

U-47447

AMIGA

Intuition

Reference Manual

Commodore Business Machines, Inc.



AMIGA INTUITION REFERENCE MANUAL

Amiga Intuition Reference Manual

Robert J. Mical and Susan Deyl

Commodore Business Machines, Inc.

Amiga Technical Reference Series



Addison-Wesley Publishing Company, Inc.

Reading, Massachusetts Menlo Park, California Don Mills, Ontario
Wokingham, England Amsterdam Sydney Singapore Tokyo
Madrid Bogota Santiago San Juan *

ACKNOWLEDGMENTS

Cover by Jack Haeger
Illustrations by Sheryl Knowles Fuller, Jack Haeger, Al McCahon, and Robert J. Mical

Programs used for the illustrations by:

Andy Finkel
Neil Katin
Dale Luck
Robert J. Mical
Jon Prince
Barry Walsh.

Editing by Patria Brown and Embe Humphreys

Intuition by Robert J. Mical

DEDICATION

To Csfayn Day Havis, without whose strength and wisdom I could not have done this
— Bob

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. First, each chapter presents 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 multipurpose input devices that you can design and attach to your windows and requesters.

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. Using the graphics, border and text 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, *Intuition Function Calls*, contains a complete description of each Intuition function.

Appendix B, *Intuition Include File*, contains the Intuition include file.

Appendix C, *Internal Procedures*, 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*

Table of Contents

Chapter 1 INTRODUCTION.....	1
How the User Sees an Intuition Application.....	3
The Fight Approach to Using Intuition.....	7
Chapter 2 GETTING STARTED.....	9
Intuition Components.....	9
General Program Requirements and Information.....	10
SIMPLE PROGRAM: OPENING A WINDOW.....	11
~ SIMPLE PROGRAM: ADDING THE CLOSE GADGET.....	14
SIMPLE PROGRAM: ADDING THE REMAINING SYSTEM GADGETS.....	15
SIMPLE PROGRAM: OPENING A CUSTOM SCREEN.....	16
SIMPLE PROGRAM: THE FINAL VERSION.....	18
Chapter 3J SCREENS.....	23
About Screens.....	24
Standard Screens.....	27
WORKBENCH.....	28
Custom Screens.....	29
INTUITION-MANAGED CUSTOM SCREENS.....	30
APPLICATION-MANAGED CUSTOM SCREENS.....	30
Screen Characteristics.....	32
DISPLAYMODES.....	32
DEPTH AND COLOR.....	33
TYPE STYLES.....	34
HEIGHT, WIDTH, AND STARTING LOCATION.....	35
SCREEN TITLE.....	37
CUSTOM GADGETS.....	38
Using Custom Screens.....	38
NEWSCREEN STRUCTURE.....	39
SCREEN STRUCTURE.....	41
SCREEN FUNCTIONS.....	42
Chapter A WINDOWS.....	45
About Windows.....	46
WINDOW INPUT/OUTPUT.....	48
OPENING, ACTIVATING AND CLOSING WINDOWS.....	49
SPECIAL WINDOW TYPES.....	50

WINDOW GADGETS.....	53
WINDOW BORDERS.....	56
PRESERVING THE WINDOW DISPLAY.....	57
REFRESHING THE WINDOW DISPLAY.....	60
WINDOW POINTER.....	61
GRAPHICS AND TEXT IN WINDOWS.....	62
WINDOW COLORS.....	63
WINDOW DIMENSIONS.....	63
Using Windows.....	64
NEWWINDOW STRUCTURE.....	65
WINDOW STRUCTURE.....	70
WINDOW FUNCTIONS.....	71
SETTING UP A SUPERBITMAP WINDOW.....	75
SETTING UP A CUSTOM POINTER.....	76
Chapters GADGETS.....	81
About Gadgets.....	82
System Gadgets.....	83
SIZING GADGET.....	85
DEPTH-ARRANGEMENT GADGETS.....	85
DRAGGING GADGET.....	85
CLOSE GADGET.....	86
Application Gadgets.....	86
RENDERING GADGETS.....	87
USER SELECTION OF GADGETS.....	89
GADGET SELECT BOX.....	90
GADGET POINTER MOVEMENTS.....	91
GADGETS IN WINDOW BORDERS.....	92
MUTUAL EXCLUDE.....	92
GADGET HIGHLIGHTING.....	93
GADGET ENABLING AND DISABLING.....	94
BOOLEAN GADGET TYPE.....	94
PROPORTIONAL GADGET TYPE.....	95
STRING GADGET TYPE.....	99
INTEGER GADGET TYPE.....	102
COMBINING GADGET TYPES.....	102
Using Application Gadgets.....	103
GADGET STRUCTURE.....	104
FLAGS.....	107
ACTIVATION FLAGS.....	108
SPECIALINFO DATA STRUCTURES.....	110
GADGET FUNCTIONS.....	H4

Chapter 6 MENUS	117
About Menus.....	118
SUBMITTING AND REMOVING MENU STRIPS.....	120
ABOUT MENU ITEM BOXES.....	120
ACTION/ATTRIBUTE ITEMS AND THE CHECKMARK.....	122
MUTUAL EXCLUSION.....	123
COMMAND-KEY SEQUENCES AND IMAGERY.....	124
ENABLING AND DISABLING MENUS AND MENU ITEMS.....	125
CHANGING MENU STRIPS.....	126
MENU NUMBERS AND MENU SELECTION MESSAGES.....	126
HOW MENU NUMBERS REALLY WORK.....	128
INTERCEPTING NORMAL MENU OPERATIONS.....	129
REQUESTERS AS MENUS.....	130
Using Menus.....	131
MENU STRUCTURES.....	132
MENU FUNCTIONS.....	137
Chapter 7 REQUESTERS AND ALERTS	139
About Requesters.....	140
RENDERING REQUESTERS.....	143
REQUESTER DISPLAY POSITION.....	143
DOUBLE-MENU REQUESTERS.....	144
GADGETS IN REQUESTERS.....	144
IDCMP REQUESTER FEATURES.....	145
A SIMPLE, AUTOMATIC REQUESTER.....	145
Using Requesters.....	147
REQUESTER STRUCTURE.....	147
THE VERY EASY REQUESTER.....	151
REQUESTER FUNCTIONS.....	152
Alerts.....	154
Chapter 8 INPUT AND OUTPUT METHODS	157
An Overview of Input and Output.....	157
About Input and Output.....	159
Using the IDCMP.....	164
INTUIMESSAGES.....	165
IDCMP FLAGS.....	167
SETTING UP YOUR OWN IDCMP MONITOR TASK AND USER PORT.....	171
An Example of the IDCMP.....	172
Using the Console Device.....	173
USING THE AMIGAD08 CONSOLE.....	174
USING THE CONSOLE DEVICE DIRECTLY.....	174
SETTING THE KEYMAP.....	176

Chapter 9 IMAGES, LINE DRAWING, AND TEXT.....	179
Using Intuition Graphics.....	180
DISPLAYING BORDERS, INTUITEXT, AND IMAGES.....	181
CREATING BORDERS.....	181
CREATING TEXT.....	185
CREATING IMAGES.....	189
INTUITION GRAPHICS FUNCTIONS.....	199
Using the Amiga Graphics Primitives.....	200
Chapter 10 MOUSE AND KEYBOARD.....	203
About the Mouse.....	204
Mouse Messages.....	206
About the Keyboard.....	206
Using the Keyboard as an Alternate to the Mouse.....	208
Chapter 11 OTHER FEATURES.....	211
Easy Memory Allocation and Deallocation.....	211
INTUITION HELPS YOU REMEMBER.....	212
HOW TO REMEMBER.....	213
THE REMEMBER STRUCTURE.....	213
AN EXAMPLE OF REMEMBERING.....	214
Preferences.....	214
PREFERENCES STRUCTURE.....	217
PREFERENCES FUNCTIONS.....	220
Remaking the ViewPorts.....	220
Current Time Values.....	221
Flashing the Display.....	221
Using Sprites in Intuition Windows and Screens.....	222
Assembly Language Conventions.....	222
Chapter 12 STYLE.....	223
Menu Style.....	224
PROJECT MENUS.....	224
EDIT MENUS.....	225
Gadget Style.....	226
Requester Style.....	227
Command Key Style.....	227
Mouse Style.....	229
The Sides of Good and Bad.....	230
Miscellaneous Style Notes.....	230
A Final Note on Style.....	231
Appendix A Intuition Function Calls.....	A-1
Appendix B Intuition Include File	B-1

Appendix C Internal Procedures	C-1
Glossary.....	G-1
Index	1-1

V

AMIGA INTUITION REFERENCE MANUAL

Figures

Figure 1-1	A Screen with Windows.....	4
Figure 1-2	Menu Items and Subitems.....	5
Figure 1-3	A Requester.....	6
Figure 1-4	An Alert.....	7
Figure 2-1	A Simple Window.....	13
Figure 2-2	A Simple Window with Gadgets.....	16
Figure 2-3	Display Created by Intuition's "Hello World" Program.....	21
Figure 3-1	A Screen and Windows.....	25
Figure 3-2	Screen and Windows with Menu List Displayed.....	26
Figure 3-3	The Workbench Screen and the Workbench Application.....	28
Figure 3-4	Topaz Font in 60-column and 80-column Types.....	35
Figure 3-5	Acceptable Placement of Screens.....	36
Figure 3-6	Unacceptable Placement of Screens.....	37
Figure 4-1	A High-resolution Screen and Windows.....	47
Figure 4-2	System Gadgets for Windows.....	55
Figure 4-3	Simple Refresh.....	58
Figure 4-4	Smart Refresh.....	59
Figure 4-5	SuperBitMap Refresh.....	60
Figure 4-6	The X-Shaped Custom Pointer.....	79
Figure 5-1	System Gadgets in a Low-resolution Window.....	84
Figure 5-2	Hand-drawn Gadget — Unselected and Selected.....	87
Figure 5-3	Line-drawn Gadget — Unselected and Selected.....	88
Figure 5-4	Example of Combining Gadget Types.....	102
Figure 6-1	Screen with Menu Bar Displayed.....	119
Figure 6-2	Example Item Box.....	121
Figure 6-3	Example Subitem Box.....	122
Figure 6-4	Menu Items with Command Key Shortcuts.....	125
Figure 7-1	Requester Deluxe.....	140
Figure 7-2	A Simple Requester Made with AutoRequestQ.....	145
Figure 7-3	Th« "Out of Memory" Alert.....	154
Figure 8-1	Watching the Stream.....	158
Figure 8-2	Input from the IDCMP, Output through the Graphics Primitives.....	isi
Figure 8-3	Input and Output through the Console Device.....	162
Figure 8-4	Full-system Input and Output (a Busy Program).....	162
Figure 8-5	Output Only.....	153

Figure 9-1	Example of Border Relative Position.....	183
Figure 9-2	Intuition's High-resolution Sizing Gadget Image.....	191
Figure 9-3	Example of PlanePick and PlaneOnOff.....	194
Figure 9-4	Example Image — the Front Gadget.....	196
Figure 11-1	Intuition Remembering.....	212
Figure 11-2	Preferences Display.....	215
Figure 12-1	The Dreaded Erase-Disk Requester.....	226

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.

Intuition's 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 that 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 multitasking, multi-graphic, and sound synthesis capabilities. Under the Amiga Executive operating system, many programs can reside in memory at the same time, sharing the system's resources with one another. Intuition allows these programs to display their information in overlapping windows without interfering with one another; in addition, it 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 to 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*. When a virtual terminal is opened, 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, 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-play field mode, different color palettes, double-buffering, and more) plus animation and speech and music synthesis. You can combine sound, graphics, and animation in your Intuition windows. As you browse through this book, you'll find many creative ways to turn Intuition and the other Amiga tools into your 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 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 (see figure 1-1). 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 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 they can be 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 arranging the depth of the screens.

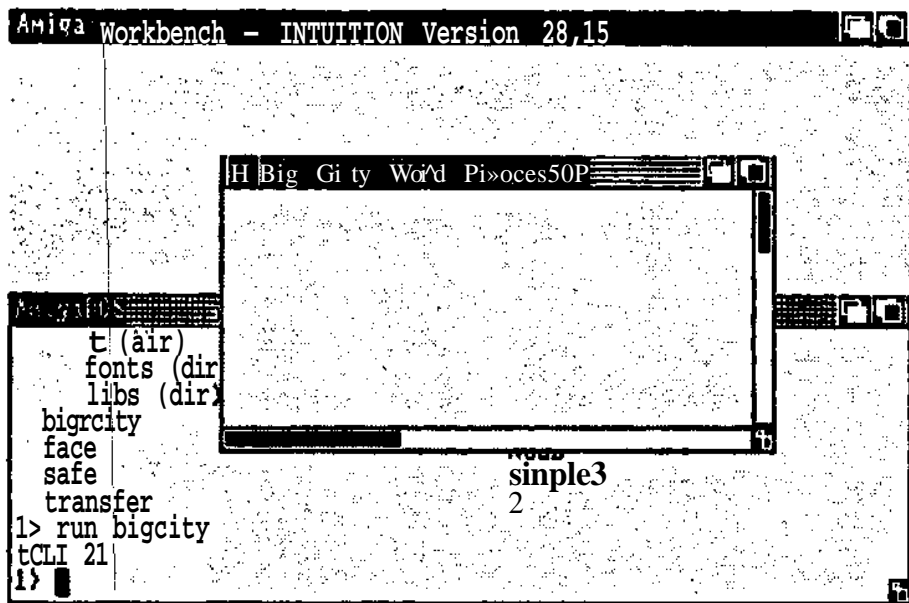


Figure 1-1: A Screen with Windows

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 requires no 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 subitem list (see figure 1-2).

