NAME
FreeSysRequest -- Frees up memory used by a call to BuildSysRequest()

SYNOPSIS
FreeSysRequest(Window);

FUNCTION
This routine frees up all memory allocated by a successful call to the BuildSysRequest() procedure. If BuildSysRequest() returned a pointer to a Window, then you are able to Wait() on the message port of that Window to detect an event which satisfies the Requester. When you want to remove the Requester, you call this procedure. It ends the Requester and deallocates any memory used in the creation of the Requester.

NOTE: if BuildSysRequest() did not return a pointer to a Window, you should not call FreeSysRequest()!

INPUTS
Window = a copy of the Window pointer returned by a successful call to
the BuildSysRequest() procedure

RESULT
None

BUGS
None

SEE ALSO
BuildSysRequest()
The Executive's Wait() instruction
AutoRequest()

NAME
GetDefPrefs -- Get a copy of the the Intuition default Preferences

SYNOPSIS
GetDefPrefs(PrefBuffer, Size);

FUNCTION
Gets a copy of the Intuition default preferences data. Writes the data into the buffer you specify. The number of bytes you want copied is specified by the Size argument.

The default Preferences are those that Intuition uses when it is first opened. If no preferences file is found, these are the preferences that are used. These would also be the startup Preferences in an AmigaDOS-less environment.

It is legal to take a partial copy of the Preferences structure. The more pertinent Preferences variables have been grouped near the top of the structure to facilitate the memory conservation that can be had by taking a copy of only some of the Preferences structure.

INPUTS
PrefBuffer =
    pointer to the memory buffer to receive your copy of the Intuition Preferences

Size =
    the number of bytes in your PrefBuffer, the number of bytes you want copied from the system's internal Preference settings

RESULT
Returns your Preferences pointer

BUGS
None

SEE ALSO
GetPrefs()

NAME

GetPrefs -- Get the current setting of the Intuition Preferences

SYNOPSIS

GetPrefs(PrefBuffer, Size);

FUNCTION

Gets a copy of the current Intuition Preferences data. Writes the data into the buffer you specify. The number of bytes you want copied is specified by the Size argument.

It is legal to take a partial copy of the Preferences structure. The more pertinent Preferences variables have been grouped near the top of the structure to facilitate the memory conservation that can be had by taking a copy of only some of the Preferences structure.

INPUTS

PrefBuffer =
>pointer to the memory buffer to receive your copy of the Intuition Preferences

Size =
>the number of bytes in your PrefBuffer, the number of bytes you want copied from the system's internal Preference settings

RESULT

Returns a copy of your Preferences pointer

BUGS

None

SEE ALSO

GetDefPrefs()

NAME
    InitRequester -- initializes a Requester structure

SYNOPSIS
    InitRequester(Requester);

FUNCTION
    Initializes a requester for general use. After calling InitRequester, you need fill in only
    those Requester values that fit your needs. The other values are set to states that Intui-
    tion regards as NULL

INPUTS
    Requester = a pointer to a Requester

RESULT
    None

BUGS
    None

SEE ALSO
    None

NAME
    IntuiTextLength -- Return the length (pixel-width) of an IntuiText

SYNOPSIS
    IntuiTextLength(IText);

FUNCTION
    This routine accepts a pointer to an instance of an IntuiText structure, and returns the
    length (the pixel-width) of the string that that instance of the structure represents.

    All of the usual IntuiText rules apply. Most notably, if the Font pointer of the structure
    is set to NULL, you'll get the pixel-width of your text in terms of the current default
    font.

INPUTS
    IText = pointer to an instance of an IntuiText structure

RESULT
    Returns the pixel-width of the text specified by the IntuiText data

BUGS
    None

SEE ALSO
    None

NAME
ItemAddress -- Returns the address of the specified MenuItem

SYNOPSIS
ItemAddress(MenuStrip, MenuNumber);

FUNCTION
This routine feels through the specified MenuStrip and returns the address of the Item specified by the MenuNumber. Typically, you will use this routine to get the address of a MenuItem from a MenuNumber sent to you by Intuition after User has played with your Menus.

This routine requires that the arguments are well-defined. MenuNumber may be equal to MENUNULL, in which case this routine returns NULL. If MenuNumber doesn't equal MENUNULL, it's presumed to be a valid Item number selector for your MenuStrip, which includes:

- a valid Menu number

- a valid Item Number

- if the Item specified by the above two components has a SubItem, the MenuNumber may have a SubItem component too

Note that there must be BOTH a Menu number and an Item number. Because a SubItem specifier is optional, the address returned by this routine may point to either an Item or a SubItem.

INPUTS
MenuStrip =
a pointer to the first Menu in your MenuStrip

MenuNumber =
the value which contains the packed data that selects the Menu and Item (and SubItem)

RESULT
If MenuNUmber === MENUNULL, this routine returns NULL else this routine returns the address of the MenuItem specified by MenuNumber.

BUGS
None

SEE ALSO
The "Menus" chapter in this book for more information about MenuNumbers

NAME

MakeScreen -- Do an Intuition-integrated MakeVPort() of a custom screen

SYNOPSIS

MakeScreen(Screen);

FUNCTION

This procedure allows you to do a MakeVPort() for the ViewPort of your Custom Screen in an Intuition-integrated way. This allows you to do your own Screen manipulations without worrying about interference with Intuition's usage of the same ViewPort.

After calling this routine, you can call RethinkDisplay() to incorporate the new ViewPort of your custom screen into the Intuition display.

INPUTS

Screen = address of the Custom Screen structure

RESULT

None

BUGS

None

SEE ALSO

RethinkDisplay()
RemakeDisplay()
The graphics library's MakeVPort()

NAME
     ModifyIDCMP -- Modify the state of the Window's IDCMP

SYNOPSIS
     ModifyIDCMP(Window, IDCMPFlags);

FUNCTION
     This routine modifies the state of your Window's IDCMP (Intuition Direct Communica-
     tion Message Port). The state is modified to reflect your desires as described by the flag
     bits in the value IDCMPFlags. When you call ModifyIDCMP(), if the IDCMPFlags
     equals NULL, you are asking that if the Port is currently opened, you want it closed. If
     you set any of the IDCMPFlags, this means that you want the message ports to be open;
     if not currently opened, the Ports will be opened now.

     The four actions that might be taken are:
          - if there is currently no IDCMP in the given Window, and IDCMPFlags is
            NULL, nothing happens
          - if there is currently no IDCMP in the given Window, and any of the IDCMP-
            Flags is selected (set), then the IDCMP of the Window is created, including
            allocating and initializing the message ports and allocating a Signal bit for
            your Port. See the "Input and Output Methods" chapter of this book for full
            details
          - if the IDCMP for the given Window is opened, and the IDCMPFlags argu-
            ment is NULL, this says that you want Intuition to close the Ports, free the
            buffers and free your Signal bit. You MUST be the same Task that was ac-
            tive when this Signal bit was allocated
          - if the IDCMP for the given Window is opened, and the IDCMPFlags argu-
            ment is not NULL, this means that you want to change the state of which
            events will be broadcast to you through the IDCMP
     NOTE: You can set up the Window->UserPort to any Port of your own before you call
     ModifyIDCMP(). If IDCMPFlags is non-null but your UserPort is already initialized, In-
     tuition will assume that it's a valid Port with Task and Signal data preset and Intuition
     won't disturb your set-up at all, Intuition will just allocate the Intuition Message Port
     half of it. The converse is true as well: if UserPort is NULL when you call here with
     IDCMPFlags === NULL, I'll deallocate only the Intuition Port. This allows you to use a
     Port that you already have allocated:
          - OpenWindow() with IDCMPFlags equal to NULL (open no ports)
          - set the UserPort variable of your Window to any valid Port of your own
            choosing
          - call ModifyIDCMP with IDCMPFlags set to what you want
          - then, to clean up later, set UserPort equal to NULL before calling CloseWin-
            dow() (leave IDCMPFlags alone)
     A Grim, Foreboding Note: if you are ever rude enough to close an IDCMP without first
     having Reply()'d to all of the messages sent to the IDCMP port, Intuition in turn will so
     rude as to reclaim and deallocate its messages without waiting for your permission.

INPUTS
Window =
pointer to the Window structure containing the IDCMP Ports
IDCMPFlags =
the flag bits describing the new desired state of the IDCMP

RESULT
None

BUGS
None

SEE ALSO
OpenWindow

NAME

ModifyProp -- Modify the current parameters of a Proportional Gadget

SYNOPSIS

ModifyProp(Gadget, Pointer, Requester,
     Flags, HorizPot, VertPot, HorizBody, VertBody);

FUNCTION

Modifies the parameters of the specified Proportional Gadget. The Gadget's internal state is then recalculated and the imagery is redisplayed, wherever it is that the Pointer argument points.

The Pointer argument can point to either a Window or a Screen structure. Which it actually points to is decided by examining the SCRGADGET flag of the Gadget; if the flag is set, Pointer points to a Screen structure, otherwise it points to a Window structure.

The Requester variable can point to a Requester structure. If the Gadget has the REQGADGET flag set, the Gadget is in a Requester and the Pointer must necessarily point to a Window. If this is not the Gadget of a Requester, the Requester argument may be NULL.

INPUTS

PropGadget =
     pointer to a Proportional Gadget

Pointer =
     pointer to the "owning" display element of the Gadget, be it a Window or a Screen

Requester =
     pointer to a Requester (may be NULL if this isn't a Requester Gadget)

Flags =
     value to be stored in the Flags variable of the PropInfo

HorizPot =
     value to be stored in the HorizPot variable of the PropInfo

VertPot =
     value to be stored in the VertPot variable of the PropInfo

HorizBody =
     value to be stored in the HorizBody variable of the PropInfo

VertBody =
     value to be stored in the VertBody variable of the PropInfo

RESULT

None

BUGS

None

SEE ALSO
        None

NAME
MoveScreen -- attempts to move the Screen by the delta amounts

SYNOPSIS
MoveScreen(Screen, DeltaX, DeltaY);

FUNCTION
Attempts to move the specified Screen. This movement must follow certain constraints (only for the current release of the software):

- the bottom of the Screen must not move higher than the bottom of the video display

- horizontal movements are ignored

If the DeltaX and DeltaY variables you specify would move the Screen in a way that violates the above restrictions, the Screen will be moved as far as possible.

INPUTS
Screen = pointer to a Screen structure DeltaX = amount to move the screen on the x-axis DeltaY = amount to move the screen on the y-axis

RESULT
None

BUGS
None

SEE ALSO
None

NAME
    MoveWindow -- Ask Intuition to move a Window

SYNOPSIS
    MoveWindow(Window, DeltaX, DeltaY);

FUNCTION
    This routine sends a request to Intuition asking to move the Window the specified dis-
    tance. The delta arguments describe how far to move the Window along the respective
    axes.

    Note that the Window will not be moved immediately, but rather will be moved the
    next time Intuition receives an input event, which happens currently at a minimum rate
    of ten times per second, and a maximum of sixty times a second.

    This routine does no error-checking. If your delta values specify some far corner of the
    Universe, Intuition will attempt to move your Window to the far corners of the
    Universe. Because of the distortions in the space-time continuum that can result from
    this, as predicted by special relativity, the result is generally not a pretty sight.

INPUTS
    Window =
        pointer to the structure of the Window to be moved
    DeltaX =
        signed value describing how far to move the Window on the x-axis
    DeltaY =
        signed value describing how far to move the Window on the y-axis

RESULT
    None

BUGS
    None

SEE ALSO
    SizeWindow()
    WindowToFront()
    WindowToBack()

NAME
OffGadget -- disables the specified Gadget

SYNOPSIS
OffGadget(Gadget, Pointer, Requester);

FUNCTION
This command disables the specified Gadget. When a Gadget is disabled, these things happen:
- its imagery is displayed ghosted
- the GADGDISABLED flag is set
- the Gadget cannot be selected by User

The Pointer argument can point to either a Window or a Screen structure. Which it actually points to is decided by examining the SCRGADGET flag of the Gadget; if the flag is set, Pointer points to a Screen structure, else it points to a Window structure. The Requester variable can point to a Requester structure. If the Gadget has the REQGADGET flag set, the Gadget is in a Requester and the Pointer must necessarily point to a Window. If this is not the Gadget of a Requester, the Requester argument may be NULL.

NOTE: it's never safe to tinker with the Gadget list yourself. Don't supply some Gadget that Intuition hasn't already processed in the usual way.

NOTE: if you have specified that this is the Gadget list of a Requester, that Requester must be currently displayed

INPUTS
Gadget =
> pointer to the Gadget that you want disabled

Pointer =
> pointer to either a Screen or Window structure (defined by the SCRGADGET flag of the Gadget)

Requester =
> pointer to a Requester (may by NULL if this isn't a Requester Gadget list)

RESULT
None

BUGS
None

SEE ALSO
None

NAME
    OffMenu -- disables the given menu or menu item

SYNOPSIS
    OffMenu(Window, MenuNumber);

FUNCTION
    This command disables a sub-item, an item, or a whole menu If the base of the menu
    number matches the menu currently revealed, the menustrip is redisplayed

INPUTS
    Window =
            pointer to the window

    MenuNumber =
            the menu piece to be enabled

RESULT
    None

BUGS
    None

SEE ALSO
    None

NAME
>   OnGadget -- enables the specified Gadget

SYNOPSIS
>   OnGadget(Gadget, Pointer, Requester);

FUNCTION
>   This command enables the specified Gadget. When a Gadget is enabled, these things happen:
>   - its imagery is displayed normally (not ghosted)
>   - the GADGDISABLED flag is cleared
>   - the Gadget can thereafter be selected by the user
>
>   The Pointer argument can point to either a Window or a Screen structure: which it actually points to is decided by examining the SCRGADGET flag of the Gadget: if the flag is set, Pointer points to a Screen struct, else it points to a Window. The Requester variable can point to a Requester structure. If the Gadget has the REQGADGET flag set, the Gadget is in a Requester and the Pointer must necessarily point to a Window. If this is not the Gadget of a Requester, the Requester argument may be NULL.
>
>   NOTE: It's never safe to tinker with the Gadget list yourself. Don't supply some Gadget that Intuition hasn't already processed in the usual way.
>
>   NOTE: If you have specified that this is the Gadget list of a Requester, that Requester must be currently displayed.