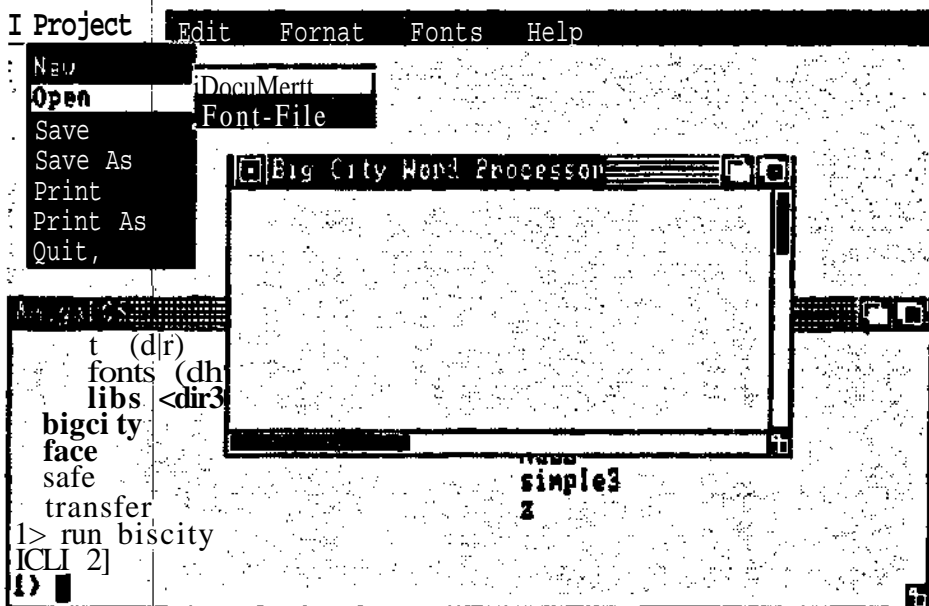


Figure 1-2: Menu Items and Subitems

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 (see figure 1-3). 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.

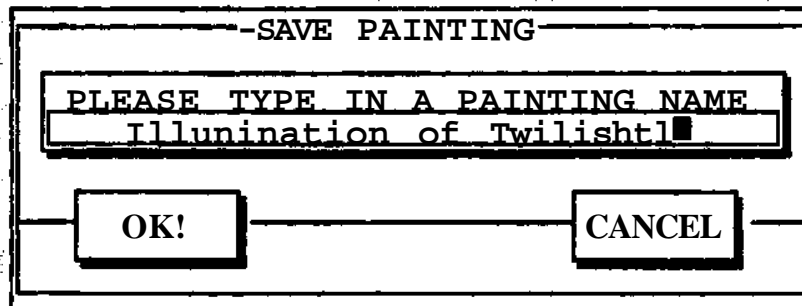
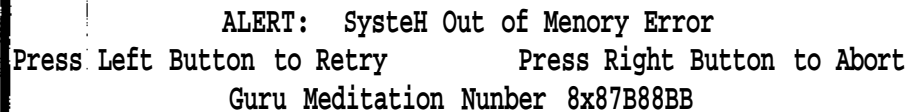


Figure 1-3: A Requester

The *alert* (see figure 1-4) is another kind of special information exchange device invoked by the system or an application. 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 to be used when a serious problem has occurred 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 8x87B88BB

Figure 1-4: An Alert

4 The Right Approach to Using Intuition

Intuition is a Very flexible program environment, with a vast number of features and defaults. The tools and devices are well defined and easily accessible. Although many default values are provided for you to rely on, few restrictions are placed on you. You are encouraged to let your creativity flow. Taking advantage of the many Intuition features enables you to spend less time implementing user-interaction mechanisms of your own, since Intuition already provides a wide range of them for you; in addition, the user of your code gets to work in an environment that does not 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. Alternatively, 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. Intuition allows you to create your own pointer and to combine elaborate graphic images and text strings in menu items. You can also choose to mix predefined features and custom designs. Your creative freedom is practically limitless under Intuition.

No matter how simple, complex, 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 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 to the sophisticate. This is the essence and the beauty of the Intuition philosophy.

Chapter 2

GETTING STARTED WITH INTUITION

Intuition Components

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

- o Windows provide the means for obtaining input from the user; they are also the normal destination for the program's output.

- o Screens provide the background for opening windows.
- o Numerous mechanisms exist for interaction between users and applications:
 - o Menus present users with options and give them an easy way of entering commands.
 - o Requesters provide a menu-like exchange of information.
 - o Gadgets are the main method of communication.
 - o Alerts are for emergency communications.
 - o The mouse is the user's primary tool for making selections and entering commands.
 - o The keyboard is used for entering text and as an alternate shortcut method of entering commands.
 - o Other input devices, like graphics tablets or music keyboards, provide additional means of user input.
- o The methods of program input and output are as follows:
 - o Input is received through the console device or Intuition Direct Communication Message Ports (known as the IDCMP).
 - o Output is transmitted through the console device or directly to the graphics, text, and animation library functions.

General Program Requirements and Information

As an introduction to the basic requirements for an Intuition application, instructions are given here for creating a simple program, which involves the following elements:

- o The necessary *header* files must be included. 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 `OpenLibraryQ` 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 **OpenWindowQ** with a pointer to that structure.
- o You open a screen by initializing the data of a **NewScreen** structure and then calling **OpenScreenQ** with a pointer to that structure.
- o Finally, the example program writes some simple text to a window, illustrating 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 plain window:

```

/****^*****^*****
*
* Simple OpenWindowQ program
*
*****/

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

#define INTUITION_REV 0
#define MILLION 1000000

mainQ
{
    struct NewWindow NewWindow;
    struct Window * Window;
    LONGi;

    /* Open the Intuition library. The result returned by this call is
     * bsd 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", INTUITIONJREV);
    if (IntuitionBase === NULL) exit(FALSE);

```

```

-/* Initialize the NewWindow structure for the call to OpenWindowQ */
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 = SMARTJREFRESH | ACTIVATE;
NewWindow.IDCMPFlags = NULL;
NewWindow.Type = WBENCHSCREEN;
NewWindow.FirstGadget = NULL;
NewWindow.CheckMark = NULL;
NewWindow.Screen = NULL;
NewWindow.BitMap = NULL;
NewWindow.Min Width = 0;
NewWindow.MinHeight = 0;
NewWindow.MaxWidth = 0;
NewWindow.MaxHeight = 0;

/* Try to open the window. Like the call to OpenLibraryQ, if
 * the OpenWindowQ call is successful, it returns a pointer to
 * the structure for your new window.
 * If the OpenWindowQ 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 example is a complete program that can be compiled as is. If run, this program would open the window shown in figure 2-1.

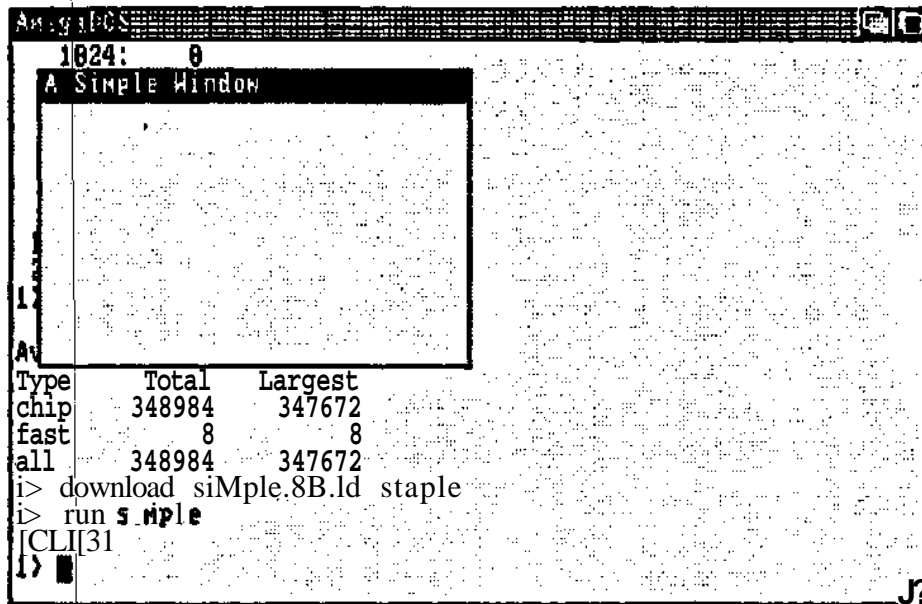


Figure 2-1: A Simple Window

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 SMAIttVREFRESH 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, you cannot do much with this window.

SEMPIE PROGRAM: ADDING THE CLOSE GADGET

The next program allows you to close the window when you like, rather than having it close automatically.

First, the program asks Intuition to attach a WINDOWCLOSE gadget. Then Intuition is instructed to tell the program when someone activates that WINDOWCLOSE gadget.

To ask for the WINDOWCLOSE gadget, change the Flags variable to include the WINDOWCLOSE flag:

```
NewWindow.Flags = WINDOWCLOSE | SMARTJIEFRESH | ACTIVATE;
```

Asking Intuition to tell the program that the gadget has been activated requires several steps:

- o Intuition must be told to inform the program about the event;
- o Intuition must wait for the event to happen;
- o When the event occurs, Intuition must know to close the window and exit.

You instruct Intuition to tell the program about the event by specifying one of the event flags in the IDCMPFlags variable. As mentioned earlier, the IDCMP is Intuition's Direct Communications Message Port system. By setting one of the IDCMP flags, you are requesting Intuition to open a pair of message ports through which the program may communicate.

```
/* Tell the program about CLOSEWINDOW events */  
NewWindow.IDCMPFlags = CLOSEWINDOW;
```

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

```
Wait(1 << Window->UserPort->mpJSigBit);  
Close Window(Window);  
exit(TRUE);
```

The variables UserPort and mpJSigBit are initialized for you by Intuition, so you can ignore these for now. WaitQ is a function of the Amiga Executive. It allows your program to do absolutely nothing, thereby freeing the processor for other jobs, until some

special event occurs. In this simple example, only one type of event will wake up the program; thus when the program is awakened, it can automatically assume that it was because someone pressed the WINDOWCLOSE gadget. When you start using the IDCMP for more elaborate functions, your programs will have to get a message from the message port and examine it to see what event has occurred to awaken the program.

SIMPLE PROGRAM: ADDING THE REMAINING SYSTEM GADGETS

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

```
NewWindow.Flags = WINDOWCLOSE | SMARTJREFRESH  
                | ACTIVATE | WINDOWDRAG | WINDOWDEPTH  
                | WINDOWIZING | NOCAREREFRESH;
```

The WINDOWDRAG flag means that you want to allow the user to drag this window around the screen. If you do not set this flag, the window cannot be moved.

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

The WINDOWIZING 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 SMARTJREFRESH windows to be refreshed (to refresh a window means to redisplay the information contained in that window). If you *really* do not care about whether you should refresh your window (as is the case here) then you can set the NOCAREREFRESH flag and Intuition will take care of all the refresh details for you. Because this example allows the window to be sized now, it must tell Intuition about the minimum and maximum sizes for the window:

```
New Window.Min Width = 100;  
NewWindow-MinHeight = 25;  
NewWindow.MaxWidth == 640;  
NewyVindow.MaxHeight = 200;
```

If you run the program with these changes, your window will look like figure 2-2.

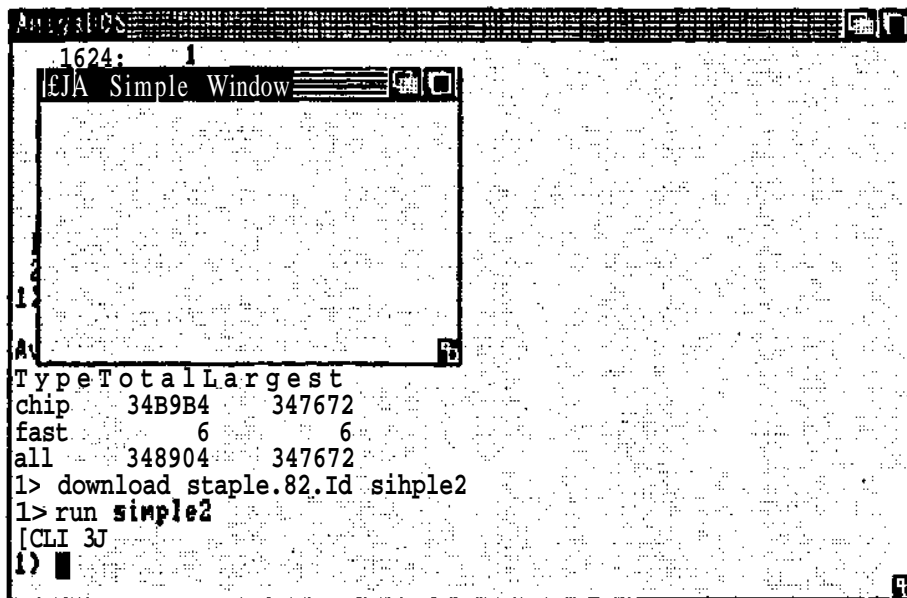


Figure 2-2: A Simple Window with Gadgets

SIMPLE PROGRAM: OPENING A CUSTOM SCREEN

To open a custom screen, you must initialize a **NewScreen** block of data and then call **OpenScreenQ** with a pointer to that data.

To open the window in the earlier example, you initialized a **NewWindow** structure by writing a long series of assignment statements. A more compact method for initializing a structure is used here to open a custom screen.