INPUTS
>   Gadget =
>   >   pointer to the Gadget that you want enabled
>
>   Pointer =
>   >   pointer to either a Screen or Window structure (defined by the SCRGADGET flag of the Gadget)
>
>   Requester =
>   >   pointer to a Requester (may be NULL if this isn't a Requester Gadget list)

RESULT
>   None

BUGS
>   None

SEE ALSO
>   None

NAME
    OnMenu -- enables the given menu or menu item

SYNOPSIS
    OnMenu(Window, MenuNumber);

FUNCTION
    This command enables a sub-item, an item, or a whole menu. If the base of the menu
    number matches the menu currently revealed, the menustrip is redisplayed.

INPUTS
    Window =
            pointer to the window

    MenuNumber =
            the menu piece to be enabled

RESULT
    None

BUGS
    None

SEE ALSO
    None

NAME
OpenScreen -- open an Intuition Screen

SYNOPSIS
OpenScreen(NewScreen); where NewScreen is a structure that is initialized with:
  Left, Top, Width, Height, Depth, DetailPen, BlockPen,
  ViewModes, Type, Font, DefaultTitle, Gadgets

FUNCTION
Opens an Intuition Screen according to the specified parameters. Does all the allocations, sets up the Screen structure and all substructures completely, and links this Screen's ViewPort into Intuition's View of the world.

Before you call OpenScreen(), you must initialize an instance of a NewScreen structure. NewScreen is a structure that contains all of the arguments needed to open a Screen. The NewScreen structure may be discarded immediately after it is used to open the Screen.

The TextAttr pointer that you supply as an argument will be used as the default font for all Intuition-managed text that appears in the Screen and its Windows. This includes, but is not limited to, the text on the title bars of both Screen and Windows.

The SHOWTITLE flag is set to TRUE by default when a Screen is opened. To change this, you must call the routine ShowTitle().

INPUTS
NewScreen = pointer to an instance of a NewScreen structure.
That structure is initialized with the following information:

Left =
  initial x-position of your Screen (should be zero for now)
Top =
  initial y-position of the opening Screen
Width =
  the width for this Screen's RastPort
Height =
  the height for his Screen's RastPort
Depth =
  number of BitPlanes
DetailPen =
  pen number for details (like gadgets or text in title bar)
BlockPen =
  pen number for block fills (like title bar)
Type =
  Screen type (if you are not Intuition, this should be equal to CUSTOMSCREEN).
  Types currently supported include:

CUSTOMSCREEN -- this is your own Screen

You may also set the Type flags CUSTOMBITMAP and then supply your own BitMap for Intuition to use rather than allocating the display memory for you.

ViewModes =
> the appropriate argument for the data type ViewPort.Modes. these might include:
>> HIRES for this screen to be HIRES width
>> INTERLACE for the display to switch to interlace
>> SPRITES for this Screen to use sprites
>> DUALPF for dual-playfield mode (not supported yet)

Font =
> pointer to the default TextAttr structure for this Screen and all Windows that open in this Screen

DefaultTitle =
> pointer to a line of text that will be displayed along the Screen's Title Bar. Null terminated, or just a NULL pointer to get no text

Gadgets =
> first in a linked list of the Gadgets you want for this Screen

CustomBitMap =
> if you're not supplying a custom BitMap, this value is ignored. However, if you have your own display memory that you want used for this Screen, the Custom-BitMap argument should point to the BitMap that describes your display memory. See the "Screens" chapter and the "Amiga ROM Kernel Manual" for more information about BitMaps.

RESULT
> If all is well, returns the pointer to your new Screen.
> If anything goes wrong, returns NULL.

BUGS
> No way

SEE ALSO
> OpenWindow

NAME
OpenWindow -- Opens an Intuition Window

SYNOPSIS
OpenWindow(NewWindow);
where the NewWindow structure is initialized with:

> Left, Top, Width, Height, DetailPen, BlockPen, Flags, IDCMPFlags, Gadgets, CheckMark, Text, Type, Screen, BitMap, MinWidth, MinHeight, MaxWidth, MaxHeight

FUNCTION
Opens an Intuition window of the given height, width and depth, including the specified system Gadgets as well as any of your own. Allocates everything you need to get going.

Before you call OpenWindow(), you must initialize an instance of a NewWindow structure. NewWindow is a structure that contains all of the arguments needed to open a Window. The NewWindow structure may be discarded immediately after it is used to open the Window.

If Type == CUSTOMSCREEN, you must have opened your own Screen already via a call to OpenScreen(). Then Intuition uses your screen argument for the pertinent information needed to get your Window going. On the other hand, if type === one of the Intuition's standard Screens, your screen argument is ignored. Instead, Intuition will check to see whether or not that Screen already exists: if it doesn't, it will be opened first before Intuition opens your window in the Standard Screen. If the flag SUPER_BITMAP is set, the bitmap variable must point to your own BitMap. The DetailPen and the BlockPen are used for system rendering; for instance, the Title bar is first filled using the BlockPen, and then the Gadgets and text are rendered using Detail-Pen. You can either choose to supply special pens for your Window, or, by setting either of these arguments to -1, the Screen's Pens will be used instead.

INPUTS
NewWindow =

> pointer to an instance of a NewWindow structure. That structure is initialized with the following data:

> --

Left =

> the initial x-position for your window

Top =

> the initial y-position for your window

Width =

> the initial width of this window

Height =

> the initial height of this window

DetailPen =

> pen number (or -1) for the rendering of Window details
> (like gadgets or text in title bar)

BlockPen =
> pen number (or -1) for Window block fills (like Title Bar)

Flags =
> specifiers for your requirements of this window, including:
>
> - which system Gadgets you want attached to your window:
>   - WINDOWDRAG allows this Window to be dragged
>   - WINDOWDEPTH lets the user depth-arrange this Window
>   - WINDOWCLOSE attaches the standard Close Gadget
>   - WINDOWSIZING allows this Window to be sized. If you ask the WINDOWSIZING Gadget, you must specify one or both of the flags SIZEBRIGHT and SIZEBBOTTOM below; if you don't, the default is SIZEBRIGHT. See the following items SIZEBRIGHT and SIZEBBOTTOM for extra information.
>   - SIZEBRIGHT is a special system Gadget flag that you set to specify whether or not you want the RIGHT Border adjusted to account for the physical size of the Sizing Gadget. The Sizing Gadget must, after all, take up room in either the right or bottom border (or both, if you like) of the Window. Setting either this or the SIZEBBOTTOM flag selects which edge will take up the slack. This will be particularly useful to applications that want to use the extra space for other Gadgets (like a Proportional Gadget and two Booleans done up to look like scroll bars) or, for for instance, applications that want every possible horizontal bit and are willing to lose lines vertically.
>
>   NOTE: if you select WINDOWSIZING, you must select either SIZEBRIGHT or SIZEBBOTTOM or both. If you select neither, the default is SIZEBRIGHT.
>   - SIZEBBOTTOM is a special system Gadget flag that you set to specify whether or not you want the BOTTOM Border adjusted to account for the physical size of the Sizing Gadget. For details, refer to SIZEBRIGHT above. NOTE: if you select WINDOWSIZING, you must select either SIZEBRIGHT or SIZEBBOTTOM or both. If you select neither, the default is SIZEBRIGHT.
> - GIMMEZEROZERO for easy but expensive output
> - What type of window raster you want, either:
>   - SIMPLE_REFRESH
>   - SMART_REFRESH
>   - SUPER_BITMAP
> - BACKDROP for whether or not you want this window to be one of Intuition's special backdrop windows. See BORDERLESS as well.
> - REPORTMOUSE for whether or not you want to "listen" to mouse movement events whenever your Window is the active one. After you've opened your Window, if you want to change you can later change the status of this via a call to ReportMouse(). Whether or not your Window is listening to Mouse is affected by Gadgets too, since they can cause you to start getting re-

ports too if you like. The mouse move reports (either InputEvents or messages on the IDCMP) that you get will have the x/y coordinates of the current mouse position, relative to the upper-left corner of your Window (GIMMEZEROZERO notwithstanding). This flag can work in conjunction with the IDCMP Flag called MOUSEMOVE, which allows you to listen via the IDCMP.

- BORDERLESS should be set if you want a Window with no Border padding. Your Window may have the Border variables set anyway, depending on what Gadgetry you've requested for the Window, but you won't get the standard border lines and spacing that comes with typical Windows. This is a good way to take over the entire Screen, since you can have a Window cover the entire width of the Screen using this flag. This will work particularly well in conjunction with the BACKDROP flag (see above), since it allows you to open a Window that fills the ENTIRE Screen.

  NOTE: this is not a flag that you want to set casually, since it may cause visual confusion on the Screen. The Window borders are the only dependable visual division between various Windows and the background Screen. Taking er takes away that visual cue, so make sure that your design doesn't need it at all before you proceed.

- ACTIVATE is the flag you set if you want this Window to automatically become the active Window. The active Window is the one that receives input from the keyboard and mouse. It's usually a good idea to to have the Window you open when your application first starts up be an ACTIVATED one, but all others opened later not be ACTIVATED (if the user is off doing something with another Screen, for instance, your new Window will change where the input is going, which would have the effect of yanking the input rug from under the user). Please use this flag thoughtfully and carefully.

- RMBTRAP, when set, causes the right mouse button events to be trapped and broadcast as events. You can receive these events through either the IDCMP or the Console.

IDCMPFlags =

IDCMP is the acronym for Intuition Direct Communications Message Port. It's Intuition's sole acronym, given in honor of all hack-heads who love to mangle our brains with maniacal names, and fashioned especially cryptic and unpronounceable to make them squirm with sardonic delight. Here's to you, my chums. Meanwhile, I still opt (and argue) for simplicity and elegance.

If any of the IDCMP Flags is selected, Intuition will create a pair of messageports and use them for direct communications with the Task opening this Window (as compared with broadcasting information via the Console Device). See the "Input and Output Methods" chapter of this book for complete details.

You request an IDCMP by setting any of these flags. Except for the special VERIFY flags, every other flag you set tells me that if a given event occurs which your program wants to know about, I'm to broadcast the details of that event through the IDCMP rather than via the Console device. device. This allows a program to interface with Intuition directly, rather than going through the Console device.

Remember, if you are going to open both an IDCMP and a Console, it will be far

better to get most of the event messages via the Console. Reserve your usage of the IDCMP for special performance cases; that is, when you aren't going to open a Console for your Window and you do want to learn about a certain set of events (for instance, CLOSEWINDOW); another example would be SIZEVERI-FY, which is a function that you get ONLY through the use of the IDCMP (because the Console doesn't give you any way to talk to Intuition directly).

On the other hand, if the IDCMPFlags argument is equal to zero, no IDCMP is created and the only way you can learn about any Window event for this Window is via a Console opened for this Window. And you have no way to SIZEVERIFY.

If you want to change the state of the IDCMP some time after you've opened the Window (including opening or closing the IDCMP) you call the routine ModifyIDCMP().

The flags you can set are:

- REQVERIFY is the flag which, like SIZEVERIFY and(see MENUVERIFY (see immediately below), specifies that you want to make sure that your graphical state is quiescent before something extraordinary happens. In this case, the extraordinary event is that a rectangle of graphical data is about to be blasted into your Window. If you're drawing into that Window, you probably will wish to make sure that you've ceased drawing before the user is allowed to bring up the DMRequest you've set up, and the same for when system has a request for the user. Set this flag to ask for that verification step.

- REQCLEAR is the flag you set to hear about it when the last Requester is cleared from your Window and it's safe for you to start output again (presuming you're using REQVERIFY)

- REQSET is a flag that you set to receive a broadcast when the first Requester is opened in your Window. Compare this with REQCLEAR above. This function is distinct from REQVERIFY. This functions merely tells you that a Requester has opened, whereas REQVERIFY requires you to respond before the Requester is opened.

- MENUVERIFY is the flag you set to have Intuition stop and wait for you to finish all graphical output to your Window before rendering the menus. Menus are currently rendered in the most memory-efficient way, which involves interrupting output to all Windows in the Screen before the Menus are drawn. If you need to finish your graphical output before this happens, you can set this flag to make sure that you do.

- SIZEVERIFY means that you will be doing output to your Window which depends on a knowledge of the current size of the Window. If the user wants to resize the Window, you may want to make sure that any queued output completes before the sizing takes place (critical Text, for instance). If this is the case, set this flag. Then, when the user wants to size, Intuition will send you the SIZEVERIFY message and Wait() until you reply that it's OK to proceed with the sizing.

  NOTE: when I say that Intuition will Wait() until you reply, what I'm really saying is that User will WAIT until you reply, which suffers the great negative potential of User-Unfriendliness. So remember: use this flag sparingly, and, as always with any IDCMP Message you receive, Reply to it promptly!

Then, after User has sized the Window, you can find out about it using NEWSIZE:

- NEWSIZE is the flag that tells Intuition to send an IDCMP Message to you after the user has resized your Window. At this point, you could examine the size variables in your Window structure to discover the new size of the Window

- REFRESHWINDOW when set will cause a Message to be sent whenever your Window needs refreshing. This flag makes sense only with SIMPLE_REFRESH and SMART_REFRESH Windows.

- MOUSEBUTTONS will get reports about Mouse-button Up/Down events broadcast to you (Note: only the ones that don't mean something to Intuition. If the user clicks the Select button over a Gadget, Intuition deals with it and you don't find out about it through here).

- MOUSEMOVE will work only if you've set the flag REPORTMOUSE above, or if one of your Gadgets has the flag FOLLOWMOUSE set. Then all mouse movements will be reported here.

- GADGETDOWN means that when the User "selects" a Gadget you've created with the GADGIMMEDIATE flag set, the fact will be broadcast through the IDCMP.

- GADGETUP means that when the User "releases" a Gadget that you've created with the RELVERIFY flag set, the fact will be broadcast through the IDCMP.

- MENUPICK selects that MenuNumber data will come this way

- CLOSEWINDOW means broadcast the CLOSEWINDOW event through the IDCMP rather than the Console

- RAWKEY selects that all RAWKEY events are transmitted via the IDCMP Note that these are absolutely RAW keycodes, which you will have to massage before using. Setting this and the MOUSE flags effectively eliminates the need to open a Console Device to get input from the keyboard and mouse. Of course, in exchange you lose all of the Console features, most notably the "cooking" of input data and the systematic output of text to your Window.