```

/* This font declaration will be used for the new screen */
struct TextAttr MyFont =
{
    "topaz.font",          /* Font Name */
    TOPAZJSIXTY,          /* 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 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 (4 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 the screen is opened:

```

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

```

Because this window will open in a custom screen, you must change the initialization of the **NewWindow** data slightly. The **Type** declaration should be changed from **WBENCHSCREEN** to **CUSTOMSCREEN**. Also, you previously set the **Screen** variable to **NULL** because you wanted the window to open in the Workbench screen. Now, you must set this variable to point to the new custom screen:

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

After you close the window, close the screen, too:

```
CloseScreen(Screen);
```

The program now opens a custom screen and then opens a window in that screen.

SIMPLE PROGRAM: THE FINAL VERSION

For a finishing touch, try writing a bit of text to the new window. This will require another declaration, another call to **OpenLibrary()**, and a call to the graphics library's **MoveQ** and **TextQ** functions:

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

All together, the finished program looks like the following:

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

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

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

#define INTUITIONJtEV 0
#define GRAPHICS_JtEV 0

struct TextAttr MyFont =
```



```

"topaz.font",      /* Font Name */
TOPAZ_SIXTY,      /* Font Height */
FSJMORMAL,        /* Style */
FPFJROMFONT,      /* Preferences */
};

```

/* This is the declaration of a pre-initialized NewScreen data block.

* It often requires less work and 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 (4 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 */
};

```

```

mainQ

```

```

{

```

```

    struct Screen *Screen;
    struct NewWindow NewWindow;
    struct Window *Window;
    LONGi;

```

/* 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 *)

```

```

    OpenLibraryf("intuition.library", INTUITIONJtEV);

```

```

    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 | SMARTJtEFRESH 1 ACTIVATE
    | WINDOWresizing | WINDOWDRAG | WINDOWDEPTH | NOCAREREFRESH;
NewWindow.IDCMPFlags = CLOSEWINDOW;
NewWindow.Type = CUSTOMSCREEN;
NewWindow.FirstGadget = NULL;
NewWindow.CheckMark = NULL;
NewWindow.Screen = Screen;
NewWindow.BitMap = NULL;
NewWindow.Min Width = 100;
NewWindow.MinHeight = 25;
NewWindow.Max Width = 640;
NewWindow.MaxHeight = 200;