Gadgets =
>   the pointer to the first of a linked list of the your own Gadgets which you want attached to this Window. Can be NULL if you have no Gadgets of your own

CheckMark =
>   a pointer to an instance of the struct Image where can be found the imagery you want used when any of your MenuItems is to be checkmarked. If you don't want to supply your own imagery and you want to just use Intuition's own checkmark, set this argument to NULL

Text =
>   a null-terminated line of text to appear on the title bar of your window (may be null if you want no text)

Type =
>   the Screen type for this window. If this equal CUSTOMSCREEN, you must have already opened a CUSTOMSCREEN (see text above). Types available include:

- WBENCHSCREEN
- CUSTOMSCREEN

Screen =
> if your type is one of Intuition's Standard Screens, then this argument is ignored. However, if Type == CUSTOMSCREEN, this must point to the structure of your own Screen

BitMap =
> if you have specified SUPER_BITMAP as the type of raster you want for this Window, then this value points to a instance of the struct BitMap. However, if the raster type is NOT SUPER_BITMAP, this pointer is ignored

MinWidth, MinHeight, MaxWidth, MaxHeight =
> the size limits for this Window. These must be reasonable values, which is to say that the minimums cannot be greater than the current size, nor can the maximums be smaller than the current size. If they are, they're ignored. Any one of these can be initialized to zero, which means that that limit will be set to the current dimension of that axis. The limits can be changed after the Window is opened by calling the WindowLimits() routine. If you haven't requested the WINDOWSIZING option, these variables are ignored so you don't have to initialize them.

RESULT
> If all is well, returns the pointer to your new Window.
> If anything goes wrong, returns NULL.

BUGS
> None

SEE ALSO
> OpenScreen()
> ModifyIDCMP()
> WindowTitles()

NAME
OpenWorkBench -- Opens the WorkBench Screen

SYNOPSIS
BOOL OpenWorkBench();

FUNCTION
This routine attempts to reopen the WorkBench. The actions taken are:

- general good stuff and nice things, and then return TRUE

- find that something has gone wrong, and return FALSE

Even though this routine does return a BOOL value, you can ignore the return value if you want

INPUTS
None

RESULT
TRUE if the WorkBench Screen opened successfully, or was already opened.
FALSE if anything went wrong and the WorkBench Screen isn't out there.

BUGS
None

SEE ALSO
None

NAME
PrintIText -- prints the text according to the IntuiText argument

SYNOPSIS
PrintIText(RastPort, IText, LeftEdge, TopEdge)

FUNCTION
Prints the IntuiText into the specified RastPort. Sets up the RastPort as specified by the IntuiText values, then prints the text into the RastPort at the IntuiText x/y coordinates offset by the left/top arguments.

This routine does Intuition window clipping as appropriate -- if you print text outside of your Window, your characters will be clipped at the Window's edge.

If the NextText field of the IntuiText argument is non-zero, the next IntuiText is rendered as well (return to the top of this FUNCTION section for details).

INPUTS
RastPort =
>   the RastPort destination of the text

IText =
>   pointer to an instance of the structure IntuiText

LeftEdge =
>   left offset of the IntuiText into the RastPort

TopEdge =
>   top offset of the IntuiText into the RastPort

RESULT
None

BUGS
None

SEE ALSO
None

NAME
RefreshGadgets -- Refresh (redraw) the Gadget display

SYNOPSIS
RefreshGadgets(Gadgets, Pointer, Requester);

FUNCTION
Refreshes (redraws) all of the Gadgets in the Gadget List starting from the specified Gadget.

The Pointer argument can point to either a Window or a Screen structure. Which it actually points to is decided by examining the SCRGADGET flag in the first Gadget of the list; if the flag is set, then Pointer points to a Screen structure, else it points to a Window structure.

The Requester variable can point to a Requester structure. If the first Gadget in the list has the REQGADGET flag set, the Gadget list refers to Gadgets in a Requester and the Pointer must necessarily point to a Window. If these are not the Gadgets of a Requester, the Requester argument may be NULL.

The two main reasons why you might want to use this routine are: first, that you've modified the imagery of the Gadgets in your display and you want the new imagery to be displayed; secondly, if you think that some graphic operation you just performed trashed the Gadgetry of your display, this routine will refresh the imagery for you.

The Gadgets argument can be a copy of the FirstGadget variable in either the Screen or Window structure that you want refreshed: the effect of this will be that all Gadgets will be redrawn. However, you can selectively refresh just some of the Gadgets by starting the refresh part-way into the list: for instance, redrawing your Window non-GIMMEZEROZERO Gadgets only, which you've conveniently grouped at the end of your Gadget list.

NOTE: It's never safe to tinker with the Gadget list yourself. Don't supply some Gadget list that Intuition hasn't already processed in the usual way.

NOTE: If you have specified that this is the Gadget list of a Requester, that Requester must be currently displayed

INPUTS
Gadgets =
> pointer to the first in the list of Gadgets wanting refreshment

Pointer =
> pointer to either a Screen or Window structure (defined by the SCRGADGET flag of the first Gadget (see next))

Requester =
> pointer to a Requester (may by NULL if this isn't a Requester Gadget list)

RESULT
> None

BUGS
> None

SEE   ALSO
> None

NAME
RemakeDisplay -- Remake the entire Intuition display

SYNOPSIS
RemakeDisplay();

FUNCTION
This is the big one.

This procedure remakes the entire Intuition display. It calls MakeScreen() for every Screen in the system, and then it calls RethinkDisplay() which rethinks the relationships of the Screens to one another and then rethinks the display copper lists.

WARNING: This routine can take several milliseconds to run, so do not use it lightly. RethinkDisplay() (called by this routine) does a Forbid() on entry and a Permit() on exit, which can seriously degrade the performance of the multi-tasking Executive.

INPUTS
None

RESULT
None

BUGS
None

SEE ALSO
RethinkDisplay()
The graphics library's MakeScreen()

NAME
     RemoveGadget -- removes a Gadget from a Window or a Screen

SYNOPSIS
     USHORT RemoveGadget(Pointer, Gadget);

FUNCTION
     Removes the given Gadget from the Gadget list of the specified Window or Screen. Re-
     turns the ordinal position of the removed Gadget. If the Gadget's SCRGADGET flag is
     set, the Pointer variable is regarded as a pointer to a Screen; else, it's regarded as a
     pointer to a Window. If the Gadget pointer points to a Gadget that isn't in the ap-
     propriate list, -1 is returned. If there aren't any Gadgets in the list, -1 is returned. If
     you remove the 65535th Gadget from the list -1 is returned.

INPUTS
     Pointer =
               pointer to the Window or Screen from which the Gadget is to be removed. the
               Gadget's SCRGADGET flag describes whether this is a pointer to a Window or a
               Screen

     Gadget =
               pointer to the Gadget to be removed. The Gadget itself describes whether this is
          ,   a Gadget that should be removed from the Window or the Screen

RESULT
     Returns the ordinal position of the removed Gadget. If the Gadget wasn't found in the
     appropriate list, or if there are no Gadgets in the list, returns -1.

BUGS
     None

SEE ALSO
     AddGadget

NAME
ReportMouse -- tells Intuition whether or not to report mouse movement

SYNOPSIS
ReportMouse(Window, Boolean);

FUNCTION
Tells Intuition whether or not to broadcast mouse-movement events to this Window when it's the active one. The Boolean value specifies whether to start or stop broadcasting position information of mouse-movement. If the Window is the active one, mouse-movement reports start coming immediately afterwards. This same routine will change the current state of the FOLLOWMOUSE function of a currently-selected Gadget too. Note that calling ReportMouse() when a Gadget is selected will only temporarily change whether or not mouse movements are reported while the Gadget is selected; the next time the Gadget is selected, its FOLLOWMOUSE flag is examined anew. Note also that calling ReportMouse() when no Gadget is currently selected will change the state of the Window's REPORTMOUSE flag, but will have no effect on any Gadget that may be subsequently selected.

The ReportMouse() function is first performed when OpenWindow() is first called; if the flag REPORTMOUSE is included among the options, then all mouse-movement events are reported to the opening task and will continue to be reported until ReportMouse() is called with a Boolean value of FALSE. If REPORTMOUSE is not set, then no mouse-movement reports will be broadcast until ReportMouse() is called with a Boolean of TRUE.

INPUTS
Window =
> pointer to a Window structure associated with this request

Boolean =
> TRUE or FALSE value specifying whether to turn this function on or off

RESULT
None

BUGS
None

SEE ALSO
None

NAME
    Request -- Activates a Requester

SYNOPSIS
    Request(Requester, Window);

FUNCTION
    Links in and displays a Requester into the specified Window.

    This routine ignores the Window's REQVERIFY flag.

INPUTS
    Requester =
        pointer to the Requester to be displayed

    Window =
        pointer to the Window into which this Requester goes

RESULT
    If the Requester is successfully opened, TRUE is returned. Otherwise, if the Requester
    could not be opened, FALSE is returned.

BUGS
    None

SEE ALSO
    None

NAME
        RethinkDisplay -- the grand manipulator of the entire Intuition display

SYNOPSIS
        RethinkDisplay();

FUNCTION
        This function performs the Intuition global display reconstruction. This includes: massaging internal state data, rethinking about all of the ViewPorts and their relationship to one another, and, finally, reconstructing the entire display based on the results of all this rethinking.

        The reconstruction of the display includes calls to the graphics library to perform MrgCop() and LoadView() for all of Intuition's Screens.

        You may perform a MakeScreen() on your Custom Screen before calling this routine. The results will be incorporated in the new display.

        WARNING: This routine can take several milliseconds to run, so do not use it lightly. RethinkDisplay() does a Forbid() on entry and a Permit() on exit, which can seriously degrade the performance of the multi-tasking Executive.

INPUTS
        None

RESULT
        None

BUGS
        None

SEE ALSO
        RemakeDisplay()
        The graphics library's MakeVPort(), MrgCop(), and LoadView()

NAME
   ScreenToBack -- send the specified Screen to the back of the display

SYNOPSIS
   ScreenToBack(Screen);

FUNCTION
   Sends the specified Screen to the back of the display

INPUTS
   Screen =
         pointer to a Screen structure

RESULT
   None

BUGS
   None

SEE ALSO
   None

NAME
>   ScreenToFront -- brings the specified Screen to the front of the display

SYNOPSIS
>   ScreenToFront(Screen);

FUNCTION
>   Brings the specified Screen to the front of the display

INPUTS
>   Screen =
>>   a pointer to a Screen structure

RESULT
>   None

BUGS
>   None

SEE ALSO
>   None

NAME
    SetDMRequest -- sets the DMRequest of the Window

SYNOPSIS
    SetDMRequest(Window, DMRequester);

FUNCTION
    Attempts to set the DMRequester into the specified window. The DMRequester is the
    special Requester that you attach to the double-click of the menu button which the user
    can then bring up on demand. This routine WILL NOT set the DMRequester if it's al-
    ready set and is currently active (in use by the user). After having called SetDMRe-
    quest(), if you want to change the DMRequester, the correct way to start is by calling
    ClearDMRequest() until it returns a value of TRUE; then you can call SetDMRequest()
    with the new DMRequester.

INPUTS
    Window =
            pointer to the window from which the DMRequest is to be set

    DMRequester =
            a pointer to a Requester

RESULT
    If the current DMRequest was not in use, sets the DMRequest
    pointer into the Window and returns TRUE.
    If the DMRequest was currently in use, doesn't change the pointer
    and returns FALSE.

BUGS
    None

SEE ALSO
    ClearDMRequest()
    Request()

NAME

SetMenuStrip -- Attaches the Menu strip to the Window

SYNOPSIS

SetMenuStrip(Window, Menu);

FUNCTION

Attaches the Menu strip to the Window. After calling this routine, if the user presses the menu button, this specified menu strip will be displayed and accessible.

NOTE: You should always design your Menu strip changes to be a two-way operation, where for every Menu strip you add to your Window you should always plan to clear that strip sometime. Even in the simplest case, where you will have just one Menu strip for the lifetime of your Window, you should always clear the Menu strip before closing the Window. If you already have a Menu strip attached to this Window, the correct procedure for changing to a new Menu strip involves calling ClearMenuStrip() to clear the old first. The sequence of events should be:

- OpenWindow()
- zero or more iterations of:
  - SetMenuStrip()
  - ClearMenuStrip()
- CloseWindow()

INPUTS

Window =
pointer to a Window structure

Menu =
pointer to the first Menu in the Menu strip

RESULT

None

BUGS

None

SEE ALSO

ClearMenuStrip()

NAME
SetPointer -- sets a Window with its own Pointer

SYNOPSIS
SetPointer(Window, Pointer, Height, Width, XOffset, YOffset);

FUNCTION
Sets up the Window with the sprite definition for the Pointer. Then whenever the Window is the active one, the Pointer image will change to its version of the Pointer. If the Window is the active one when this routine is called, the change takes place immediately.

The XOffset and YOffset are used to offset the top-left corner of the hardware sprite imagery from what Intuition regards as the current position of the Pointer. Another way of describing it is as the offset from the "hot spot" of the Pointer to the top-left corner of the sprite. For instance, if you specify offsets of zero, zero, then the top-left corner of your sprite image will be placed at the Pointer position. On the other hand, if you specify an XOffset of -7 (remember, sprites are 16 pixels wide) then your sprite will be centered over the Pointer position. If you specify an XOffset of -15, the right-edge of the sprite will be over the Pointer position.

INPUTS
Window =
> pointer to the Window to receive this Pointer definition

Pointer =
> pointer to the data definition of a Sprite

Height =
> the height of the Pointer

Width =
> the Width of the sprite (must be less than or equal to sixteen)

XOffset =
> the offset for your sprite from the Pointer position

YOffset =
> the offset for your sprite from the Pointer position

RESULT
None

BUGS
None

SEE ALSO
ClearPointer()

NAME
SetWindowTitles -- Sets the Window's titles for both Window and Screen

SYNOPSIS
SetWindowTitles(Window, WindowTitle, ScreenTitle);

FUNCTION
Allows you to set the text which appears in the Window and/or Screen title bars.

The Window Title appears at all times along the Window Title Bar. The Window's Screen Title appears at the Screen Title Bar whenever this Screen is the active one.

When this routine is called, your Window Title will be changed immediately. If your Window is the active one when this routine is called, the Screen Title will be changed immediately.

You can specify a value of -1 (negative one) for either of the title pointers. This designates that you want to Intuition to leave the current setting of that particular title alone, and modify only the other one. Of course, you could set both to -1.

Furthermore, you can set a value of 0 (zero) for either of the title pointers. Doing so specifies that you want no title to appear (the title bar will be blank).

INPUTS
Window =
> pointer to your Window structure

WindowTitle =
> pointer to a null-terminated text string, or set to either the value of -1 (negative one) or 0 (zero)

ScreenTitle =
> pointer to a null-terminated text string, or set to either the value of -1 (negative one) or 0 (zero)

RESULT
None

BUGS
None

SEE ALSO
OpenWindow()

NAME
>   ShowTitle -- Set the Screen title bar display mode

SYNOPSIS
>   ShowTitle(Screen, ShowIt);

FUNCTION
>   This routine sets the SHOWTITLE flag of the specified Screen, and then coordinates the redisplay of the Screen and its Windows.
>
>   The Screen title bar can appear either in front of or behind BACKDROP Windows. This is contrasted with the fact that non-BACKDROP Windows always appear in front of the Screen Title Bar. You specify whether you want the Screen Title Bar to be in front of or behind the Screen's BACKDROP Windows by calling this routine.
>
>   The ShowIt argument should be set to either TRUE or FALSE. If TRUE, the Screen's Title Bar will be shown in front of BACKDROP Windows. If FALSE, the Title Bar will be rendered behind all Windows.
>
>   When a Screen is first opened, the default setting of the SHOWTITLE flag is TRUE.

INPUTS
>   Screen =
>>   pointer to a Screen structure
>
>   ShowIt =
>>   Boolean TRUE or FALSE describing whether to show or hide the Screen Title Bar

RESULT
>   None

BUGS
>   None

SEE ALSO
>   None

Intuition Version 29.4

NAME

SizeWindow -- Ask Intuition to size a Window

SYNOPSIS

SizeWindow(Window, DeltaX, DeltaY);

FUNCTION

This routine sends a request to Intuition asking to size the Window the specified amounts. The delta arguments describe how much to size the Window along the respective axes.

Note that the Window will not be sized immediately, but rather will be sized the next time Intuition receives an input event, which happens currently at a minimum rate of ten times per second, and a maximum of sixty times a second. You can discover when you Window has finally been sized by setting the NEWSIZE flag of the IDCMP of your Window. See the "Input and Output Methods" chapter of this book for description of the IDCMP.

This routine does no error-checking. If your delta values specify some far corner of the Universe, Intuition will attempt to size your Window to the far corners of the Universe. Because of the distortions in the space-time continuum that can result from this, as predicted by special relativity, the result is generally not a pretty sight.

INPUTS

Window =

pointer to the structure of the Window to be sized

DeltaX =

signed value describing how much to size the Window on the x-axis

DeltaY =

signed value describing how much to size the Window on the y-axis

RESULT

None

BUGS

None

SEE ALSO

MoveWindow()

WindowToFront()

WindowToBack()

NAME
> ViewAddress -- returns the address of the Intuition View structure

SYNOPSIS
> ViewAddress();

FUNCTION
> Returns the address of the Intuition View structure. If you want to use any of the
> graphics, text, or animation primitives in your Window and that primitive requires a
> pointer to a View, this routine will return the address of the View for you.

INPUTS
> None

RESULT
> Returns the address of the Intuition View structure

BUGS
> Would be hard for this routine to have a bug

SEE ALSO
> All of the graphics, text, and animation primitives

NAME
ViewPortAddress -- returns the address of a Window's ViewPort structure

SYNOPSIS
ViewPortAddress(Window);

FUNCTION
Returns the address of the ViewPort associated with the specified Window. The ViewPort is actually the ViewPort of the Screen within which the Window is displayed. If you want to use any of the graphics, text, or animation primitives in your Window and that primitive requires a pointer to a ViewPort, you can use this call.

INPUTS
Window =
pointer to the Window for which you want the ViewPort address

RESULT
Returns the address of the Intuition View structure

BUGS
Would be hard for this routine to have a bug

SEE ALSO
All of the graphics, text, and animation primitives

NAME
>   WBenchToBack -- Sends the WorkBench Screen in back of all Screens

SYNOPSIS
>   WBenchToBack();

FUNCTION
>   Causes the WorkBench Screen, if it's currently opened, to go to the background. This
>   does not 'move' the Screen up or down, instead only affects the depth-arrangement of
>   the Screen.
>
>   If the WorkBench Screen was opened, this function returns TRUE, otherwise it returns
>   FALSE.

INPUTS
>   None

RESULT
>   If the WorkBench Screen was opened, this function returns TRUE, otherwise it returns
>   FALSE.

BUGS
>   None

SEE ALSO
>   WBenchToFront()

NAME
> WBenchToFront -- Brings the WorkBench Screen in front of all Screens

SYNOPSIS
> WBenchToFront();

FUNCTION
> Causes the WorkBench Screen, if it's currently opened, to come to the foreground. This does not 'move' the Screen up or down, instead only affects the depth-arrangement of the Screen.
>
> If the WorkBench Screen was opened, this function returns TRUE, otherwise it returns FALSE.

INPUTS
> None

RESULT
> If the WorkBench Screen was opened, this function returns TRUE, otherwise it returns FALSE.

BUGS
> None

SEE ALSO
> WBenchToBack()

NAME
>   WindowLimits -- Set the minimum and maximum limits of the Window

SYNOPSIS
>   WindowLimits(Window, MinWidth, MinHeight, MaxWidth, MaxHeight);

FUNCTION
>   Sets the minimum and maximum limits of the Window's size. Until this routine is
>   called, the Window's size limits are equal to the Window's initial size, which means that
>   the user won't be able to size it at all. After the call to this routine, the Window will be
>   able to be sized to any dimensions within the specified limits.
>
>   If you don't want to change any one of the dimensions, set the limit argument for that
>   dimension to zero. If any of the limit arguments is equal to zero, that argument is ig-
>   nored and the initial setting of that parameter remains undisturbed.
>
>   If any of the arguments is out of range (minimums greater than the current size, max-
>   imums less than the current size), that limit will be ignored, though the others will still
>   take effect if they are in range. If any are out of range, the return value from this pro-
>   cedure will be FALSE. If all arguments are valid, the return value will be TRUE.
>
>   If the user is currently sizing this Window, the new limits will not take effect until after
>   the sizing is completed.

INPUTS
>   Window =
>>   pointer to a Window structure
>
>   MinWidth, MinHeight, MaxWidth, MaxHeight =
>>   the new limits for the size of this Window. If any of these is set to zero, it will
>>   be ignored and that setting will be unchanged.

RESULT
>   Returns TRUE if everything was in order. If any of the parameters was out of range
>   (minimums greater than current size, maximums less than current size), FALSE is re-
>   turned and the errant limit request is not fulfilled (though the valid ones will be).

BUGS
>   None

SEE ALSO
>   None

NAME
WindowToBack -- Ask Intuition to send this Window to the back

SYNOPSIS
WindowToBack(Window);

FUNCTION
This routine sends a request to Intuition asking to send the Window in back of all other Windows in the Screen.

Note that the Window will not be depth-arranged immediately, but rather will be arranged the next time Intuition receives an input event, which happens currently at a minimum rate of ten times per second, and a maximum of sixty times a second.

Remember that BACKDROP Windows cannot be depth-arranged.

INPUTS
Window =
> pointer to the structure of the Window to be sent to the back

RESULT
None

BUGS
None

SEE ALSO
MoveWindow(), SizeWindow(), WindowToFront()

NAME
WindowToFront -- Ask Intuition to bring this Window to the front

SYNOPSIS
WindowToFront(Window);

FUNCTION
This routine sends a request to Intuition asking to bring the Window in front of all other Windows in the Screen.

Note that the Window will not be depth-arranged immediately, but rather will be arranged the next time Intuition receives an input event, which happens currently at a minimum rate of ten times per second, and a maximum of sixty times a second.

Remember that BACKDROP Windows cannot be depth-arranged.

INPUTS
Window =
  pointer to the structure of the Window to be brought to front

RESULT
None

BUGS
None

SEE ALSO
MoveWindow()
SizeWindow()
WindowToBack()