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

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

Waitfl < Window->UserPort->mp_SigBit);
Close Window(Window);
CloseScreen(Screen);
exit(TRUE);
}

```

The display created by the final version looks like figure 2-3.

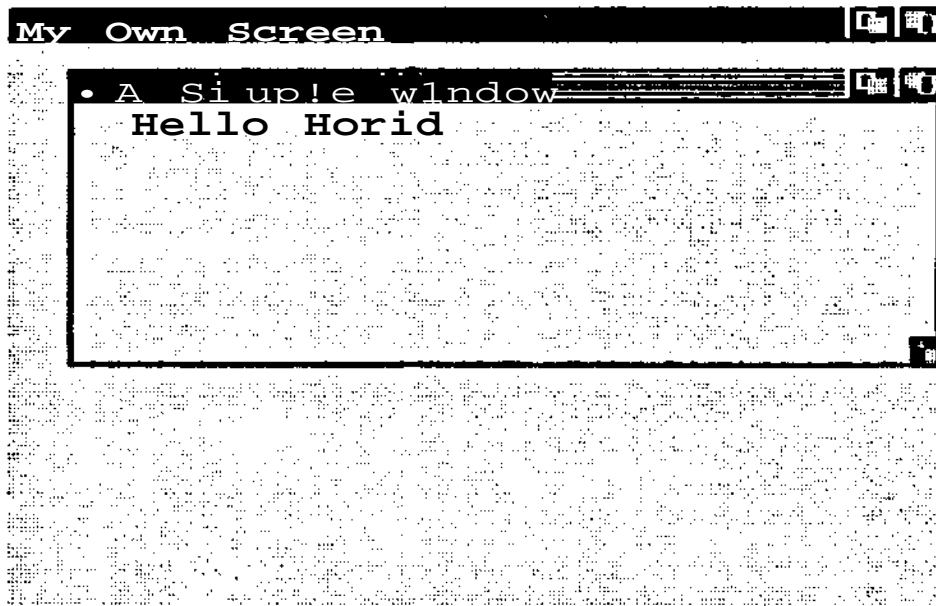


Figure 2-3: Display Created by Intuition's "Hello World" Program

Chapter 3

SCREENS

Screens 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.

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 eight different colors in low-resolution mode or 32 colors in interlaced 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 one another 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, allowing 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.

Screens provide display memory, which is the RAM in which 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. 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, screens can be moved up or down and arranged at different depths by using special control mechanisms called gadgets. Unlike windows, however, screens cannot be made larger or smaller, and they cannot be moved left or right.

The dragging and depth-arrangement gadgets reside in the title bar at the top of all Intuition screens. In the title bar there may also be a line of text identifying the screen and its windows.

Figure 3-1 shows a screen with open windows. The depth-arrangement gadgets (front gadget and back gadget) are at the extreme right of the screen title bar. The drag gadget (for moving the screen) 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.

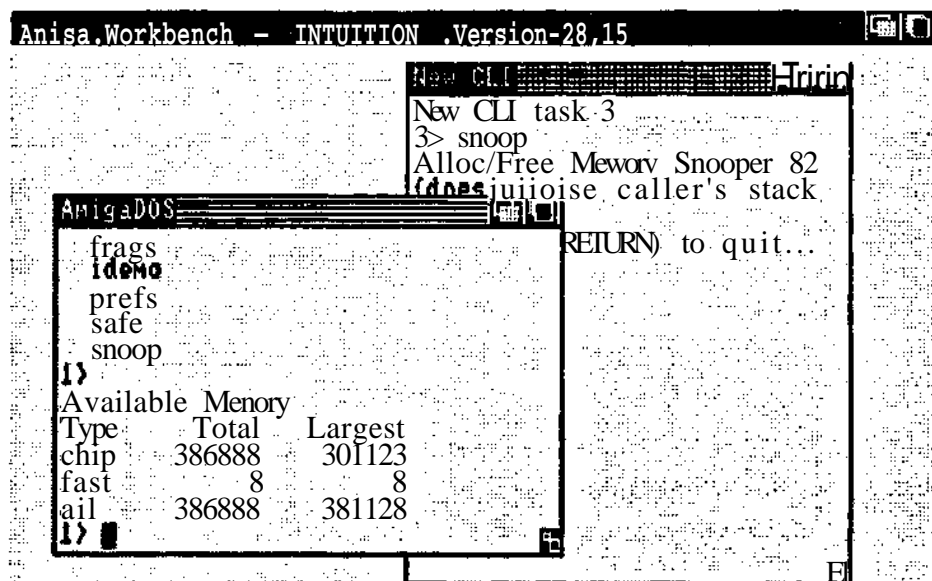


Figure 3-1: A Screen and Windows

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

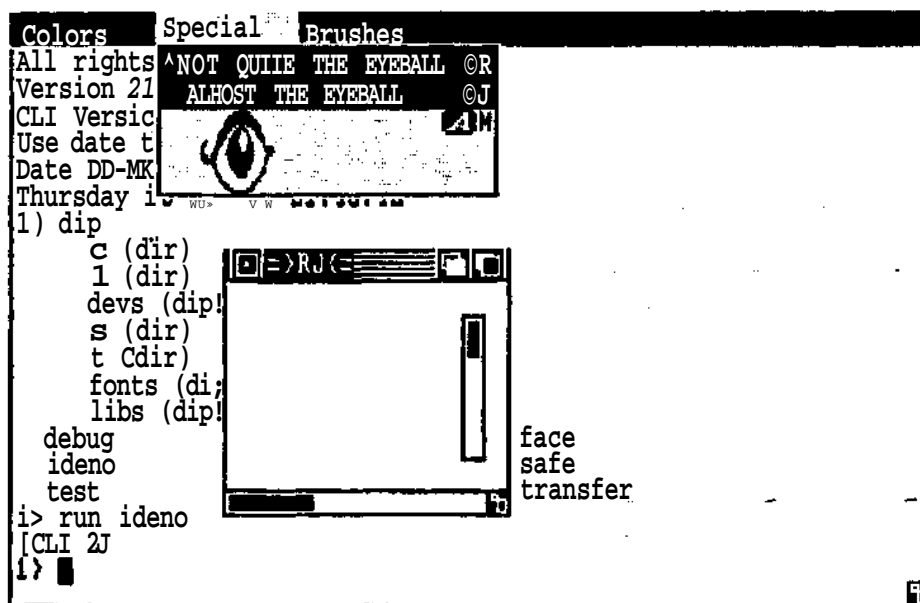


Figure 3-2: Screen and Windows with Menu List Displayed

By further mouse movement and mouse button manipulation, the user can see a list of menu items and subitems for each of the topics in the menu list. The menu list, menu items, and subitems that are displayed pertain to the currently active window, which is the window receiving the user's input. Only one window is active 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 putting menus together and attaching 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

Standard screens differ from custom screens in three basic ways.

- o Standard screens close and disappear if all their windows are closed. Only the Workbench standard screen (described below) differs in this regard. You can think of the Workbench as the default screen—even if all its windows close, it remains open. If the Workbench isn't already open when all other screens close, it will open then.
- o Standard screens are opened differently. No function need be called to explicitly open a standard screen. You simply specify the screen type in the window structure; if the screen is not already open, Intuition opens it. This is contrasted with custom screens, which you must explicitly open 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 predefined 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, that program's graphics display will not be able to share the screen with you, which defeats the purpose of standard screens.

All of Intuition's 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 the Intuition standard screen. It is both a screen and an application. It is a high-resolution (640 pixels x 200 lines) four-color screen. The default colors are blue for the background, white and black for details, and orange for the cursor and pointer (see figure 3-3).

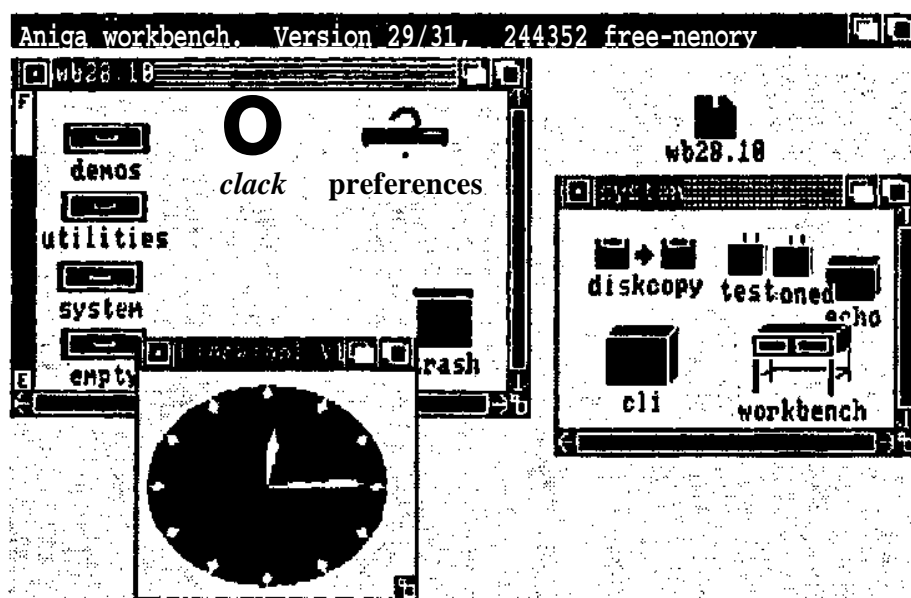


Figure 3-3: The Workbench Screen and the Workbench Application

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 very memory-efficient, because you will not be allocating the memory for your own custom screen.

Besides your own custom screen, the Workbench screen is the only one 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 is available, it can attempt to reclaim the memory used by the Workbench screen by calling **CloseWorkBenchQ**. 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 **CloseWorkBenchQ**. If all programs do this, it will help to present the user with as consistent and dependable an 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 **CloseWorkBenchQ**, even though the Workbench tool may have open windows in the screen.

You have access to a body of library functions that allows you to create and manipulate the Workbench application's objects and iconography. The functions in the library allow you to create disk files that 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 is 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. Alternatively, 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 do not close automatically when all windows in them close (as system standard screens do, except for the Workbench). If you have opened a custom screen, you must call **CloseScreenQ** before your program exits. Otherwise, your screen stays around forever.

You can create two kinds of custom screens: one that is entirely managed by Intuition or one that uses the Amiga graphics primitives to write directly into the display memory or otherwise directly modify the screen display characteristics. If the second kind of screen is used, you must take on some of the responsibility of managing the display.

INTUITION-MANAGED CUSTOM SCREENS

If you want Intuition to manage your 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 for doing block fills, such as the title bar area.
- o Display modes—high or low resolution, interlaced or non-interlaced, sprites, and dual playfields.
- o Initial display memory.

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

APPLICATION-MANAGED CUSTOM SCREENS

In an application-managed custom screen, the same structures and functions are used, but another dimension is added when you access the display memory directly. 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 more. Although you can still combine such a screen with other Intuition features — for example, windows, menus, and requesters—certain interactions can trash the display. The interactions described in the next paragraph are those that take place 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. Therefore, be 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, 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 carefully so that they interact without conflict. 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), you should use a custom screen. If you want to control the display memory *and* run windows and menus in the custom screen, 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.

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? A special kind of window 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 management 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) 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: interlaced and non-interlaced. 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 does not have the potential for flickering, as interlaced displays tend to do. Intuition supports interlaced mode 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 eight 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 sixteen bits wide and any number of lines tall, can have three colors (plus transparent), and pairs of sprites can be joined to create a fifteen-color (plus transparent) sprite. They are also reusable vertically, so you can really have more than eight at one time. The Amiga GELS system described in the *Amiga ROM Kernel Manual* provides just such a *multiplexing*, 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 that allows you to 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 that 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.

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

If you want to use sprites, dual playfields, or hold-and-modify mode, 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 are 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 three-bit-plane display is defined by three bits. The binary number formed by these three 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 eight registers numbered 0 through 7. The thirty-two 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 **SetRGB4()**. 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 one to five bit-planes. For dual playfields, you can have from two to six bit-planes, which are divided between the two playfields. For hold-and-modify mode you need six 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. **BlockPen** is used for block fills, such as 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 Eight display lines tall with 80 characters per line in a 640-pixel high-resolution display (40 characters in low resolution).
- o Nine 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 (see figure 3-4). You can use or ignore the user's choice of default font size. See chapter 11, "Other Features," for more information about Preferences items.

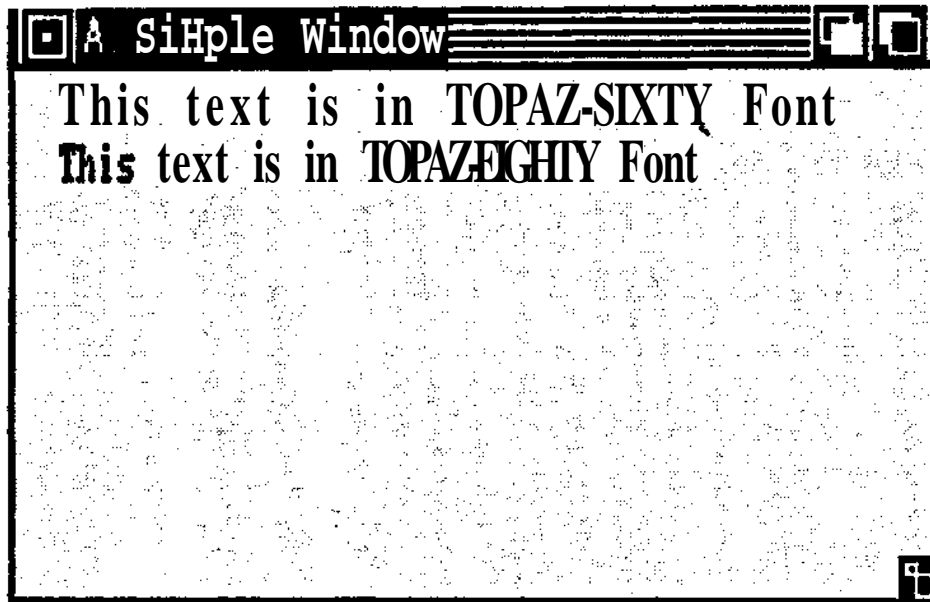


Figure 3-4: Topaz Font in 60-column and 80-column Types

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 **TopEdge** and **LeftEdge** fields of the screen structure. After that, the user can drag the screen up or down. You must always set the **LeftEdge** 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-interlaced mode the maximum is 200 lines; for interlaced mode, 400 lines. You set the width to 320 for low-resolution mode or 640 for high-resolution mode.

In setting the TopEdge 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, a display can look like figure 3-5, but not like figure 3-6.

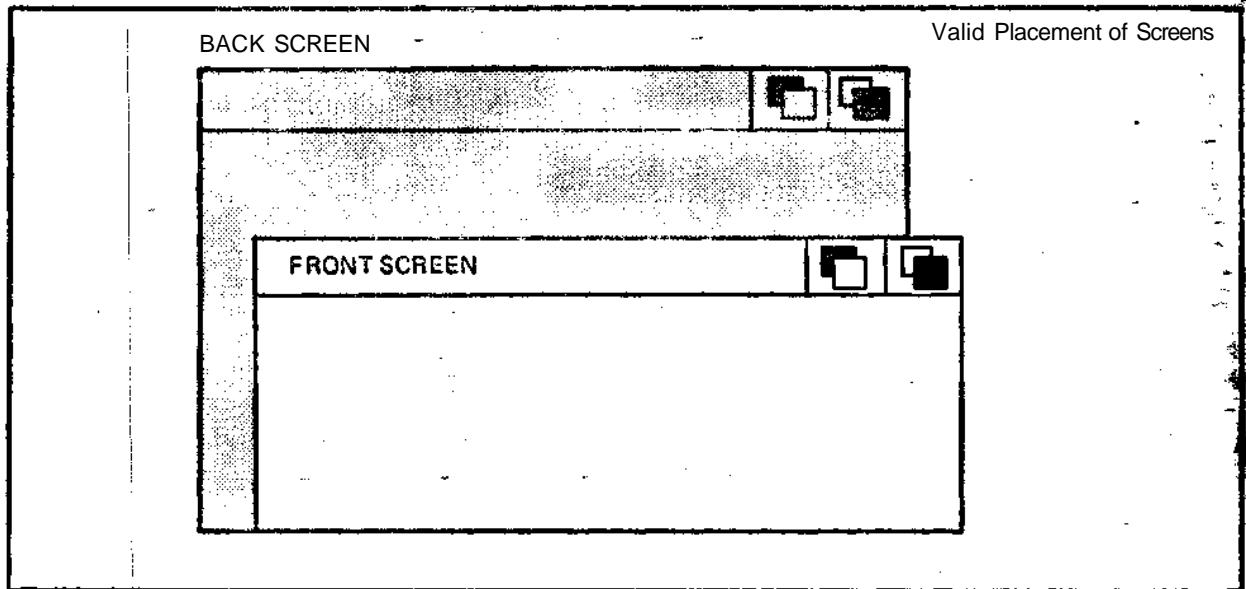


Figure 3-5: Acceptable Placement of Screens

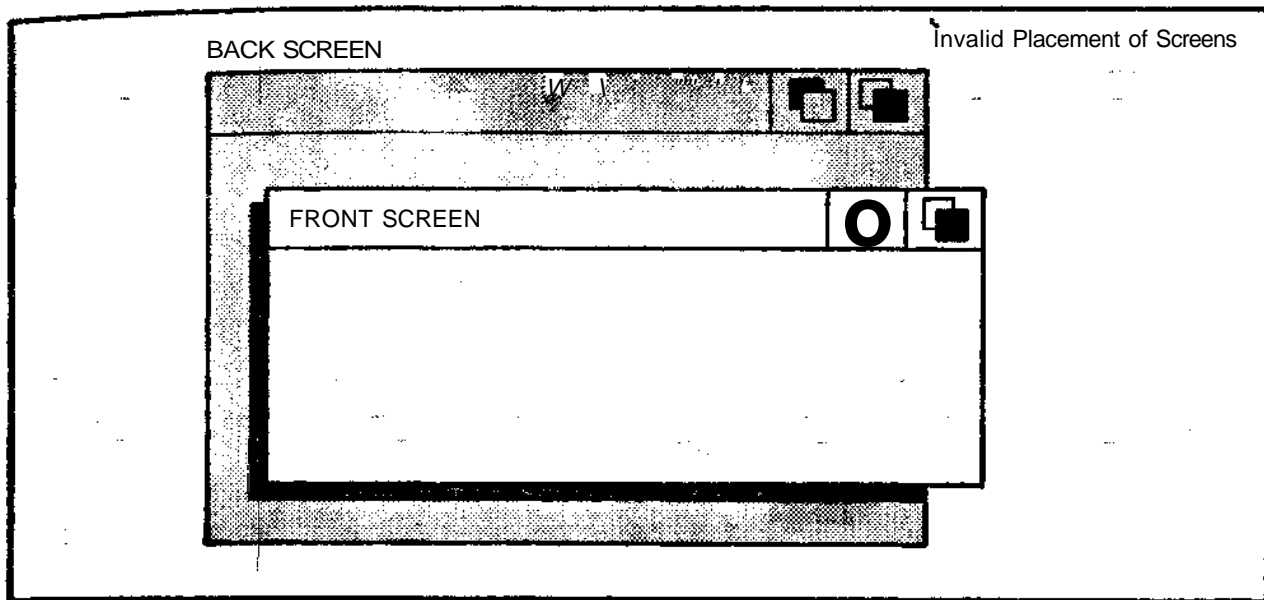


Figure 3-6: Unacceptable Placement of Screens

SCREEN TITLE

The screen title is used for two purposes: to identify the screen like an identification tab on a file folder and 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 `ShowTitleQ`, 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 `SetWindowTitlesQ`.

CUSTOM GADGETS

You cannot attach custom gadgets directly to a screen. You can attach custom gadgets to a borderless backdrop window and monitor their activity through the window's input/output channels. See chapter 5, "Gadgets," for information about using custom gadgets.

Using Custom Screens

To create a custom screen, 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 **`OpenScreenQ`** returns a pointer to your new screen (or returns `NULL` if your screen cannot be opened).
3. After you call **`OpenScreenQ`**, the **`NewScreen`** structure is no longer needed. If you have 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 in non-interlaced or 400 interlaced 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 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 the **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-interlaced mode and 400 for interlaced 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 interlaced mode (400 lines). The default is 200 lines.

SPRITES

Set this flag if you 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 mode.

Typ<5

Set this to CUSTOMSCREEN. You may also set the CUSTOMBITMAP flag if you want to use your own bit-map and display memory 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; this should be set to NULL if you want a blank title bar. Null-terminated means that the last character in the text string is NULL.

Gadgets This field is not used at this time. It should be set to NULL.

CustomBitMap

A pointer to a **BitMap** structure, used if you want your own display memory to be 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 it.

SCREEN STRUCTURE

If you have successfully opened a screen by calling the **OpenScreenQ** function, you receive a pointer to a **Screen** structure. The following list shows the variables of the **Screen** structure that may be of interest to you. This is not a complete list of the **Screen** variables; only the more useful ones are described. 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.

ViewPprt, RastPort, BitMap, LayerInfo

For 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.

BarLayer

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

SCREEN FUNCTIONS

Here is a quick rundown of Intuition screen functions. For a complete description of these functions, see appendix A.

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 sub-structures, does all the memory allocations, and links the screen's **ViewPort** into Intuition.

OpenScreen (NewScreen)

The argument is a pointer to an instance of a NewScreen structure.

Showing a Screen Title Bar

This function causes the screen's title bar to be displayed or concealed, according to your specification of the **Showit** variable and the position of the various types of windows that may be opened in the screen.

Show Title (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 not enough memory exists to open the screen, this routine fails. Also, if the Workbench tool is active, it will attempt to reopen its windows.

CloseWorkBenchQ

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, the Workbench tool will close its windows and allow the screen to close.

WBenchToFrontQ, WBenchToBackQ

If the Workbench screen is opened, calling these routines will cause it to be in front or in back of other screens, depending on which command is used. If the Workbench screen is closed, 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 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 MakeVPortQ graphics library function. MakeScreenQ 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 that contains the ViewPort that you want remade.

RethinkDisplayQ

This procedure performs the Intuition global display reconstruction, which includes massaging some of Intuition's internal state data, rethinking 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 MrgCopQ and LoadViewQ.

RemakeDisplayQ

This routine remakes the entire Intuition display. It performs a MakeVPortQ (graphics primitive) on every Intuition screen and then calls RethinkDisplayQ 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—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 Communications 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 and expand them, or overlap them.

Windows are rectangular display areas whose size and location can be adjusted in many ways. The user can shape windows by making them wider or longer or both to reveal more of the information being output by the program. He can also shrink windows into long, narrow strips or small boxes to reveal other windows or to 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 need not 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, a graphics tablet, or practically any other input device).

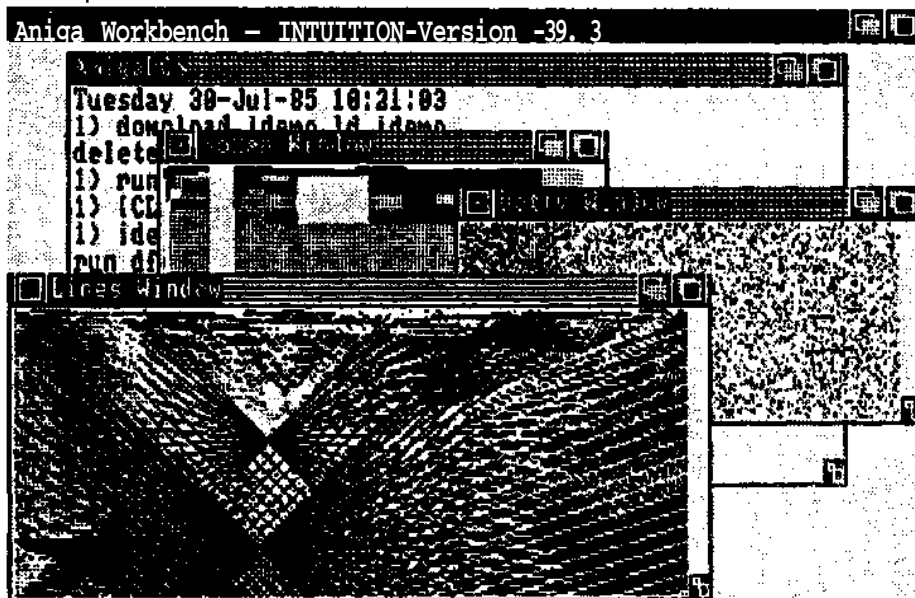


Figure 4-1: A High-resolution Screen and Windows

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 later chapters. The balance of this section deals with some important concepts you'll need to know before attempting to open your own windows.

WINDOW INPUT/OUTPUT

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

- o [ntuition Direct Communications Message Ports (IDCMPs). The message ports give you mouse (or other controller) events, keyboard events, and Intuition messages in their most raw form; in addition, these ports supply the way for your program to send messages *to* Intuition.
- o Console device. The console ports give you processed input data, including key-codes 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 automatic line wrapping and scrolling.
- o Cgraphics are output through the general-purpose Amiga graphics primitives, which provide rendering functions such as area fill and line-drawing and animation functions.

If you use the console device for input, 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 does not offer text formatting or character positioning, it has many special features that you may want, and it requires less RAM and less processing overhead.

For more information about I/O methods read chapter 8, "Input and Output Methods."

OPENING, ACTIVATING AND CLOSING WINDOWS

Before your jprogram 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 is one of the special windows, and the screen in which the window should open.

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 is receiving user input through a keyboard and mouse (or some other controller). Some areas of the [active window are displayed more boldly than those on inactive windows. 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 need not worry about whether or not one of its windows is active. The inactive windows can just wait for the user to get back to them, or they can be doing some background task that requires no 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 left 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. Although windows opened after the first one may have the `ACTIVATE` flag set, they do not need to. It is up to you to design the flow of information and control.

After your window is opened, you can discover when it is activated and when it is inactivated by setting the IDCMP flags `ACTIVE WINDOW` and `INACTIVE WINDOW`. If you set these flags, the program 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 the program a message about the user's action. The program 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 **CloseWindowQ**.

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

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 provides a way to take over the display without taking over the machine. The Borderless window supplies 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 are drawing into the window. Finally, the SuperBitMap window not only gives you your own display memory in which to draw, but also 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 have 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 is 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 does not cover the entire display, be aware that its lack of borders may 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 in which the window title, gadgets, and border are placed; and a smaller, inner plane (also called the *inner window*) in 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 in which you can draw is formally defined as the area within the variables **BorderLeft**, **BorderTop**, **BorderRight**, and **BorderBottom**. These variables are computed by Intuition when the window is opened. To draw in 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 about the window borders.

The Gimmezerozero window uses more RAM than other window types and degrades performance in the 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 that 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. 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 output goes into an inner window automatically; you need not worry about gadgets. Therefore, you do not need a Gimmezerozero window just for the purpose of text output. See 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 them. See chapter 7, "Requesters and Alerts," for more information about requester location.

To specify a Gimmezerozero window, set the GIMMEZEROZERO flag in the window structure's flags. All system gadgets you attach to this type of window will go into the outer window automatically; 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 do not, Intuition will draw 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 that:

- o It always opens behind all other windows (including other Backdrop windows that you might already have 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 in which 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 drawing directly into the display memory of a custom screen. If you want to draw in 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 is 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. 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 use 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 that you get your own bit-map instead of using the one belonging to the screen. 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 handy when 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, set the SUPERJBITMAP 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 predefined and managed by Intuition and your own custom application gadgets.

System Gadgets

System gadgets are supplied to allow the user to manage the following 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 drawing you may be doing in the window, Intuition will let you know. For more information, read about the IDCMP verify functions in 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 it about on the screen and 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 predefined, and Intuition always displays them in the same standard 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.

Figure 4-2 shows how all the system window gadgets look and where they are located in the window borders.

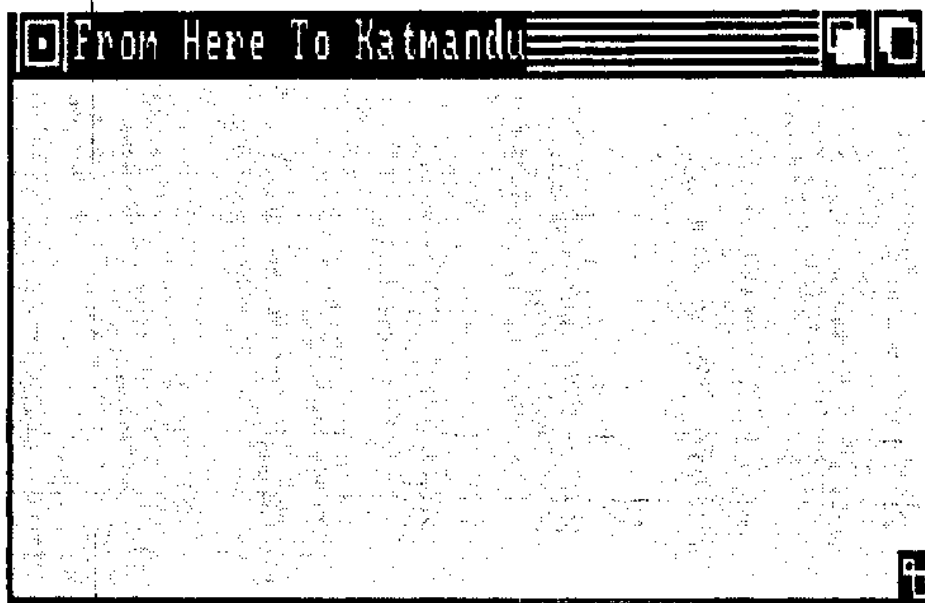


Figure 4-2: System Gadgets for Windows

Application Gadgets

Four types of (application gadgets are available—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 gadgets as text and 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, set a pointer in the **NewWindow** structure to point to the first gadget in the list. For details about creating gadgets, see chapter 5, "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, which 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* in which 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 **New Window** 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, 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**, **BorderRight**, 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 in the window with graphics primitives, which require you to specify a set of coordinates as the beginning and ending points for the line. In a typical window, you cannot 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 — although much more convenient to use—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 unless you specify one of the following:

- o A window title.
- 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, however, 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 draw freely across 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, such as rtenu 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 draw in virtual display memory and translates them into real operations that are placed in save buffers (for Smart Refresh) or in areas of a private bit-map (for SuperBitMap) maintained by the application.

The three methods of preservation are explained below. You must choose one of them. Figures 4-3, 4-4, and 4-5 compare the three methods.

Simple Refresh

With the Simple Refresh redrawing method, Intuition does not 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 redisplay 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.

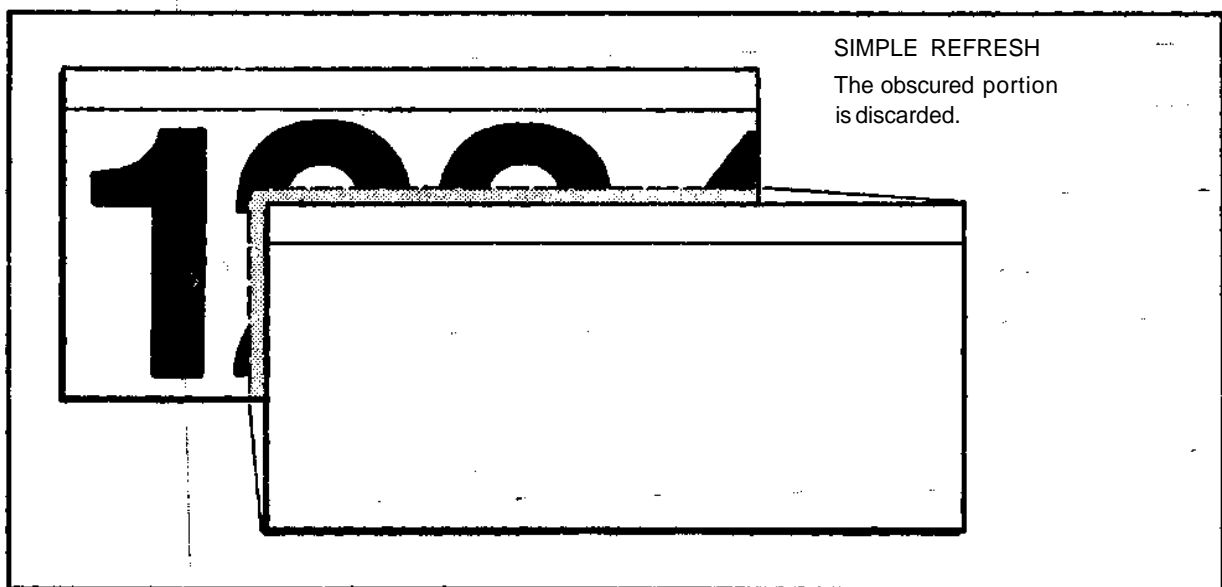


Figure 4-3: Simple Refresh

Smart Refresh

With the Smart Refresh 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. 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 redraws the display faster than Simple Refresh.

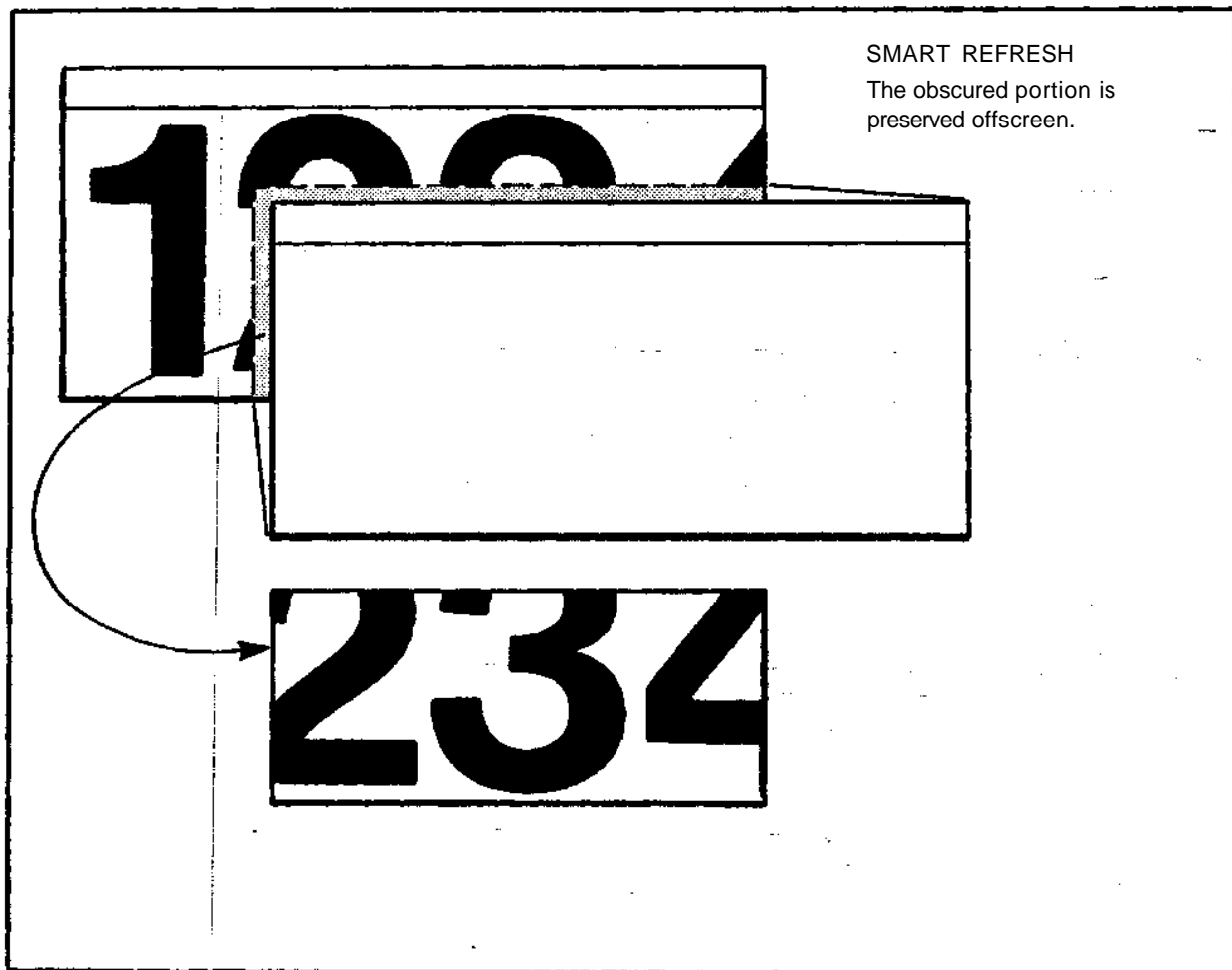


Figure 4-4: Smart Refresh

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.

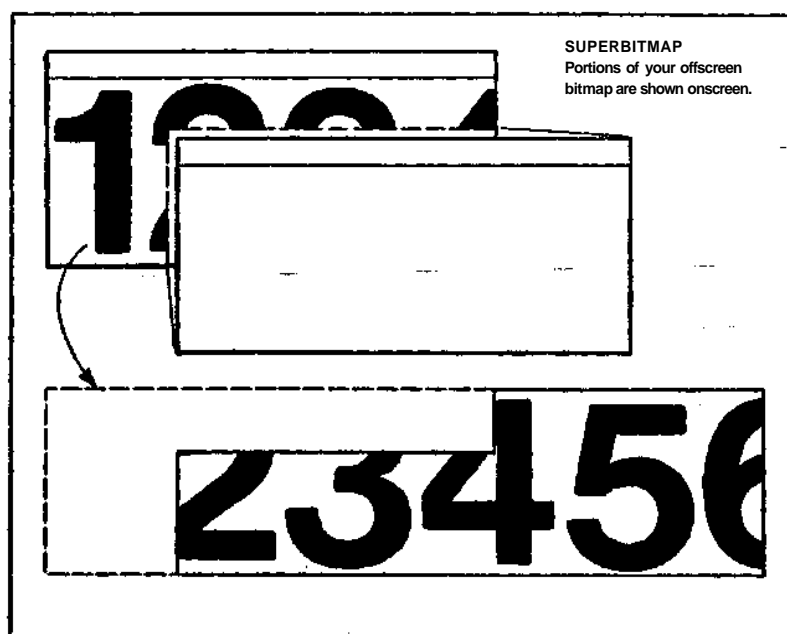


Figure 4-5: SuperBitMap Refresh

REFRESHING THE WINDOW DISPLAY

If you open either a Simple Refresh or a Smart Refresh window, your program may be asked to refresh part of your display at some time. 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 must be sized larger on either axis to generate a REFRESHWINDOW event.

The program finds out that the window needs refreshing via either source of input, the IDCMP or the console device. A message of the class REFRESHWINDOW arrives at the IDCMP, telling the program that the window needs to be refreshed. Every time the program learns that it should refresh a window, it must take some action, even if it is just the acceptable minimum action described below.

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

After `BeginRefresh()` is called, the program should redraw its display. Then, call `EndRefresh()` to restore the state of the internal structures.

Even if you don't want the program to redraw immediately, you should make sure the program at least calls `Begin/EndRefresh()` each time it is asked to refresh a window. This helps Intuition and the layer library keep things sorted and organized.

If you are opening a window that you will never care to refresh, no matter what happens to or around it, then you can avoid calling `BeginRefresh()` and `EndRefresh()` by setting the `NOCAREREFRESH` flag in the `NewWindow` structure when you open your window.

WINDOW POINTER

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

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 **ReportMouseQ**.

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, the current pointer position is reported 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, call **SetPointer()**. To remove your custom pointer from the window, 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 in which 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 (such as 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 **New Window** 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 do not ask that the window sizing gadget be attached to the window, then you need not initialize any of these maximum and minimum variables.

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

If you want to change the sizing limits after you have opened the window, you can call `WindowLimitsQ` with the new values.

Using Windows

To create a window, follow these steps:

1. Initialize a **New Window** structure.
2. When you are ready to display the window, call **OpenWindow()** with a pointer to the **New Window** structure.
3. After you call **OpenWindowQ**, 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:
 - o Which system gadgets you want.
 - o Preservation method for the window display.
 - o Special window features—Gimmezerozero, Borderless, Backdrop, or SuperBitMap.
 - o Type of input from the Intuition Direct Communications Message Ports (if any).
 - o Pointer movement broadcasts.
 - o Other characteristics, such as starting position and size and color of the pens used to draw borders and fill blocks.
 - o Custom images, such as a custom "check mark" for the menus or a custom pointer.

NEWWINDOW STRUCTURE

Here are the specifications for the **New Window** structure:

```
struct NewWindow
{
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    UBYTE DetailPen, BlockPen;
    ULONG 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;
};
```

The fields in the **NewWindow** structure are explained below. 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 fields 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

These fields 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

You can set any of the following flags.

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

WINDOWSIZING

This flag allows the user to change the size of the window. Intuition places the window's 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 The **SIZEBRIGHT** flag, which is the default, puts the sizing gadget in the right border.
- o The **SIZEBBOTTOM** flag puts the sizing gadget in the bottom border. You might wish to set this flag 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

This flag 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 is 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 flag 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 moving the mouse or other controller.

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. If no text is provided, 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:

SIMPLEJIEFRESH

When this flag is set, every time a portion of the window is revealed the application program must redraw its display.

SMARTJIEFRESH

When this flag is set, the only time you have to redraw your display is when the window's sizing gadget is used to make the window larger.

NOTE: If you open a SMARTJIEFRESH window without asking for the sizing gadget, 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

This flag sets the window to receive pointer movements as x,y coordinates. Also see the description of the IDCMP flag, MOUSEMOVE, in chapter 8s "Input and Output Methods."

BORDERLESS

This flag creates a window with none of the default border padding and border lines.

NOTE: Be careful when you set this flag. It may cause visual confusion on the screen. Also, there may still be some borders if you have 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 flag is set, the window automatically becomes active when it is opened.

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

NOCAREREFRESH

Set this flag if you do not want to receive messages telling you to refresh your window.

RMBTRAP

Set this flag if you do not want any menu operations at all for your window. Whenever the user presses the right mouse button while this window is active, the program will receive normal MOUSEBUTTON events.

IDCMPFlags

The **IDCMPFlags** are listed and described in appendix A 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

This is a pointer to the first in the linked list of custom **Gadget** structures that you want included in the window.

CheckMark

This is 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 (vO> set to this field to NULL.

Text

This is 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

This contains the screen type for this window. The currently available types are **WBENCHSCREEN** and **CUSTOMSCREEN**.

NOTE: If you choose **CUSTOMSCREEN**, you must have already 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 flag 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 **WindowLimitsQ**.

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 have successfully opened a window by calling the **OpenWindow()** function, you receive a pointer to a **Window** structure. This section describes some of the more useful variables of the **Window** structure. A complete description of the Window structure is given 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 and the offset into normal windows after taking the borders into account.

ReqCount

You can examine this 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 have opened this window **in** a custom screen of your own making, you should already know the address of the screen. However, if you have opened this window in one of the standard screens, this variable will point you to that screen's data structure.

RPort

This 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, Border Bottom

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, in which the border gadgets are kept.

UserData

This is a memory location that is 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 (New Window)

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)

This function attaches menus to a window, manages the display of windows, and reports to the application when the user makes a menu choice.

ClearMenuStrip(Window)

This function removes the menu strip from a window. After this is done, the user can no longer access menus for this window. If you have called **SetMenuStrip()**, you should call **ClearMenu Strip ()** 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)`

This function determines whether or not mouse movements in this window are reported.

Closing the Window

After the user selects the close gadget, the program can do whatever it needs to do to clean up and then actually close the window with the `CloseWindow (Window)` function. This function closes a window. If its screen is a standard screen (but not the `WorkBench`) that would be empty without the window, this function closes the screen as well.

Requesters in the Window

The following two functions allow requesters to become active:

`Request (Requester, Window)`

This function activates a requester in the window.

`SetDMRequest (Window, Requester)`

This function sets up a requester that the user can bring up in the window by clicking the menu button twice.

These two functions disable requesters:

`EndRequest (Requester, Window)`

7. This function removes a requester from the window.

ClearDMRequest (Window, Requester)

This function 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)

This function sets up the window with a sprite definition for a custom pointer. If the window is active, the change takes place immediately.

ClearPointer (Window)

This function 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)

This function changes the maximum and minimum sizing of the window from the initial dimensions in the NewWindow structure. If you do not 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)

This function changes the window title (and screen title, if this is the

active window) immediately. "**WindowTitle** or **ScreenTitle** can be **-1**, **0**, or a null-terminated string:

- 1** Do not change this title.
- 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)

This function 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 be refreshed will actually be redrawn; the other drawing commands will be discarded.

EndRefresh (Window)

After you've refreshed your window, call **EndRefreshQ** 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 were activating the associated system window gadgets:

MoveWindow (Window, DeltaX, DeltaY)

This function 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)

This function causes your window **to** move in front of all other windows in this screen.

WindowToBack (Window)

This function 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 will not be using the screen's display memory. To set up the bit-map, you need to create a **BitMap** structure and allocate memory space for it.

The general-purpose graphics function **InitBitMap()** prepares a BitMap structure, which 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, bit height)

The arguments you supply are:

bitmap

This is a pointer to the **BitMap** structure to be initialized.

depth

This specifies the number of bit-planes to set up.

bitwidth

This specifies how wide each bit-plane should be, in bits. Should be a multiple of 16.

bitheight

This specifies how high each bit-plane should be, in lines.

The general-purpose graphics function **AllocRasterQ** allocates the memory space for the **BitMap**. 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 SetPointerQ. If your window is active, the new pointer will be attached to the window.

An extra requirement is imposed on sprite data (and Image data). It must be located in chip memory, which is memory that can be accessed by the special Amiga hardware chips. Chip memory is in the lower 512 Kbytes of RAM. In expanded machines (the Amiga can be expanded up to 8,000 Kbytes), the Amiga chips still cannot address memory locations greater than the 512-Kbyte limit. In hexadecimal notation, 512 K spans memory addresses \$00000 to \$7FFFF.

To write a program that will survive in any possible configuration of Amiga hardware, you are obliged to ensure that your sprite and Image data resides in this chip memory. You can make sure that your data is in chip memory by using the ATOM tool on the file containing the data. The loader will then automatically load that portion of your program into chip memory. See the *AmigaDOS User's Manual* for information about **ATOM** and the loader.

As of the time of this writing, the only way to check whether your data is in chip memory is by comparing its load address after it has been loaded into Amiga memory. If the address of the end of your data is less than \$80000, you are safe. If the address is equal to or greater than \$80000, you must allocate chip memory and copy your data into the new location. To allocate chip memory, call the Exec function **AllocMemQ** with MEMF_CHIP as the **requirements** argument.

The Sprite Data Structure

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 0s. All the other words define the appearance of the pointer, two words for each line. For example, the data structure for a sprite shaped like an "X" is shown below.