Appendix B

INTUITION INCLUDE FILE

This appendix contains a printout of the Intuition "include" file, which contains the definitions of all the Intuition data types and structures, constants, and macros. You include this file in all Intuition-based applications.

```
#ifndef INTUITION_INTUITION_H
#define INTUITION_INTUITION_H TRUE

/*** intuition.h ********************************************************
 *
 *   intuition.h main include for c programmers
 *
 *   Confidential Information: Commodore-Amiga Computer, Inc.
 *   Copyright (c) Commodore-Amiga Computer, Inc.
 *                               Modification History
 *      date    :   author :   Comments
 *      ------      ------     ----------------------------------------
 *      1-30-85     -=RJ=-     created this file!
 *
 ***********************************************************************
 * CONFIDENTIAL and PROPRIETARY
 * Copyright (C) 1985, COMMODORE-AMIGA, INC.
 * All Rights Reserved
 **********************************************************************/

#ifndef INTUITION_INTUITIONBASE_H
#include "intuition/intuitionbase.h"
#endif

#ifndef GRAPHICS_GFX_H
#include "graphics/gfx.h"
#endif

#ifndef GRAPHICS_CLIP_H
#include "graphics/clip.h"
#endif

#ifndef GRAPHICS_VIEW_H
#include "graphics/view.h"
#endif

#ifndef GRAPHICS_RASTPORT_H
#include "graphics/rastport.h"
#endif

#ifndef GRAPHICS_LAYERS_H
#include "graphics/layers.h"
#endif

#ifndef GRAPHICS_TEXT_H
#include "graphics/text.h"
#endif

#ifndef EXEC_PORTS_H
#include "exec/ports.h"
#endif

#ifndef DEVICES_TIMER_H
#include "devices/timer.h"
#endif
```

```
#ifndef DEVICES_INPUTEVENT_H
#include "devices/inputevent.h"
#endif


/* ====================================================================== */
/* === Menu ============================================================= */
/* ====================================================================== */
struct Menu
{
    struct Menu *NextMenu;        /* same level */
    SHORT LeftEdge, TopEdge;      /* dimensions of the select box */
    SHORT Width, Height;          /* dimensions of the select box */
    USHORT Flags;                 /* see flag definitions below */
    BYTE *MenuName;               /* text for this Menu Header */
    struct MenuItem *FirstItem;   /* pointer to first in chain */

    /* these mysteriously-named variables are for internal use only */
    SHORT JazzX, JazzY, BeatX, BeatY;
};


/* FLAGS SET BY BOTH THE APPLIPROG AND INTUITION */
#define MENUENABLED 0x0001        /* whether or not this menu is enabled */

/* FLAGS SET BY INTUITION */
#define MIDRAWN 0x0100            /* this menu's items are currently drawn */




/* ====================================================================== */
/* === MenuItem ========================================================= */
/* ====================================================================== */
struct MenuItem
{
    struct MenuItem *NextItem;    /* pointer to next in chained list */
    SHORT LeftEdge, TopEdge;      /* dimensions of the select box */
    SHORT Width, Height;          /* dimensions of the select box */
    USHORT Flags;                 /* see the defines below */

    LONG MutualExclude;           /* set bits mean this item excludes that */

    APTR ItemFill;                /* points to Image, IntuiText, or NULL */

    /* when this item is pointed to by the cursor and the items highlight
     *  mode HIGHIMAGE is selected, this alternate image will be displayed
     */
    APTR SelectFill;              /* points to Image, IntuiText, or NULL */

    BYTE Command;                 /* only if appliprog sets the COMMSEQ flag */

    struct MenuItem *SubItem;     /* if non-zero, DrawMenu shows "->" */
```

```
      /* The NextSelect field represents the menu number of next selected
       *  item (when user has drag-selected several items)
       */
      USHORT NextSelect;
};


/* FLAGS SET BY THE APPLIPROG */
#define CHECKIT          0x0001  /* whether to check this item if selected */
#define ITEMTEXT         0x0002  /* set if textual, clear if graphical item */
#define COMMSEQ          0x0004  /* set if there's an command sequence */
#define ITEMENABLED      0x0010  /* set if this item is enabled */

/* these are the SPECIAL HIGHLIGHT FLAG state meanings */
#define HIGHFLAGS        0x00C0  /* see definitions below for these bits */
#define HIGHIMAGE        0x0000  /* use the user's "select image" */
#define HIGHCOMP         0x0040  /* highlight by complementing the selectbox */
#define HIGHBOX          0x0080  /* highlight by "boxing" the selectbox */
#define HIGHNONE         0x00C0  /* don't highlight */

/* FLAGS SET BY BOTH APPLIPROG AND INTUITION */
#define CHECKED          0x0100  /* if CHECKIT, then set this when selected */

/* FLAGS SET BY INTUITION */
#define ISDRAWN          0x1000  /* this item's subs are currently drawn */
#define HIGHITEM         0x2000  /* this item is currently highlighted */




/* ============================================================= */
/* === Requester ================================================ */
/* ============================================================= */
struct Requester
{
      /* the ClipRect and BitMap and used for rendering the requester */
      struct Requester *OlderRequest;
      SHORT LeftEdge, TopEdge;              /* dimensions of the entire box */
      SHORT Width, Height;                  /* dimensions of the entire box */
      SHORT RelLeft, RelTop;                /* for Pointer relativity offsets */

      struct Gadget *ReqGadget;             /* pointer to a list of Gadgets */
      struct Border *ReqBorder;             /* the box's border */
      struct IntuiText *ReqText;            /* the box's text */
      USHORT Flags;                         /* see definitions below */

      /* pen number for back-plane fill before draws */
      UBYTE BackFill;
      struct ClipRect ReqCRect;

      /* If the BitMap plane pointers are non-zero, this tells the system
       * that the image comes pre-drawn (if the appliprog wants to define
       * it's own box, in any shape or size it wants!);  this is OK by
```

```
        * Intuition as long as there's a good correspondence between
        * the image and the specified Gadgets
        */
     struct BitMap *ImageBMap;    /* points to the BitMap of PREDRAWN imagery */
     struct BitMap ReqBMap;
};


/* FLAGS SET BY THE APPLIPROG */
#define POINTREL 0x0001  /* if POINTREL set, TopLeft is relative to pointer*/
#define PREDRAWN 0x0002  /* if ReqBMap points to predrawn Requester imagery */
/* FLAGS SET BY BOTH THE APPLIPROG AND INTUITION */

/* FLAGS SET BY INTUITION */
#define REQOFFWINDOW     0x1000  /* part of one of the Gadgets was offwindow */
#define REQACTIVE        0x2000  /* this requester is active */
#define SYSREQUEST       0x4000  /* this requester caused by system */
#define DEFERREFRESH     0x8000  /* this Requester stops a Refresh broadcast */




/* ============================================================= */
/* === Gadget ================================================== */
/* ============================================================= */
struct Gadget
{
        struct Gadget *NextGadget;  /* next gadget in the list */

        SHORT LeftEdge, TopEdge;    /* "hit box" of gadget */
        SHORT Width, Height;        /* "hit box" of gadget */

        USHORT Flags;               /* see below for list of defines */

        USHORT Activation;          /* see below for list of defines */

        USHORT GadgetType;          /* see below for defines */

        /* appliprog can specify that the Gadget be rendered as either as Border
         * or an Image.  This variable points to which (or equals NULL if there's
         * nothing to be rendered about this Gadget)
         */
        APTR GadgetRender;

        /* appliprog can specify "highlighted" imagery rather than algorithmic
         * this can point to either Border or Image data
         */
        APTR SelectRender;

        struct IntuiText *GadgetText;   /* text for this gadget */

        /* by using the MutualExclude word, the appliprog can describe
         * which gadgets mutually-exclude which other ones.  The bits
         * in MutualExclude correspond to the gadgets in object containing
```

```
     * the gadget list.  If this gadget is selected and a bit is set
     * in this gadget's MutualExclude and the gadget corresponding to
     * that bit is currently selected (e.g. bit 2 set and gadget 2
     * is currently selected) that gadget must be unselected.
     * Intuition does the visual unselecting (with checkmarks) and
     * leaves it up to the program to unselect internally
     */
    LONG MutualExclude;   /* set bits mean this gadget excludes that gadget */

    /* pointer to a structure of special data required by Proportional,
     * String and Integer Gadgets
     */
    APTR SpecialInfo;

    USHORT GadgetID;      /* user-definable ID field */
    APTR UserData;        /* ptr to general purpose User data (ignored by In) */
};


/* --- FLAGS SET BY THE APPLIPROG --------------------------------------------- */
/* combinations in these bits describe the highlight technique to be used */
#define GADGHIGHBITS    0x0003
#define GADGHCOMP       0x0000  /* Complement the select box */
#define GADGHBOX        0x0001  /* Draw a box around the image */
#define GADGHIMAGE      0x0002  /* Blast in this alternate image */
#define GADGHNONE       0x0003  /* don't highlight */

/* set this flag if the GadgetRender and SelectRender point to Image imagery,
 * clear if it's a Border
 */
#define GADGIMAGE 0x0004

/* combinations in these next two bits specify to which corner the gadget's
 *  Left & Top coordinates are relative.  If relative to Top/Left,
 *  these are "normal" coordinates (everything is relative to something in
 *  this universe)
 */
#define GRELBOTTOM      0x0008  /* set if rel to bottom, clear if rel top */
#define GRELRIGHT       0x0010  /* set if rel to right, clear if to left */
/* set the RELWIDTH bit to spec that Width is relative to width of screen */
#define GRELWIDTH       0x0020
/* set the RELHEIGHT bit to spec that Height is rel to height of screen */
#define GRELHEIGHT      0x0040

/* the SELECTED flag is initialized by you and set by Intuition.  It
 * specifies whether or not this Gadget is currently selected/highlighted
 */
#define SELECTED        0x0080


/* the GADGDISABLED flag is initialized by you and later set by Intuition
 * according to your calls to On/OffGadget().  It specifies whether or not
 * this Gadget is currently disabled from being selected
 */
#define GADGDISABLED    0x0100
```

```
/* --- These are the Activation flag bits --------------------------------- */
/* RELVERIFY is set if you want to verify that the pointer was still over
 *   the gadget when the select button was released
 */
#define RELVERIFY          0x0001

/* the flag GADGIMMEDIATE, when set, informs the caller that the gadget
 *   was activated when it was activated.  this flag works in conjunction with
 *   the RELVERIFY flag
 */
#define GADGIMMEDIATE      0x0002

/* the flag ENDGADGET, when set, tells the system that this gadget, when
 *  selected, causes the Requester or AbsMessage to be ended.  Requesters or
 *   AbsMessages that are ended are erased and unlinked from the system */
#define ENDGADGET          0x0004

/* the FOLLOWMOUSE flag, when set, specifies that you want to receive
 * reports on mouse movements (ie, you want the REPORTMOUSE function for
 * your Window).  When the Gadget is deselected (immediately if you have
 * no RELVERIFY) the previous state of the REPORTMOUSE flag is restored
 * You probably want to set the GADGIMMEDIATE flag when using FOLLOWMOUSE,
 * since that's the only reasonable way you have of learning why Intuition
 * is suddenly sending you a stream of mouse movement events.  If you don't
 * set RELVERIFY, you'll get at least one Mouse Position event.
 */
#define FOLLOWMOUSE        0x0008

/* if any of the BORDER flags are set in a Gadget that's included in the
 * Gadget list when a Window is opened, the corresponding Border will
 * be adjusted to make room for the Gadget
 */
#define RIGHTBORDER        0x0010
#define LEFTBORDER         0x0020
#define TOPBORDER          0x0040
#define BOTTOMBORDER       0x0080

#define TOGGLESELECT       0x0100  /* this bit for toggle-select mode */

#define STRINGCENTER       0x0200  /* should be a StringInfo flag, but it's OK */
#define STRINGRIGHT        0x0400  /* should be a StringInfo flag, but it's OK */

#define LONGINT            0x0800  /* this String Gadget is actually LONG Int */

#define ALTKEYMAP          0x1000  /* this String has an alternate keymap */


/* --- GADGET TYPES ------------------------------------------------------- */
/* These are the Gadget Type definitions for the variable GadgetType
 * gadget number type MUST start from one.  NO TYPES OF ZERO ALLOWED.
 * first comes the mask for Gadget flags reserved for Gadget typing
 */
#define GADGETTYPE         0xFC00  /* all Gadget Global Type flags (padded) */
#define SYSGADGET          0x8000  /* 1 = SysGadget, 0 = AppliGadget */
#define SCRGADGET          0x4000  /* 1 = ScreenGadget, 0 = WindowGadget */
```

```
#define GZZGADGET          0x2000   /* 1 = Gadget for GIMMEZEROZERO borders */
#define REQGADGET          0x1000   /* 1 = this is a Requester Gadget */
/* system gadgets */
#define SIZING             0x0010
#define WDRAGGING          0x0020
#define SDRAGGING          0x0030
#define WUPFRONT           0x0040
#define SUPFRONT           0x0050
#define WDOWNBACK          0x0060
#define SDOWNBACK          0x0070
#define CLOSE              0x0080
/* application gadgets */
#define BOOLGADGET         0x0001
#define GADGET0002         0x0002
#define PROPGADGET         0x0003
#define STRGADGET          0x0004
```

```
/* ================================================================
/* === PropInfo ====================================================
/* ================================================================
/* this is the special data required by the proportional Gadget
 * typically, this data will be pointed to by the Gadget variable SpecialI
 */
struct PropInfo
{
        USHORT Flags;          /* general purpose flag bits (see defines below) *

        /* You initialize the Pot variables before the Gadget is added to
         * the system.  Then you can look here for the current settings
         * any time, even while User is playing with this Gadget.  To
         * adjust these after the Gadget is added to the System, use
         * ModifyProp();  The Pots are the actual proportional settings,
         * where a value of zero means zero and a value of MAXPOT means
         * that the Gadget is set to its maximum setting.
         */
        USHORT HorizPot;       /* 16-bit FixedPoint horizontal quantity percentag
        USHORT VertPot;        /* 16-bit FixedPoint vertical quantity percentage

        /* the 16-bit FixedPoint Body variables describe what percentage of
         * the entire body of stuff referred to by this Gadget is actually
         * shown at one time.  This is used with the AUTOKNOB routines,
         * to adjust the size of the AUTOKNOB according to how much of
         * the data can be seen.  This is also used to decide how far
         * to advance the Pots when User hits the Container of the Gadget.
         * For instance, if you were controlling the display of a 5-line
         * Window of text with this Gadget, and there was a total of 15
         * lines that could be displayed, you would set the VertBody value to
         *      (MAXBODY / (TotalLines / DisplayLines)) = MAXBODY / 3.
         * Therefore, the AUTOKNOB would fill 1/3 of the container, and
         * if User hits the Cotainer outside of the knob, the pot would
         * advance 1/3 (plus or minus) If there's no body to show, or
```

```
 * the total amount of displayable info is less than the display area,
 * set the Body variables to the MAX.  To adjust these after the
 * Gadget is added to the System, use ModifyProp();
 */
USHORT HorizBody;               /* horizontal Body */
USHORT VertBody;                /* vertical Body */

/* these are the variables that Intuition sets and maintains */
USHORT CWidth;          /* Container width (with any relativity absolute) */
USHORT CHeight;         /* Container height (with any relativity absolute) */
USHORT HPotRes, VPotRes; /* pot increments */
USHORT LeftBorder;              /* Container borders */
USHORT TopBorder;               /* Container borders */
};


/* --- FLAG BITS ----------------------------------------------------------- */
#define AUTOKNOB        0x0001  /* this flag sez:  gimme that old auto-knob */
#define FREEHORIZ       0x0002  /* if set, the knob can move horizontally */
#define FREEVERT        0x0004  /* if set, the knob can move vertically */
#define PROPBORDERLESS  0x0008  /* if set, no border will be rendered */
#define KNOBHIT         0x0100  /* set when this Knob is hit */

#define KNOBHMIN        6       /* minimum horizontal size of the Knob */
#define KNOBVMIN        4       /* minimum vertical size of the Knob */
#define MAXBODY         0xFFFF  /* maximum body value */
#define MAXPOT          0xFFFF  /* maximum pot value */




/* ======================================================================= */
/* === StringInfo ======================================================== */
/* ======================================================================= */
/* this is the special data required by the string Gadget
 * typically, this data will be pointed to by the Gadget variable SpecialInfo
 */
struct StringInfo
{
    /* you initialize these variables, and then Intuition maintains them */
    UBYTE *Buffer;          /* the buffer containing the start and final string */
    UBYTE *UndoBuffer;      /* optional buffer for undoing current entry */
    SHORT BufferPos;        /* character position in Buffer */
    SHORT MaxChars;         /* max number of chars in Buffer (including NULL) */
    SHORT DispPos;          /* Buffer position of first displayed character */

    /* Intuition initializes and maintains these variables for you */
    SHORT UndoPos;          /* character position in the undo buffer */
    SHORT NumChars;         /* number of characters currently in Buffer */
    SHORT DispCount;        /* number of whole characters visible in Container */
    SHORT CLeft, CTop;      /* topleft offset of the container */
    struct Layer *LayerPtr;         /* the RastPort containing this Gadget */
    LONG LongInt;
```

```
        /* If you want this Gadget to use your own Console keymapping, you
         * set the ALTKEYMAP bit in the Activation flags of the Gadget, and then
         * set this variable to point to your keymap.  If you don't set the
         * ALTKEYMAP, you'll get the standard ASCII keymapping.
         */
        struct KeyMap *AltKeyMap;
};


/* ================================================================ */
/* === IntuiText ================================================== */
/* ================================================================ */
/* IntuiText is a series of strings that start with a screen location
 *  (always relative to the upper-left corner of something) and then the
 *  text of the string.  The text is null-terminated.
 */
struct IntuiText
{
    UBYTE FrontPen, BackPen;      /* the pen numbers for the rendering */
    UBYTE DrawMode;               /* the mode for rendering the text */
    SHORT LeftEdge;               /* relative start location for the text */
    SHORT TopEdge;                /* relative start location for the text */
    struct TextAttr *ITextFont;   /* if NULL, you accept the default */
    UBYTE *IText;                 /* pointer to null-terminated text */
    struct IntuiText *NextText;   /* continuation to TxWrite another text */
};


/* ================================================================ */
/* === Border ===================================================== */
/* ================================================================ */
/* Data type Border, used for drawing a series of lines which is intended for
 *  use as a border drawing, but which may, in fact, be used to render any
 *  arbitrary vector shape.
 *  The routine DrawBorder sets up the RastPort with the appropriate
 *  variables, then does a Move to the first coordinate, then does Draws
 *  to the subsequent coordinates.
 *  After all the Draws are done, if NextBorder is non-zero we call DrawBorder
 *  recursively
 */
struct Border
{
    SHORT LeftEdge, TopEdge;      /* initial offsets from the origin */
    UBYTE FrontPen, BackPen;      /* pens numbers for rendering */
    UBYTE DrawMode;               /* mode for rendering */
    BYTE Count;                   /* number of XY pairs */
    SHORT *XY;                    /* vector coordinate pairs rel to LeftTop*/
    struct Border *NextBorder;    /* pointer to any other Border too */
```

```
};
```

```
/* ================================================================= */
/* === Image ======================================================= */
/* ================================================================= */
/* This is a brief image structure for very simple transfers of
 * image data to a RastPort
 */
struct Image
{
    SHORT LeftEdge;             /* starting offset relative to some origin */
    SHORT TopEdge;              /* starting offsets relative to some origin */
    SHORT Width;                /* pixel size (though data is word-aligned) */
    SHORT Height, Depth;        /* pixel sizes */
    USHORT *ImageData;          /* pointer to the actual word-aligned bits */

    /* the PlanePick and PlaneOnOff variables work much the same way as the
     * equivalent GELS Bob variables.  It's a space-saving
     * mechanism for image data.  Rather than defining the image data
     * for every plane of the RastPort, you need define data only
     * for the planes that are not entirely zero or one.  As you
     * define your Imagery, you will often find that most of the planes
     * ARE just as color selectors.  For instance, if you're designing
     * a two-color Gadget to use colors two and three, and the Gadget
     * will reside in a five-plane display, bit plane zero of your
     * imagery would be all ones, bit plane one would have data that
     * describes the imagery, and bit planes two through four would be
     * all zeroes.  Using these flags allows you to avoid wasting all
     * that memory in this way:  first, you specify which planes you
     * want your data to appear in using the PlanePick variable.  For
     * each bit set in the variable, the next "plane" of your image
     * data is blitted to the display.  For each bit clear in this
     * variable, the corresponding bit in PlaneOnOff is examined.
     * If that bit is clear, a "plane" of zeroes will be used.
     * If the bit is set, ones will go out instead.  So, for our example:
     *    Gadget.PlanePick = 0x02;
     *    Gadget.PlaneOnOff = 0x01;
     * Note that this also allows for generic Gadgets, like the
     * System Gadgets, which will work in any number of bit planes.
     * Note also that if you want an Image that is only a filled
     * rectangle, you can get this by setting PlanePick to zero
     * (pick no planes of data) and set PlaneOnOff to describe the pen
     * color of the rectangle.
     */
    UBYTE PlanePick, PlaneOnOff;

    /* if the NextImage variable is not NULL, Intuition presumes that
     * it points to another Image structure with another Image to be
     * rendered
     */
    struct Image *NextImage;
```

```
};




/* ============================================================= */
/* === IntuiMessage ============================================ */
/* ============================================================= */
struct IntuiMessage
{
    struct Message ExecMessage;

    /* the Class bits correspond directly with the IDCMP Flags, except for the
     * special bit LONELYMESSAGE (defined below)
     */
    ULONG Class;

    /* the Code field is for special values like MENU number */
    USHORT Code;

    /* the Qualifier field is a copy of the current InputEvent's Qualifier */
    USHORT Qualifier;

    /* IAddress contains particular addresses for Intuition functions, like
     * the pointer to the Gadget or the Screen
     */
    APTR IAddress;

    /* when getting mouse movement reports, any event you get will have the
     * the mouse coordinates in these variables.  the coordinates are relative
     * to the upper-left corner of your Window (GIMMEZEROZERO notwithstanding)
     */
    SHORT MouseX, MouseY;

    /* the time values are copies of the current system clock time.  Micros
     * are in units of microseconds, Seconds in seconds.
     */
    ULONG Seconds, Micros;

    /* the IDCMPWindow variable will always have the address of the Window of
     * this IDCMP
     */
    struct Window *IDCMPWindow;

    /* system-use variable */
    struct IntuiMessage *SpecialLink;
};


/* --- IDCMP Classes ------------------------------------------------------ */
#define SIZEVERIFY      0x00000001  /* See the Programmer's Guide */
#define NEWSIZE         0x00000002  /* See the Programmer's Guide */
#define REFRESHWINDOW   0x00000004  /* See the Programmer's Guide */
#define MOUSEBUTTONS    0x00000008  /* See the Programmer's Guide */
```

```