```

/* The sprite image for the "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 XPOINTERHEIGHT 9
#define XPOINTERJCOFFSET -4
#define XPOINTER_YOFFSET -4

```

```

USHORT XPointerfl =
{
    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.

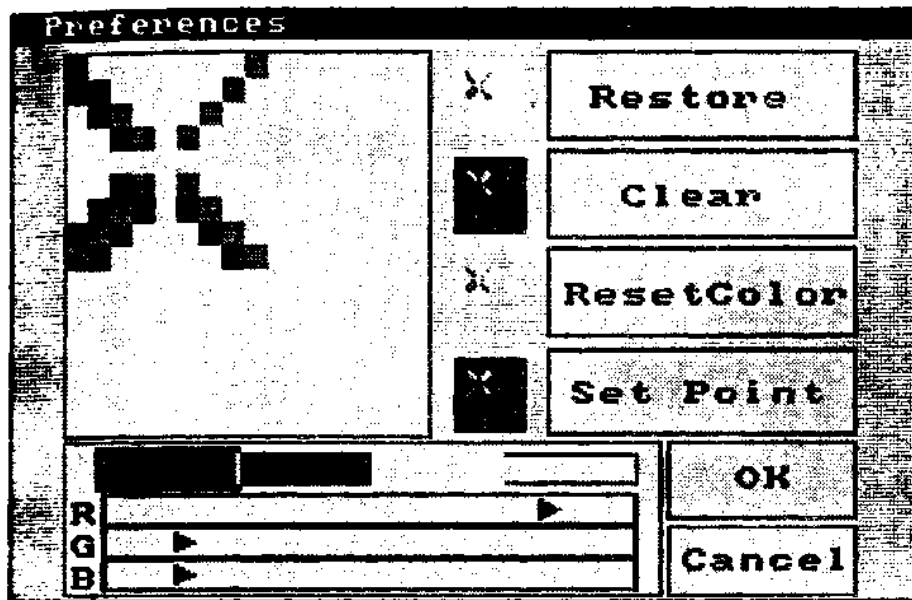


Figure 4-6: The X-Shaped Custom Pointer

Attaching the Pointer to the Window

You call `SetPointerQ` with the following arguments:

Window

This is a pointer to the window that is to receive this pointer definition.

Pointer

This is a pointer to the data definition of a sprite.

Height

This specifies the height of the pointer; it can be as tall as you like.

Width

This specifies the width of the sprite (must be less than or equal to 16).

XOffset, YOffset

These arguments specify the 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, 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 multipurpose 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 movement and depth arrangement, as well as window sizing and closing.

About Gadgets

Gadgets can make the user's interaction with your application consistent, easy, and fun. There are two kinds of gadgets: predefined system gadgets and custom application gadgets. The system gadgets help to make the user interface consistent. They are used for dragging and arranging the depth of screens and for dragging, sizing, closing and arranging the depth of 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 and requesters.

There are four basic types of application gadgets:

- o Boolean gadgets elicit true/false or yes/no kinds of answers from the user.
- o Proportional gadgets are 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 that allows the user to enter integer values only.

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

Application gadgets are not directly supported in screens. Placing a gadget in a back-drop window allows you to receive gadget-related messages through that window's input/out channels. See chapter 8, "Input and Output Methods," for details.

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 that they remain proportional to the size of the window. Also, window gadgets can be located relative to one of the window's borders so that they move with the borders as the user shapes or sizes the window. If you want the gadget in the border, as are 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 gadget box; you define the dimensions of this area. Next, the user takes some action that happens according to the type of gadget. For a Boolean gadget, the user may simply execute 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 with the mouse or click the mouse button to move the knob by a set increment.

Although you attach a list of predefined application gadgets when you define a window requester 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 these messages.

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 for dragging and depth arrangement. The system gadgets for windows are for dragging, depth arrangement, sizing, and closing.

Standard gadgets have fixed, standard locations in screens and windows, as shown in table 5-1 (we also figure 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); you can attach these gadgets to your windows and let Intuition do the work of responding to the user's wishes.

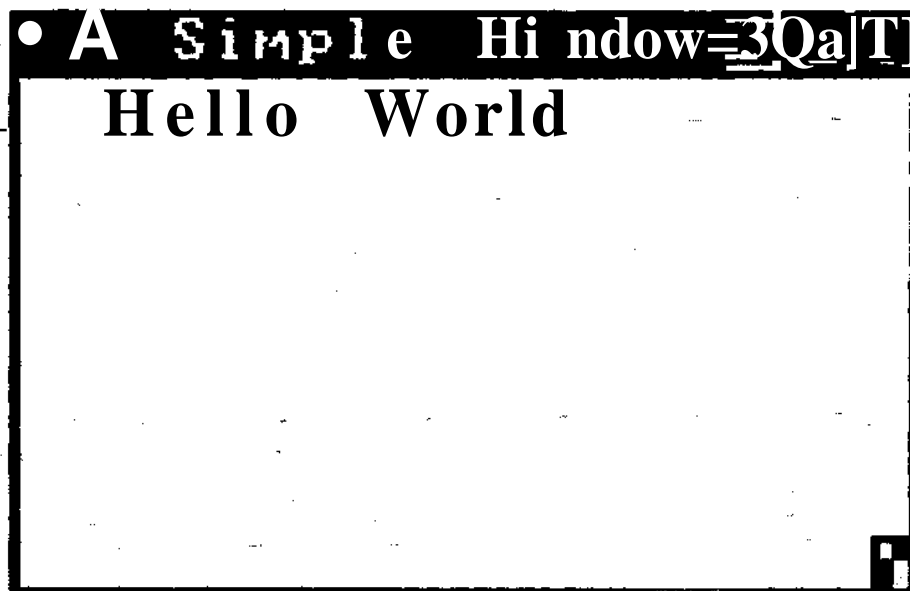


Figure 5-1: System Gadgets in a Low-resolution Window

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, which 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. A special `IDCMP` flag, `SIZEVERIFY`, allows you to hold off window sizing until you are ready for it. See chapter 8, "Input and Output Streams," 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 default depth arrangement of windows and screens is transparent to your program. The only time you might learn about it even indirectly is 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 window you open.

DRAGGING GADGET

Dragging gadgets are also known as drag bars because they occupy the entire title bar area that is not taken up by other gadgets. Users can slide screens up and down, much as some classroom blackboards can be moved, 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 is then up to the program to call GloseWindowQ when ready. You may want or need to take some actions before the window closes; for instance, you may want to 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, such as light switches or joysticks, as well as the freedom to create devices that satisfy your own unique needs.

HAND-DRAWN GADGETS

Visuals & draw your gadgets by hand, specify a series of lines for a simple line gadget, or use imagery at all.

Hand-drawn Gadgets

Because you are allowed to supply a hand-drawn image, there is 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.

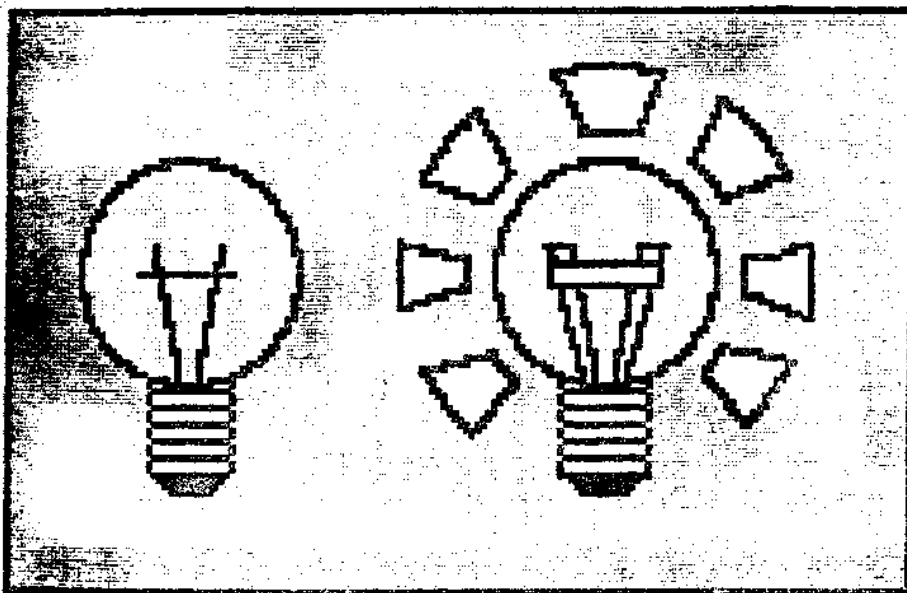


Figure 5-2: Hand-drawn Gadget — Unselected and Selected

To incorporate a hand-drawn image into your gadget by setting the GADGMAGE_Sag gadget Variable Flags to indicate that this gadget should be rendered as an image. Then you put 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. These lines can go around or through the select box of your gadget, and you can specify more than one group of lines, each with its own color and drawing mode. You create line-drawn 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-drawn imagery. It also shows an example of the complement-mode method of highlighting a gadget when it is selected. Furthermore, it shows additional text that has been included in the gadget imagery.

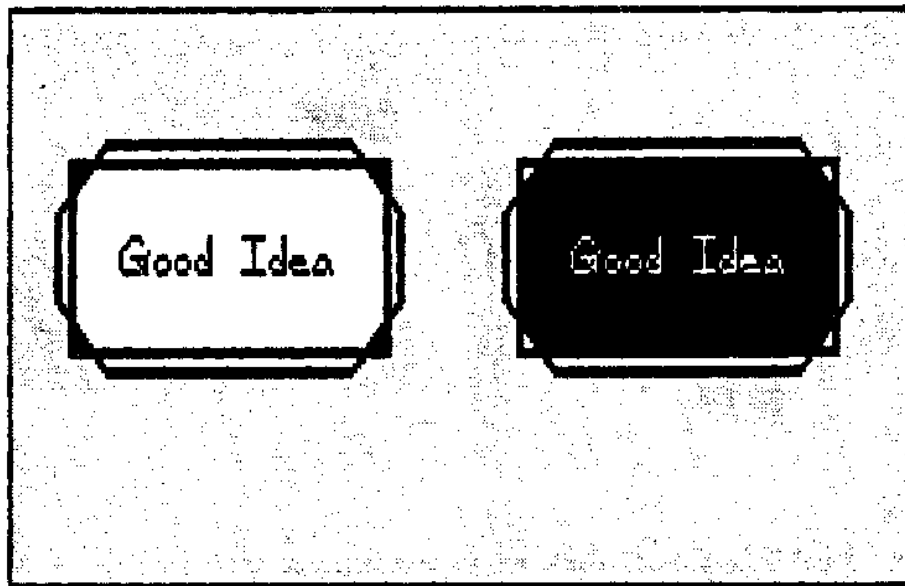


Figure 5-3: Line-drawn Gadget — Unselected and Selected

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-drawn imagery into your gadget by *not* setting the GADGIMAGE flag in the gadget's **Flags** variable, thus specifying that this is a **Border**, not an **Image**. Also, you put the address of your **Border** structure into the gadget variable `GadgetRender`.

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

Gadgets without Imagery

You can also create gadgets that have no imagery at all. For instance, you may want to follow the user's mouse activity without cluttering the display with unnecessary graphics. An example of such a gadget is the window and screen dragging gadget, which displays no actual imagery. The title bar itself sufficiently implies the imagery of the gadget.

You specify no imagery by *not* setting the gadget's GADGIMAGE flag and by setting the GftdgRender 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 the program 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, an IDCMP event of class GADGETDOWN will be received. If you set only this flag, the program will hear nothing more about that gadget until it is 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, the program will learn that the gadget was selected only 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 window close gadget—whose consequences may be serious. If you set the RELVERIFY flag, the program will learn about these events via an IDCMP message of this class GADGETUP. There are two main benefits to RELVERIFY: the unsure user gets one last chance to reconsider, and using RELVERIFY helps avoid casual errors caused by the user brushing against or resting fingers on the mouse button.

If you want the program to receive both a GADGETDOWN and GADGETUP message, set both the GADGIMMEDIATE and RELVERIFY 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 *current* 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 corner.*'

- o In the window-dragging gadget, the **LeftEdge** and **TopEdge** are always absolute in relation to the top left corner of the window. Also, Height is always an absolute quantity. Width of the gadget, however, is defined to be zero. When **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 in which you use the mouse movement to change some quality of the sound. You might also want to follow the mouse in a game in which you use it for aiming a weapon.

The broadcasts received differ according to the following flag settings:

- o If you set the GADGIMMEDIATE *and* RELVERIFY flags the program learns that the gadget was selected, gets some mouse reports (at least one), and finds out that the mouse button was released over the gadget.
- o If you set only the GADGIMMEDIATE flag, the program learns that the gadget was selected and get some mouse reports. Then the mouse reports will stop (when the user releases the select button), although the program will have no way of knowing for sure that this has happened.
- o If you set only the RELVERIFY flag, the program gets some mysterious, anonymous mouse reports (which may be just what you want to get) followed, perhaps, by a release event for a gadget.
- o If you set neither the GADGIMMEDIATE nor the RELVERIFY flag, the program gets only mouse reports. This may be exactly what you want the program to receive.

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 AddGadgetQ 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 makes sense to put a gadget only 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 is all border.

The border flags are called **RIGHTBORDER**, **LEFTBORDER**, **TOPBORDER**, and **BOTTOMBORDER**; they are set 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 is 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 feature works only on the first 32 gadgets in a list.

GADGET HIGHLIGHTING

•• a call to `ChangeGadgetAppearance` to let the user know that the gadget has been selected.

•• 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 do not want any highlighting, set the `GADGHNONE` bit.

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 case, 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) to select one of the pixels in the selected gadget, those pixels get changed to whatever color is in color register 1 (binary 01). Figure 5-3 shows an example of complement highlighting. Note that only the select box of the gadget is complemented; the text, which is outside 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

•• highlight by drawing a simple border around the gadget's select box, set the `&CHBOX` bit in the `Flags` field.

Highlighting with an Alternate Image or Alternate Border

You can supply alternate Image or Border imagery as highlighting. When the gadget is selected, the alternate Image or Border is displayed in place of the nonhighlighted imagery. If the nonhighlighted imagery is an Image, the highlight imagery should be an Image as well; the same is true for Border imagery. Figure 5-2 shows an example of this method of highlighting. For this highlighting method, you should set the `AlternateImage` or `AlternateBorder` 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 have submitted a gadget for Intuition to display, you can change its current enable state by calling OJUGadget() or OffGadgetQ. If it is a requester gadget, the requester must currently be displayed. If you use OnGadgetQ to enable a previously disabled gadget, its image is returned to its normal, nonghosted state.

BOOLEAN GADGET TYPE

Boolean gadgets are simple TRUE or FALSE gadgets. You can choose from two methods of selecting such gadgets—*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: `THGGLESELECT` and `SELECTED` have an effect upon toggle-selection:

- You need to set the `THGGLESELECT` flag in the Activation field of the Gadget Structure to mark the gadget to be toggle-selected.
- The `SELECTED` flag in the Gadget structure Flags determines the initial and current on/off state of a toggle-selected gadget. If `SELECTED` is set, the gadget will be selected. You can set the `SELECTED` flag before submitting the gadget to the system if you like. The program can examine this flag at any time to determine whether the user has selected this gadget.

A Boolean gadget is the simplest; the user, the application will hear about it. If it is selected, the application never knows. In this respect, Boolean gadgets differ from a string or proportional SSE&S, which always are set to some value, even if that value is the one initialized.

• PROPORTIONAL GADGET TYPE

Proportional gadgets are extremely flexible input devices. You can use one of these to get & proportional settings from the user or to display a proportional value to the user. In fact, of all, you can use the same gadget to accomplish both of these feats.