#define MOUSEMOVE          0x00000010      /* See the Programmer's Guide          */
#define GADGETDOWN         0x00000020      /* See the Programmer's Guide          */
#define GADGETUP           0x00000040      /* See the Programmer's Guide          */
#define REQSET             0x00000080      /* See the Programmer's Guide          */
#define MENUPICK           0x00000100      /* See the Programmer's Guide          */
#define CLOSEWINDOW        0x00000200      /* See the Programmer's Guide          */
#define RAWKEY             0x00000400      /* See the Programmer's Guide          */
#define REQVERIFY          0x00000800      /* See the Programmer's Guide          */
#define REQCLEAR           0x00001000      /* See the Programmer's Guide          */
#define MENUVERIFY         0x00002000      /* See the Programmer's Guide          */
#define NEWPREFS           0x00004000      /* See the Programmer's Guide          */
#define DISKINSERTED       0x00008000      /* See the Programmer's Guide          */
#define DISKREMOVED        0x00010000      /* See the Programmer's Guide          */
#define WBENCHMESSAGE      0x00020000      /* See the Programmer's Guide          */
#define ACTIVEWINDOW       0x00040000      /* See the Programmer's Guide          */
#define INACTIVEWINDOW     0x00080000      /* See the Programmer's Guide          */
#define DELTAMOVE          0x00100000      /* See the Programmer's Guide          */
/* NOTEZ-BIEN:            0x80000000 is reserved for internal use by IDCMP       */

/* the IDCMP Flags do not use this special bit, which is cleared when
 * Intuition sends its special message to the Task, and set when Intuition
 * gets its Message back from the Task.  Therefore, I can check here to
 * find out fast whether or not this Message is available for me to send
 */
#define LONELYMESSAGE      0x80000000


/* --- IDCMP Codes ------------------------------------------------------------ */
/* This group of codes is for the MENUVERIFY function */
#define MENUHOT            0x0001  /* IntuiWants verification or MENUCANCEL       */
#define MENUCANCEL         0x0002  /* HOT Reply of this cancels Menu operation    */
#define MENUWAITING        0x0003  /* Intuition simply wants a ReplyMsg() ASAP    */

/* This group of codes is for the WBENCHMESSAGE messages */
#define WBENCHOPEN         0x0001
#define WBENCHCLOSE        0x0002



/* ========================================================================= */
/* === Window ============================================================== */
/* ========================================================================= */
struct Window
{
     struct Window *NextWindow;            /* for the linked list in a screen */

     SHORT LeftEdge, TopEdge;              /* screen dimensions of window */
     SHORT Width, Height;                  /* screen dimensions of window */

     SHORT MouseY, MouseX;                 /* relative to upper-left of window */

     SHORT MinWidth, MinHeight;            /* minimum sizes */
     SHORT MaxWidth, MaxHeight;            /* maximum sizes */

     ULONG Flags;                          /* see below for defines */
```

```
        struct Menu *MenuStrip;                /* the strip of Menu headers */

        UBYTE *Title;                          /* the title text for this window */

        struct Requester *FirstRequest;        /* all active Requesters */

        struct Requester *DMRequest;           /* double-click Requester */

        SHORT ReqCount;                        /* count of reqs blocking Window */

        struct Screen *WScreen;                /* this Window's Screen */
        struct RastPort *RPort;                /* this Window's very own RastPort */

        /* the border variables describe the window border.  If you specify
         * GIMMEZEROZERO when you open the window, then the upper-left of the
         * ClipRect for this window will be upper-left of the BitMap (with correct
         * offsets when in SuperBitMap mode; you MUST select GIMMEZEROZERO when
         * using SuperBitMap).  If you don't specify ZeroZero, then you save
         * memory (no allocation of RastPort, Layer, ClipRect and associated
         * Bitmaps), but you also must offset all your writes by BorderTop,
         * BorderLeft and do your own mini-clipping to prevent writing over the
         * system gadgets
         */
        BYTE BorderLeft, BorderTop, BorderRight, BorderBottom;
        struct RastPort *BorderRPort;


        /* You supply a linked-list of Gadgets for your Window.
         * This list DOES NOT include system gadgets.  You get the standard
         * window system gadgets by setting flag-bits in the variable Flags (see
         * the bit definitions below)
         */
        struct Gadget *FirstGadget;

        /* these are for opening/closing the windows */
        struct Window *Parent, *Descendant;

        /* sprite data information for your own Pointer
         * set these AFTER you Open the Window by calling SetPointer()
         */
        USHORT *Pointer;      /* sprite data */
        BYTE PtrHeight;       /* sprite height (not including sprite padding) */
        BYTE PtrWidth;        /* sprite width (must be less than or equal to 16) */
        BYTE XOffset, YOffset;      /* sprite offsets */

        /* the IDCMP Flags and User's and Intuition's Message Ports */
        ULONG IDCMPFlags;     /* User-selected flags */
        struct MsgPort *UserPort, *WindowPort;
        struct IntuiMessage *MessageKey;

        UBYTE DetailPen, BlockPen;  /* for bar/border/gadget rendering */

        /* the CheckMark is a pointer to the imagery that will be used when
         * rendering MenuItems of this Window that want to be checkmarked
         * if this is equal to NULL, you'll get the default imagery
         */
```

```
    struct Image *CheckMark;

    UBYTE *ScreenTitle; /* if non-null, Screen title when Window is active */

    /* These variables have the mouse coordinates relative to the
     * inner-Window of GIMMEZEROZERO Windows.  This is compared with the
     * MouseX and MouseY variables, which contain the mouse coordinates
     * relative to the upper-left corner of the Window, GIMMEZEROZERO
     * notwithstanding
     */
    SHORT GZZMouseX;
    SHORT GZZMouseY;
    /* these variables contain the width and height of the inner-Window of
     * GIMMEZEROZERO Windows
     */
    SHORT GZZWidth;
    SHORT GZZHeight;

    UBYTE *ExtData;

    BYTE *UserData;       /* general-purpose pointer to User data extension */

};


/* --- FLAGS REQUESTED (NOT DIRECTLY SET THOUGH) BY THE APPLIPROG --------- */
#define WINDOWSIZING    0x0001   /* include sizing system-gadget? */
#define WINDOWDRAG      0x0002   /* include dragging system-gadget? */
#define WINDOWDEPTH     0x0004   /* include depth arrangement gadget? */
#define WINDOWCLOSE     0x0008   /* include close-box system-gadget? */

#define SIZEBRIGHT      0x0010  /* size gadget uses right border */
#define SIZEBBOTTOM     0x0020  /* size gadget uses bottom border */

/* --- refresh modes -------------------------------------------------- */
/* combinations of the REFRESHBITS select the refresh type */
#define REFRESHBITS     0x00C0
#define SMART_REFRESH   0x0000
#define SIMPLE_REFRESH  0x0040
#define SUPER_BITMAP    0x0080
#define OTHER_REFRESH   0x00C0

#define BACKDROP        0x0100  /* this is an ever-popular BACKDROP window */

#define REPORTMOUSE     0x0200  /* set this to hear about every mouse move */

#define GIMMEZEROZERO   0x0400  /* make extra border stuff */

#define BORDERLESS      0x0800  /* set this to get a Window sans border */

#define ACTIVATE        0x1000  /* when Window opens, it's the Active one */

/* FLAGS SET BY INTUITION */
#define WINDOWACTIVE  0x2000      /* this window is the active one */
#define INREQUEST     0x4000      /* this window is in request mode */
#define MENUSTATE     0x8000      /* this Window is active with its Menus on */
```

```
/* --- Other User Flags ------------------------------------------------------- */
#define RMBTRAP         0x00010000      /* Catch RMB events for your own */
#define NOCAREREFRESH   0x00020000      /* not to be bothered with REFRESH */


/* --- Other Intuition Flags -------------------------------------------------- */
#define WINDOWREFRESH   0x01000000      /* Window is currently refreshing */
#define WBENCHWINDOW    0x02000000      /* WorkBench Window */

#define SUPER_UNUSED    0xFCFC0000      /* bits of Flag unused yet */


/* --- see struct IntuiMessage for the IDCMP Flag definitions ------------ */




/* ============================================================================ */
/* === NewWindow ============================================================== */
/* ============================================================================ */
struct NewWindow
{
    SHORT LeftEdge, TopEdge;            /* screen dimensions of window */
    SHORT Width, Height;               /* screen dimensions of window */

    UBYTE DetailPen, BlockPen;         /* for bar/border/gadget rendering */

    ULONG IDCMPFlags;                  /* User-selected IDCMP flags */

    ULONG Flags;                       /* see Window struct for defines */

    /* You supply a linked-list of Gadgets for your Window.
     *  This list DOES NOT include system Gadgets.  You get the standard
     *  system Window Gadgets by setting flag-bits in the variable Flags (see
     *  the bit definitions under the Window structure definition)
     */
    struct Gadget *FirstGadget;

    /* the CheckMark is a pointer to the imagery that will be used when
     * rendering MenuItems of this Window that want to be checkmarked
     * if this is equal to NULL, you'll get the default imagery
     */
    struct Image *CheckMark;

    UBYTE *Title;                              /* the title text for this window */

    /* the Screen pointer is used only if you've defined a CUSTOMSCREEN and
     * want this Window to open in it.  If so, you pass the address of the
     * Custom Screen structure in this variable.  Otherwise, this variable
     * is ignored and doesn't have to be initialized.
     */
    struct Screen *Screen;

    /* SUPER_BITMAP Window?  If so, put the address of your BitMap structure
     * in this variable.  If not, this variable is ignored and doesn't have
```

```
      * to be initialized
      */
     struct BitMap *BitMap;

     /* the values describe the minimum and maximum sizes of your Windows.
      * these matter only if you've chosen the WINDOWSIZING Gadget option,
      * which means that you want to let the User to change the size of
      * this Window.  You describe the minimum and maximum sizes that the
      * Window can grow by setting these variables.  You can initialize
      * any one these to zero, which will mean that you want to duplicate
      * the setting for that dimension (if MinWidth == 0, MinWidth will be
      * set to the opening Width of the Window).
      * You can change these settings later using SetWindowLimits().
      * If you haven't asked for a SIZING Gadget, you don't have to
      * initialize any of these variables.
      */
     SHORT MinWidth, MinHeight;        /* minimums */
     SHORT MaxWidth, MaxHeight;        /* maximums */

     /* the type variable describes the Screen in which you want this Window to
      * open.  The type value can either be CUSTOMSCREEN or one of the
      * system standard Screen Types such as WBENCHSCREEN.  See the
      * type definitions under the Screen structure
      */
     USHORT Type;
};
```


```
/* ============================================================ */
/* === Screen ================================================= */
/* ============================================================ */
struct Screen
{
     struct Screen *NextScreen;        /* linked list of screens */
     struct Window *FirstWindow;       /* linked list Screen's Windows */

     SHORT LeftEdge, TopEdge;          /* parameters of the screen */
     SHORT Width, Height;              /* parameters of the screen */

     SHORT MouseY, MouseX;             /* position relative to upper-left */

     USHORT Flags;                     /* see definitions below */

     UBYTE *Title;                     /* null-terminated Title text */
     UBYTE *DefaultTitle;              /* for Windows without ScreenTitle */

     /* Bar sizes for this Screen and all Window's in this Screen */
     BYTE BarHeight, BarVBorder, BarHBorder, MenuVBorder, MenuHBorder;
     BYTE WBorTop, WBorLeft, WBorRight, WBorBottom;

     struct TextAttr *Font;            /* this screen's default font */
```

```
        /* the display data structures for this Screen */
        struct ViewPort ViewPort;          /* describing the Screen's display */
        struct RastPort RastPort;          /* describing Screen rendering */
        struct BitMap BitMap;              /* auxiliary graphexcess baggage */
        struct Layer_Info LayerInfo;       /* each screen gets a LayerInfo */

        /* You supply a linked-list of Gadgets for your Screen.
         *  This list DOES NOT include system Gadgets.  You get the standard
         *  system Screen Gadgets by default
         */
        struct Gadget *FirstGadget;

        UBYTE DetailPen, BlockPen;         /* for bar/border/gadget rendering */

        /* the following variable(s) are maintained by Intuition to support the
         * DisplayBeep() color flashing technique
         */
        USHORT SaveColor0;

        /* This layer is for the Screen and Menu bars */
        struct Layer *BarLayer;

        UBYTE *ExtData;

        UBYTE *UserData;     /* general-purpose pointer to User data extension */
};


/* --- FLAGS SET BY INTUITION ------------------------------------------ */
/* The SCREENTYPE bits are reserved for describing various Screen types
 * available under Intuition.
 */
#define SCREENTYPE       0x000F  /* all the screens types available */
/* --- the definitions for the Screen Type ---------------------------- */
#define WBENCHSCREEN     0x0001  /* Ta Da!  The Workbench */
#define CUSTOMSCREEN     0x000F  /* for that special look */

#define SHOWTITLE        0x0010  /* this gets set by a call to ShowTitle() */

#define BEEPING          0x0020  /* set when Screen is beeping */

#define CUSTOMBITMAP     0x0040  /* if you are supplying your own BitMap */




/* ==================================================================== */
/* === NewScreen ====================================================== */
/* ==================================================================== */
struct NewScreen
{
     SHORT LeftEdge, TopEdge, Width, Height, Depth;  /* screen dimensions */

     UBYTE DetailPen, BlockPen;             /* for bar/border/gadget rendering */

     USHORT ViewModes;            /* the Modes for the ViewPort (and View) */
```

```
     USHORT Type;                      /* the Screen type (see defines below) */

     struct TextAttr *Font;            /* this Screen's default text attributes */

     UBYTE *DefaultTitle;              /* the default title for this Screen */

     struct Gadget *Gadgets;           /* your own Gadgets for this Screen */

     /* if you are opening a CUSTOMSCREEN and already have a BitMap
      * that you want used for your Screen, you set the flags CUSTOMBITMAP in
      * the Types variable and you set this variable to point to your BitMap
      * structure.  The structure will be copied into your Screen structure,
      * after which you may discard your own BitMap if you want
      */
     struct BitMap *CustomBitMap;
};




/* =========================================================================== */
/* === Preferences =========================================================== */
/* =========================================================================== */

/* these are the definitions for the printer configurations */
#define FILENAME_SIZE   30                          /* Filename size */

#define POINTERSIZE (1 + 16 + 1) * 2     /* Size of Pointer data buffer */

/* These defines are for the default font size.  These actually describe the
 * height of the defaults fonts.  The default font type is the topaz
 * font, which is a fixed width font that can be used in either
 * eighty-column or sixty-column mode.  The Preferences structure reflects
 * which is currently selected by the value found in the variable FontSize,
 * which may have either of the values defined below.  These values actually
 * are used to select the height of the default font.  By changing the
 * height, the resolution of the font changes as well.
 */
#define TOPAZ_EIGHTY 8
#define TOPAZ_SIXTY 9


struct Preferences
{
     /* the default font height */
     BYTE FontHeight;                       /* height for system default font  */

     /* constant describing what's hooked up to the port */
     UBYTE PrinterPort;                      /* printer port connection         */

     /* the baud rate of the port */
     USHORT BaudRate;                        /* baud rate for the serial port   */

     /* various timing rates */
     struct timeval KeyRptSpeed;             /* repeat speed for keyboard       */
```

```
        struct timeval KeyRptDelay;               /* Delay before keys repeat      */
        struct timeval DoubleClick;               /* Interval allowed between clicks */

        /* Intuition Pointer data */
        USHORT PointerMatrix[POINTERSIZE];        /* Definition of pointer sprite   */
        BYTE XOffset;                             /* X-Offset for active 'bit'      */
        BYTE YOffset;                             /* Y-Offset for active 'bit'      */
        USHORT color17;                           /**********************************/
        USHORT color18;                           /* Colours for sprite pointer     */
        USHORT color19;                           /**********************************/
        USHORT PointerTicks;                      /* Sensitivity of the pointer     */

        /* Workbench Screen colors */
        USHORT color0;                            /**********************************/
        USHORT color1;                            /*   Standard default colours     */
        USHORT color2;                            /*    Used in the Workbench        */
        USHORT color3;                            /**********************************/

        /* positioning data for the Intuition View */
        BYTE ViewXOffset;                         /* Offset for top lefthand corner */
        BYTE ViewYOffset;                         /* X and Y dimensions             */
        WORD ViewInitX, ViewInitY;                /* View initial offset values     */

        BOOL EnableCLI;                           /* CLI availability switch */

        /* printer configurations */
        USHORT PrinterType;                       /* printer type                   */
        UBYTE PrinterFilename[FILENAME_SIZE];/* file for printer           */

        /* print format and quality configurations */
        USHORT PrintPitch;                        /* print pitch                    */
        USHORT PrintQuality;                      /* print quality                  */
        USHORT PrintSpacing;                      /* number of lines per inch       */
        UWORD PrintLeftMargin;                    /* left margin in characters      */
        UWORD PrintRightMargin;                   /* right margin in characters     */
        USHORT PrintImage;                        /* positive or negative           */
        USHORT PrintAspect;                       /* horizontal or vertical         */
        USHORT PrintShade;                        /* b&w, half-tone, or color       */
        WORD PrintThreshold;                      /* darkness ctrl for b/w dumps    */

        /* print paper descriptors */
        USHORT PaperSize;                         /* paper size                     */
        UWORD PaperLength;                        /* paper length in number of lines */
        USHORT PaperType;                         /* continuous or single sheet     */

        BYTE padding[50];                         /* For further system expansion   */
};


/* PrinterPort */
#define PARALLEL_PRINTER 0x00
#define SERIAL_PRINTER   0x01


/* BaudRate */
#define BAUD_110        0x00
#define BAUD_300        0x01
```

```
#define BAUD_1200        0x02
#define BAUD_2400        0x03
#define BAUD_4800        0x04
#define BAUD_9600        0x05
#define BAUD_19200       0x06
#define BAUD_MIDI        0x07

/* PaperType */
#define FANFOLD          0x00
#define SINGLE           0x80

/* PrintPitch */
#define PICA             0x000
#define ELITE            0x400
#define FINE             0x800

/* PrintQuality */
#define DRAFT            0x000
#define LETTER           0x100

,* PrintSpacing */
#define SIX_LPI          0x000
#define EIGHT_LPI        0x200

/* Print Image */
#define IMAGE_POSITIVE   0x00
#define IMAGE_NEGATIVE   0x01

/* PrintAspect */
#define ASPECT_HORIZ     0x00
#define ASPECT_VERT      0x01

/* PrintShade */
#define SHADE_BW         0x00
#define SHADE_GREYSCALE  0x01
#define SHADE_COLOR      0x02

/* PaperSize */
#define US_LETTER        0x00
#define US_LEGAL         0x10
#define N_TRACTOR        0x20
#define W_TRACTOR        0x30
#define CUSTOM           0x40

/* PrinterType */
#define CUSTOM_NAME      0x00
#define ALPHA_P_101      0x01
#define BROTHER_15XL     0x02
#define CBM_MPS1000      0x03
#define DIAB_630         0x04
#define DIAB_ADV_D25     0x05
#define DIAB_C_150       0x06
#define EPSON            0x07
#define EPSON_JX_80      0x08
#define OKIMATE_20       0x09
#define QUME_LP_20       0x0A
```

```
/********************************************************************
**       Written by      Jon Prince  (Commodore Business Machines (UK) Ltd.) **
**          &            Barry Walsh (Commodore Business Machines (UK) Ltd.) **
**                                                                  **
**       Intuitioned by  -=RJMical=- (Commodore-Amiga Inc, don'cha know)     **
********************************************************************/




/* ================================================================= */
/* === Remember ==================================================== */
/* ================================================================= */
/* this structure is used for remembering what memory has been allocated to
 * date by a given routine, so that a premature abort or systematic exit
 * can deallocate memory cleanly, easily, and completely
 */
struct Remember
{
    struct Remember *NextRemember;
    ULONG RememberSize;
    UBYTE *Memory;
};




/* ================================================================= */
/* === Miscellaneous =============================================== */
/* ================================================================= */

/* = MACROS ======================================================== */
#define MENUNUM(n)  (n & Ox1F)
#define ITEMNUM(n)  ((n >> 5) & OxOO3F)
#define SUBNUM(n)  ((n >> 11) & OxOO1F)

#define SHIFTMENU(n)  (n & Ox1F)
#define SHIFTITEM(n)  ((n & Ox3F) << 5)
#define SHIFTSUB(n)  ((n & Ox1F) << 11)

/* = MENU STUFF ==================================================== */
#define NOMENU OxOO1F
#define NOITEM OxOO3F
#define NOSUB  OxOO1F
#define MENUNULL OxFFFF


/* = =RJ='s peculiarities ========================================== */
#define FOREVER for(;;)
#define SIGN(x) ( ((x) > O) - ((x) < O) )
#define NOT !

/* these defines are for the COMMSEQ and CHECKIT menu stuff.  If CHECKIT,
 * I'll use a generic Width (for all resolutions) for the CheckMark.
```

```
 * If COMMSEQ, likewise I'll use this generic stuff
 */
#define CHECKWIDTH        19
#define COMMWIDTH         27
#define LOWCHECKWIDTH     13
#define LOWCOMMWIDTH      16


/* these are the AlertNumber defines.  if you are calling DisplayAlert()
 * the AlertNumber you supply must have the ALERT_TYPE bits set to one
 * of these patterns
 */
#define ALERT_TYPE        0x80000000
#define RECOVERY_ALERT    0x00000000      /* the system can recover from this */
#define DEADEND_ALERT     0x80000000      /* no recovery possible, this is it */


/* When you're defining IntuiText for the Positive and Negative Gadgets
 * created by a call to AutoRequest(), these defines will get you
 * reasonable-looking text.  The only field without a define is the IText
 * field; you decide what text goes with the Gadget
 */
#define AUTOFRONTPEN      0
#define AUTOBACKPEN       1
#define AUTODRAWMODE      JAM2
#define AUTOLEFTEDGE      6
#define AUTOTOPEDGE       3
#define AUTOITEXTFONT     NULL
#define AUTONEXTTEXT      NULL


/* --- RAWMOUSE Codes and Qualifiers (Console OR IDCMP) ------------------- */
#define SELECTUP          (IECODE_LBUTTON | IECODE_UP_PREFIX)
#define SELECTDOWN        (IECODE_LBUTTON)
#define MENUUP            (IECODE_RBUTTON | IECODE_UP_PREFIX)
#define MENUDOWN          (IECODE_RBUTTON)
#define ALTLEFT           (IEQUALIFIER_LALT)
#define ALTRIGHT          (IEQUALIFIER_RALT)
#define AMIGALEFT         (IEQUALIFIER_LCOMMAND)
#define AMIGARIGHT        (IEQUALIFIER_RCOMMAND)
#define AMIGAKEYS         (AMIGALEFT | AMIGARIGHT)

#define CURSORUP          0x4C
#define CURSORLEFT        0x4F
#define CURSORRIGHT       0x4E
#define CURSORDOWN        0x4D
#define KEYCODE_Q         0x10
#define KEYCODE_N         0x36
#define KEYCODE_M         0x37



#endif
```

Appendix C

INTERNAL PROCEDURES

This appendix discusses the more esoteric and internal Intuition functions. These functions are definitely *not* for the casual user. Using these functions can seriously alter the user's environment, which is potentially a hazardous thing to do. You have more leeway when using these functions in an machine environment where you've taken complete control of the Amiga and do not intend to allow other tasks to co-exist with yours. However, if you intend to have your program run in the multitasking environment, please use these routines very thoughtfully, since the effects you can cause on other people's programs and on the user's understanding of the normal course of events can be dramatic at best, and can cause serious loss of data and the user's confidence in using the Amiga.

With that caveat aside, here's a list of the functions covered in this appendix:

SetPrefs()
> This routine allows you to set Intuition's internal state of the Preferences.

AlohaWorkbench()
> This routine allows the Workbench tool to make its presence and departure known to Intuition.

Intuition()
> This is the main entry point into Intuition, where input events arrive and are dispatched.

SetPrefs()

This routine configures Intuition's internal data states according to the specified Preferences structure. Normally, this routine is called only by:

o   the Preferences program itself after the user has changed the Preferences. The Preferences program also saves the user's Preferences data into a disk file named devs/system-configuration.

o   AmigaDOS when the system is being booted up. AmigaDOS opens the devs/system-configuration file and passes the information there to the *SetPrefs()* routine. This way, the user can create an environment and have that environment restored every time the system is booted.

Note that the intended use for the *SetPrefs()* call is entirely to serve the user. You should never use this routine to make your programming or design job easier at the cost of yanking the rug out from beneath the user.

The synopsis of this function is:

> *SetPrefs(Preferences, Size, RealThing)*
>> *Preferences* - a pointer to a Preferences structure
>>
>> *Size* - the number of bytes contained in your Preferences structure. Typically, you will use "sizeof(struct Preferences)" for this argument.
>>
>> *RealThing* - a Boolean TRUE or FALSE designating whether or not this is an intermediate or final version of the Preferences. The difference is that final changes to Intuition's preferences causes a global broadcast of NEWPREFS events to everyone who's listening for this event. Intermediate changes may be used, for instance, to update the screen colors while the user is playing with the color gadgets.

Refer to Chapter 11, "Other Features", for information about the Preferences structure and the standard Preferences procedure calls.

AlohaWorkbench()

In Hawaiian, "aloha" means both hello and goodbye. The *AlohaWorkbench()* routine allows the Workbench program to inform Intuition that it's become active and that it's shutting down.

If the Workbench program is active, Intuition is able to tell it to open and close its windows when someone uses the Intuition *OpenWorkBench()* and *CloseWorkBench()* functions to open or close the Workbench screen. If the Workbench program is not active, presumably it has no opened windows, so there is no need for this communication.

This routine is called with an argument that is either a pointer to an initialized message port which designates that Workbench is active and communications can take place, or NULL to designate that the Workbench tool is shutting down.

When the message port is active, Intuition will send IntuiMessages to the port. The messages will have the *Class* field set to WBENCHMESSAGE. The *Code* field will equal either WBEN-CHOPEN or WBENCHCLOSE, depending on whether the Workbench application should open or close its windows. Intuition assumes that Workbench will comply, so as soon as the message is replied to, Intuition proceeds with the expectation that the windows have been opened or closed accordingly.

The procedure synopsis is:

AlohaWorkbench(WBPort)

        WBPort - a pointer to an initialized MsgPort structure where the special communications are to take place

Intuition()

This is Intuition's main entry point. All of Intuition's I/O operations originate here. The input stream flows into Intuition at this portal.

This routine accepts a single argument: a pointer to a linked list of InputEvent structures. These events have all the real-time state information that Intuition needs to create its art. Refer to the *Amiga ROM Kernel Manual* for more information about InputEvent structure and the operation of the Input Device.

When *Intuition()* exits, it returns a pointer to a linked list of InputEvent structures. This list of InputEvents has no dependable correspondence to the list that was initially submitted to *Intuition()*. Intuition may add events to the list and extract events from the list. This list of events is normally intended for the Console Device.

If you are considering feeding false input events to Intuition, please think again. And then think about it one more time. If you are running in an environment where you've taken over the machine, it's probably safe to fool Intuition in a controlled way. But if you are running in a multi-tasking environment, especially where the Input Device is still feeding input events directly into the stream, you can easily cause more harm than good. You may not be able to anticipate the things that could go wrong when other programs try to exist in an environment that you are modifying.

An important note is that *Intuition()* is sometimes required to call the Exec *Wait()* function. Normally, *Intuition()* is called from within the Input Device's task, so the Input Device enters the wait state when these situations arise. If you call *Intuition()* directly, your task may have to wait. The obvious problem with this the classic lockout problem where your task can't create the required response because your task has forced itself to wait, which will cause the system to freeze. The best way to get around this is to have a separate task that calls *Intuition()* and does nothing more.

The synopsis of this function is:

    Intuition(InputEvent)

        InputEvent - a pointer to the first in a linked list of InputEvent structures

GLOSSARY

active screen

The screen containing the active window.

active window

The window receiving user input. Only one window is active at a time.

alert

Information exchange device displayed by the system or the application when serious problems occur or when immediate action is necessary.

ALT keys

Two command keys on the keyboard to the left and right of the Amiga keys.

alternate

An image or border used in gadget highlighting. When the gadget is selected, the alternate image or border is substituted for the original image or border.

Amiga keys

Two command keys on the keyboard to the left and right of the space bar.

AmigaDOS

The Amiga disk operating system.

application gadget

A custom gadget created by the developer.

auto-knob

The special automatic knob for proportional gadgets; changes its shape according to the current proportional settings.

Backdrop window

A window that stays anchored to the back of the display.

bit-map

The complete definition of a display in memory, consisting of one or more bit-planes and information about how to organize the rectangular display.

bit-plane

A contiguous series of memory words, treated as if it were a rectangular shape.

body variables	Proportional gadget variables that contain the increment by which the pot variables may change.
Boolean gadget	A simple yes-or-no gadget.
border area	The area containing border gadgets.
border line	The default double-line drawn around the perimeter of all windows, except the Borderless window.
Borderless window	A window with no drawn border lines.
buffer	An area of continuous memory, typically used for storing blocks of data such as text strings.
checkmark	A small image that appears next to a menu item showing that the user has selected that item. By default, the checkmark is √ , but a custom image can be substituted.

CLI = Command Line Interpreter

click	To quickly press and release a mouse button.
ClipBoard	Workbench file used to store the last data cut (removed) from a project.
clipping	Causing a graphical rendering to appear only in some bounded area, such as only within the non-concealed areas of a window.
close	To remove a window or screen from the display.
close gadget	Gadget in the upper left corner of a screen or window that the user selects to request that a window or screen be closed.
color indirection	The method used by Amiga for coloring individual pixels, in which the binary number formed from all the bits that define a given pixel refers to one of the 32 color registers. Each of the 32 color registers can be set equal to any of 4096 colors.
color palette	The set of colors available in a screen.
color register	One of 32 hardware registers containing colors that you can define.
column	A set of adjoining pixels that form a vertical line on the video display.

command keys	Keys that combine with alphanumeric keys to create command key sequences, which substitute for making selections with the mouse buttons.
Command Line Interpreter	The command line interface to system commands and utilities.
complement	The binary complement of a color, used as a method of gadget highlighting and in flashing the screen. To complement a binary number means to change all the 1's to 0's and all the 0's to 1's.
Console Device	A communication path for both user input and program output. Especially recommended for input/output of text-only applications.
container	Part of a proportional gadget; the area within which the knob or slider can move; the select box of the gadget.
control escape sequence	Special sequences of characters that start with the "Escape" character.
controller	Hardware device, such as mouse or light pen, used to move the pointer or furnish some other input.
coordinates	A pair of numbers shown in the form, (x,y), where x is an offset from the left side of the display or display component and y is an offset from the top.
Copper	Display-synchronized coprocessor that handles the Amiga video display.
coprocessor = Copper	
cursor keys	The arrow keys, which can be used as a substitute for using the mouse to move the pointer.
data structure	The grouping together of the components required to define some data element.
depth	Number of bit-planes in a display.
depth-arrangement gadgets	Gadgets in the title bar of a screen or window used to send the screen or window to the back of the display or bring it up front.
disable	To make something unavailable to the user.

display	Put up a screen, window, requester, alert, or any other graphics object on the video display.
display field	One complete scanning of the video beam from top to bottom of the video display screen.
display memory	The RAM that contains the information for the display imagery; the hardware translates the contents of the display memory into video signals.
display modes	Display parameters set in the definition of a screen. The modes are high or low horizontal resolution, interlace or non-interlace vertical resolution, sprite mode, and dual playfield mode.
double-click	To quickly press and release a mouse button twice.
double-menu requester	A requester that the user can open by double-clicking the mouse menu button.
drag	To move an icon, gadget, window, or screen by placing the pointer over the object to moved and holding down the selection button while moving the mouse.
drag gadget	The portion of a window or screen title bar that contains no other gadgets, used for moving a window or screen around on the video display.
dual playfield mode	A display mode that allows you to manage two separate display memories, giving you two separately controllable displays at the same time.
edit menu	A menu for text processing that includes various text editing functions.
enable	To make something available to the user; a menu item or gadget that is enabled can be selected by the user.
Exec	Low level primitives that comprise the Amiga multi-tasking operating system.
extended selection	A technique for selecting more than one menu item at a time.
fill	To put a color or pattern within an enclosed area.
flag	A mechanism for selecting an option or detecting a state; a name representing a bit to be set or cleared.

font	A set of letters, numbers, and symbols that share the same basic design.
gadget	Any of the control devices provided within a window, screen, or requester; employed by users to change what is being displayed or to communicate with an application or with Intuition.
ghost	Display less distinctly (overlay an area with a faint pattern of dots) to indicate that something, such as a gadget or a window, is not available or not active.
ghost shape	The new outline of a window that shows briefly when the user is dragging or sizing a window.
Gimmezerozero window	A window with a separate bit-map for the window border.
header file	A file that is included at the beginning of a C program and contains definitions of data types and structures, constants, and macros.
high resolution	A horizontal display mode where there are 640 pixels displayed across a horizontal line.
highlight	To modify the display of a selected menu item or gadget in a way that distinguishes it from its non-selected state.
hit select	A method of gadget selection where the gadget is unselected as soon as the select button is released.
hold-and-modify	A display mode that gives you extended color selection — up to 4,096 colors on the screen at one time.
hue	The characteristic of a color that is determined by the color's position in the color spectrum.
icon	A visual representation of an object in the Workbench, such as a program, file, or disk.
IDCMP	"Intuition Direct Communications Message Ports"; the primary communication path for user input to an application. Gives mouse and keyboard events and Intuition events in raw form. Provides a path for communicating to Intuition.
initialize	To set up an Intuition component with certain default parameters.

input event	The message created by the Input Device whenever a signal is detected at one of the Amiga input ports.
interlace	A vertical display mode where 400 lines are displayed from top to bottom of the video display.
IntuiMessage	The input message created by Intuition for application programs; the message is the medium in this case.
keymap	Translation table used by the Console Device to translate keycodes into normal characters.
knob	Part of a proportional gadget; the user manipulates the knob to set a proportional value.
library	A collection of pre-defined functions that can be used by any program.
linked list	A collection of like objects linked together by having a pointer variable in one contain the address of the next; the last object in the list has a next-pointer of NULL.
low resolution	A horizontal display mode where 320 pixels are displayed across a horizontal line.
menu	A category that has menu items associated with it. One of the entries in the menu list displayed in the screen title bar.
menu bar	A strip in the screen title bar that shows the menu list when the user holds down the menu button.
menu button	The right-hand button on the mouse.
menu item	One of the choices in a menu; the options presented to the user.
menu list	List of menus displayed in the screen title bar when the user holds down the menu button.
menu shortcut	An alternate way of choosing a menu item by pressing a key on the keyboard while holding down the right AMIGA key.
menu title = menu	
Message Ports	A software mechanism managed by the Amiga Exec which allows inter-task communications.

mouse	A controller device used to move the pointer and make selections.
multi-tasking	A system where many tasks can be operating at the same time, with no task forced to be aware of any other task.
mutual exclusion	Selecting a menu item (or gadget) can cause other menu items (or gadgets) to become deselected.
non-interlace	A display mode where 200 lines are displayed from top to bottom of the video display.
null-terminated	A text string must always end with a byte of zero.
offset	A position in the display that is relative to some other position.
open	For the user, to display a window. For an application, to display a window or screen.
option	A feature that, once selected, persists until it is deselected.
parallel port	A connector on the back of the Amiga used to attach printers and other add-ons.
pen	A variable containing a color register number used for drawing lines or filling background.
pixel	Short for "picture element". The smallest addressable element in the video display. Each pixel is one dot of color.
playfield	One of the basic elements in Amiga graphics; the background for all the other display elements.
pointer	A small object, usually an arrow, that moves on the display when the user moves the mouse (or the cursor keys). It is used to choose menu items, open windows, and drag and select other objects.
pot variables	Proportional gadget variables that contain the actual proportional values.
Preferences	A program that allows the user to change various settings of an Amiga.
preserve	To keep overlapped portions of the display in hidden memory buffers.

primitives	Amiga low-level library functions.
project menu	A menu for opening and saving project files.
proportional gadget	Gadget used to display a proportional value or get a proportional setting from the user. Consists of a knob or slider and a container.
RAM	Random access (volatile) memory.
raster	The area in memory where the bit-map is located.
RastPort	The data structure that defines the general parameters of a window or screen.
refresh	Recreate a display that was hidden and is now revealed.
render	To draw or write into display memory.
requester	A rectangular information exchange region in a window. When a requester appears, the user must select a gadget in the requester to close the requester before doing anything else in the window.
resolution	On a video display, the number of pixels that can be displayed in the horizontal and vertical directions.
screen	A full-width area of the display with a set color palette, resolution, and other display modes. Windows open in screens.
scroll	To move the contents of display memory within a window.
scroll bar	A proportional gadget with which the user can display different parts of the display memory.
select	To pick a gadget or menu item.
select box	The sensitive area of a gadget or menu item. When the user moves the pointer within a gadget's select box, the gadget becomes selected.
select button	The left-hand button on a mouse.
selected option	An option that is currently in effect.

selection shortcut	A quick way to select a gadget by pressing some key while holding down the left Amiga key.
serial port	A connector on the back of the Amiga used to attach modems and other serial add-ons.
shortcut	A quick way, from the keyboard, to choose a menu item or select a gadget.
simple refresh	A method of refreshing window display where concealed areas are redrawn by the program when they are revealed
size	To change the dimensions of a window or screen.
sizing gadget	A gadget for the user to change the size of a window or a screen.
slider	Part of a proportional gadget, used to pick a value within a range, by dragging the slider or by moving the slider by increments with clicks of a mouse button.
smart refresh	A method of refreshing window display where Intuition keeps information about concealed areas in off-display buffers and refreshes the display from this information. If the window is sized, the program may have to recreate the display.
sprite	Small, easily movable graphic object. You can have multiple sprites in a window at the same time.
sprite mode	A display mode that allows you to have sprites in your windows.
string gadget	Gadget that prompts the user to enter a text string or an integer.
structure = data structure	
sub-menu	An additional menu that appears when some menu items are chosen by the user.
SuperBitMap refresh	A method of window refresh where the display is recreated from a separate bit-map area.
SuperBitMap window	A window with its own bit-map; doesn't use the screen's bit-map.

system gadgets	Predefined gadgets for windows and screens; for screens, dragging and depth arranging; for windows; dragging, depth arranging, sizing, and closing.
task	Operating system module or application program. Each task appears to have full control over its own virtual 68000 machine.
text cursor	In programs containing text and in string gadgets, a marker that indicates your position in the text.
title bar	A strip at the top of a screen or window that contains gadgets and an optional name for the screen or window.
toggle select	A method of gadget selection where the gadget remains selected when the user releases the select button, and does not become deselected until the user picks it again.
tool	An application program.
Topaz	The default system font. It is a fixed-width font in two sizes: 60-columns wide and 8 lines tall; 80-columns wide and 9 lines tall.
transparent	A special color register definition that allows a background color to show through. Used in dual playfield mode.
typeface = font	
type style	A variation of a typeface, such as italic or bold.
undo	A text editing function that reverses an action.
UserPort	The message port created for you when you request IDCMP functionality. You receive messages from Intuition via this port.
vector	A line segment.
video display	Everything that appears on the screen of a video monitor or television.
View	The graphics library data structure used to create the Intuition display.
ViewPort	The graphics library data structure used to create and manage the Intuition screen.

virtual terminal   An Intuition window; it accepts input from the user and display
                     output from the application.

window   Rectangular display in a screen that accepts input from the user
                     and displays output from the application.

WindowPort   The message port created for you when you request IDCMP
                     functionality. You respond to messages from Intuition via this
                     port.

Workbench   A program to manipulate AmigaDOS disk file objects.

WorkBench screen   The primary Intuition screen.

Index