The user can adjust the KSC of a proportional gadget to specify how much of some measurable data of a feature is desired. For instance, the user may adjust a proportional gadget to specify & jam in a text file or a desired volume setting. The current setting of & proportional SSE may also be set by the program as an indicator of how much of some measurable attribute is visible or available. For instance, the proportional gadget of a window might show how many lines are currently displayed out of the pages in the text file. A graphics program may allow the user to move the amount of screen, and blue in a color, providing a proportional gadget. Each of the three: a graphics program would initialize these settings to design how much red, green, & blue already contained in the color. An audio program may allow the user to adjust the sound being produced by providing a gadget that allows the user to indicate the sine wave to see what the current volume is in relation to the highest and lowest possible volume settings.

Proportional gadgets can, in fact, do these things and more because they can take many shapes and sizes and get proportional 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 `pctTBEatoa`, 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 to center his gunsights on a fleeing enemy must 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 gadgets, you might want to initialize them to some current color. The program may read the values in the pots at any time after it has submitted the gadget to the user via Intuition. The values 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 set the step increment to a large number of lines, leaving 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 is fifteen lines long, and five 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 (the gadget box), 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 the mouse button with the cursor 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 the mouse button 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, such as 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 is 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 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 adjusts its size 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 one-third of a text file is being displayed in the window, the auto-knob fills one-third 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 one-third. For an example of such a scroll bar, see figure 5-4. This is yet another visual aid for the user, one that helps make 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 that 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.

```

    RealTime = (PropInfo.HorizPot + 1) * MAXSECONDS;

```

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

```

    Seconds = RealTime >> 16;

```

Because the lower 16 bits represent only a fraction, this value must be multiplied by some other meaningful constant if it is to mean something real. Because the fractional portion represents microseconds, and there are a million microseconds to the second, multiply the fraction by one million. Then, in the integer portion (the upper 16 bits), you will find the whole number of millionths of a second contained in RealTime. (By the way, in the lower 16 bits of the multiplication, which are shifted away into the bottomless bit bucket, is a fraction representing the fractional part of a millionth of a second contained in RealTime. To be technically accurate, you should test whether this fraction is greater than or equal to 0x8000 (one-half) and round your MicroSeconds result up if it is.)

```

    MicroSeconds = ((RealTime & 0xFFFF) * MILLION) >> 16;

```

You set up a proportional gadget as you do 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 Gadget-Render to point to an Image. In this case, you do not 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, your program 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 the program receives a message saying that the user has played with this gadget, the program can examine the **KNOBHIT** flag in the **PropInfo** structure. This flag indicates whether the user hit the knob or hit in the container but not on the knob itself. If the flag is set, the user 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 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 preset the initial string; by doing so you give the user the choice of editing the initial string or simply accepting the default initial string.

If a string gadget has an undo buffer, the current string is copied into the undo buffer when the user selects the gadget. The user can revert to this initial string at any time by typing "Right AMIGA - Q." (To type this key sequence, the user holds down the right AMIGA key while pressing the Q key.)

Because there is only one active gadget at a time, all string gadgets can share the same undo buffer as long as the undo buffer is as large as the largest input buffer.

You specify the size of the container into which the user types the string. Like the container for 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 is 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.

An 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 the program 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 deselected by the user (either by hitting the `RETURN` key or by using the mouse to select some other operation), the program 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 simple editing functions shown in table 5-2 are available to the user.

Table 5-2: Editing Keys and Their Functions

Key(3)	Function
← or →	Move the cursor around the current string.
SHIFT 4- or →	Move the cursor to the beginning or end of current string.
DEL	Delete the character under the cursor.
BACKSPACE	Delete the character to left of cursor.
RETURN	Terminate input and deselect the gadget. If the RELVERIFY activation flag is set, the program is notified that the user has selected this gadget.
Right AMIGA - Q	Undo (cancel) the last editing change to the string.
Right AMIGA - X	Clears the input buffer. The undo buffer is left undisturbed.

You can supply any type of image for the rendering of this gadget—Image, Border, or no image at all. For this release of Intuition, 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 input attributes and the font of the screen in which it appears. If you have not 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. 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, but 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 of Intuition, you need to preset 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.

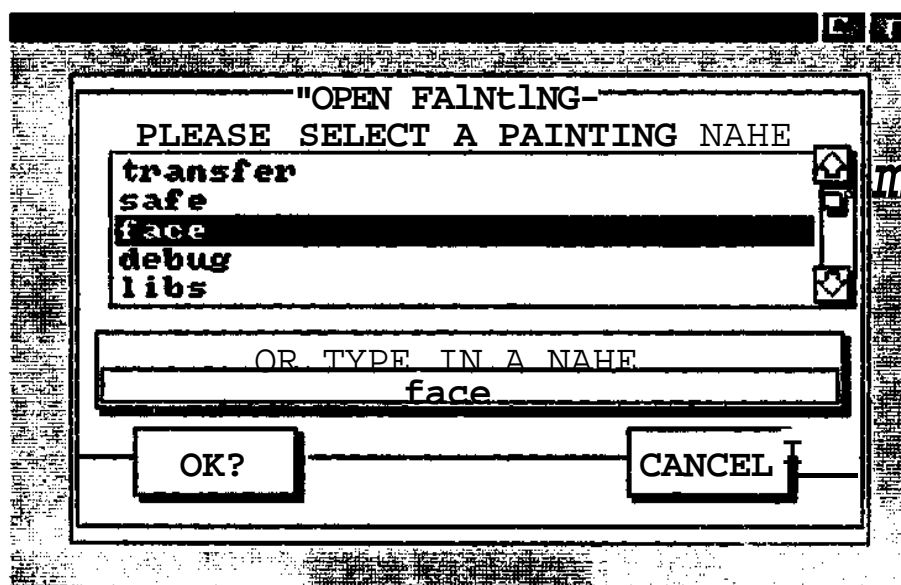


Figure 5-4: Example of Combining Gadget Types

If the scroll bar goes in the right border of the window, you may wish to place the system sizing gadget in the right border by setting the flag **SIZEBRIGHT** in the **NewWindow** structure. Remember that the sizing gadget has to fit in either the right or the bottom border. 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, follow these steps:

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

Each **Gadget** structure includes specifications for:

- o **An Image**, 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 absolute or relative to the current borders of the window or requester containing the gadget.
- o Width and height dimensions that are absolute or relative to the current size of the window or requester 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;
}
```

The variables and flags in the **Gadget** structure are explained below.

Next Gadget

This is 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 location 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 display element, according to how **GRELRIGHT** and **GRELBOTTOM** are set in the **Flags** variable (see 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 (see 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 *must* 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 requester or a Gimmezerozero window:

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 do not 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

Set this bit if this is a requester gadget; 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, **GADGMAGE**.

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.

Currently, Intuition ignores this field.

SpecialInfo

If this is a proportional gadget, this variable points to an instance of a **PropInfo** data structure. If this is a string or integer gadget, this variable points to a **StringInfo** data structure. 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.

Gadget©

This variable is strictly for your own use. Assign any value you would 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 would care to associate with this particular gadget. This variable is ignored by Intuition.

FLAGS

The 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

This flag complements all of the bits contained within this gadget's select box.

GADGHBOX

This flag draws a box around this gadget's select box.

GADGHIMAGE

This flag displays an alternate **Image** or **Border**.

GADGHNONE

Set this flag if you want no highlighting. -

GADGMAGE

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 is 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 preselect the on/off selected state for a toggle-selected gadget. If the flag is set, the gadget starts off being on and 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 **OffGadgetQ**.

You do not 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 state of the gadget (and its imagery) toggles each time it is hit.

You preset the selection state with the gadget **Flag** **SELECTED** (see above); the program later discovers the selected state by examining **SELECTED**.

GADGIMMEDIATE

Set this bit if you want the program 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 requesters. To make a requester go away, the user must 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, the program 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 preset the string gadget input buffer by putting an initial integer string in it.

ALTKEYMAP

This flag specifies that you have an alternate keymap. You also need to put a pointer to the keymap in the `StringInfp` structure variable `AltKeyMap`.

SPECIALINFO DATA STRUCTURES

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

ProplInfo Structure

This is the special data required by the proportional gadget.

```
struct ProplInfo
{
    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

In the **Flags** variable, these general-purpose flag bits can be specified:

AUTOKNOB

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

FREEHORIZ

If this is set, the knob can move horizontally.

FREEVERT

If this is set, the knob can move vertically.

KNOBHIT

This is set by Intuition 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 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 below.

You initialize the following variables and Intuition maintains them:

Buffer

This is a pointer to a buffer containing the start and final string. The string you write into this buffer must be null-terminated.

UndoBuffer

This is an 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 starting and final strings.

MaxChars

This must be set to the maximum number of characters that will fit in the buffer, including the terminating NULL.

BufferPos

This specifies the initial character position of the cursor in the buffer.

DiapPos

This specifies the buffer position of the first displayed character.

Intuition initializes and maintains these variables for you:

UndoPos

This specifies the character position in the undo buffer.

NumChars

This specifies the number of characters currently in the buffer.

DispCount

This specifies the number of whole characters visible in the container.

CLeft, CTop

This specifies the top left offset of the container.

LayerPtr

This specifies 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.

`AddGadget(AddPtr, Gadget, Position)`

This function adds a gadget to the gadget list of a window.

AddPtr is a pointer to the window.

Gadget is a pointer to the gadget.

Position is where the new gadget should go in the list.

`RemoveGadget(RemPtr, Gadget)`

This function removes a gadget from the gadget list of the specified window.

RemPtr is a pointer to the window from which the 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)`

This function enables the specified gadget.

Gadget points to the gadget you want enabled.

Ptr points to a Window structure.

Requester points to a requester or is NULL.

`OffGadget(Gadget, Ptr, Requester)`

This function^{*1} disables the specified gadget.

Gadget points to the gadget to be disabled.
Ptr points to a Window structure.
Requester points to a requester or is NULL.

Redraw the Gadget Display

The following function redraws all of the gadgets in the gadget list of a 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** structure.
Requester points to a requester or is NULL.

Modifying a Proportional Gadget

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

ModifyProp(Gadget, Ptr, Requester, Flags, HorizPot, VertPot, HorizBody, VertBody)

This function 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.

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 facilities.
- o Formatting capabilities.
- o Multiple fonts.
- o A general help facility.

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 a window and a 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 does not

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 can 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 (see figure 6-1).

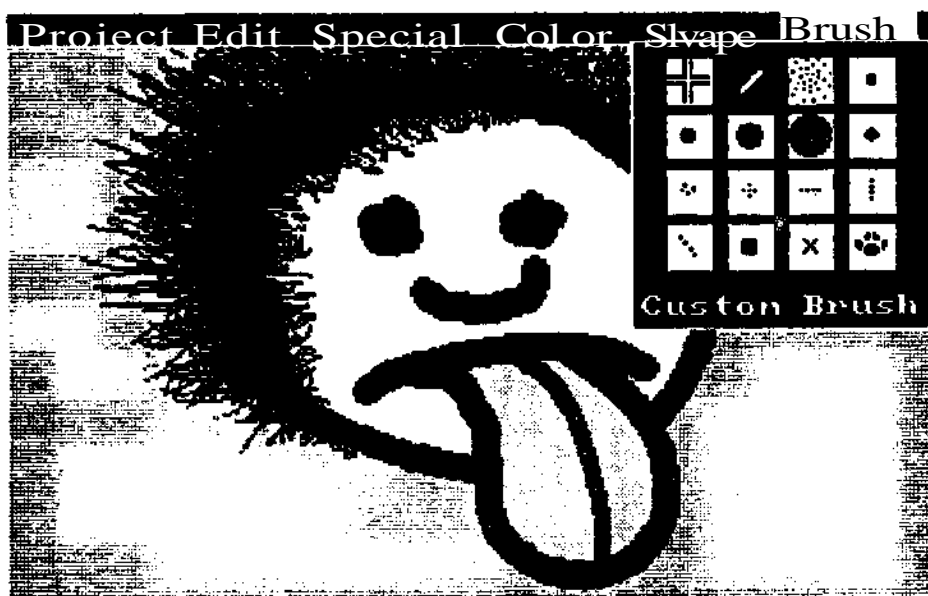


Figure 6-1: Screen with Menu Bar Displayed

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 subitem list, moving the pointer over the item reveals a list of subitems. The user moves the pointer over one of the

subitems 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 **SetMenuStripQ**. You must always remove every menu strip that you have submitted. When you want to remove the menu strip, you call **ClearMenuStripQ**. If you want to change the menu strip, you call **ClearMenuStripQ**, change the menu, and resubmit it with **SetMenuStripQ**.

The flow of events for menu operations should be:

1. **OpenWindowQ**.
2. Zero or more iterations of **SetMenuStripQ** and **ClearMenuStripQ**.
3. **CloseWindowQ**.

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

ABOUT MENU ITEM BOXES

The item box is **the** rectangle containing your menu items or subitems. You do not have to describe the size and location of the item or subitem boxes directly. You describe the size indirectly by how you place items and subitems. 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 subitems. See figures 6-2 and 6-3 for examples of item and subitem box structures.

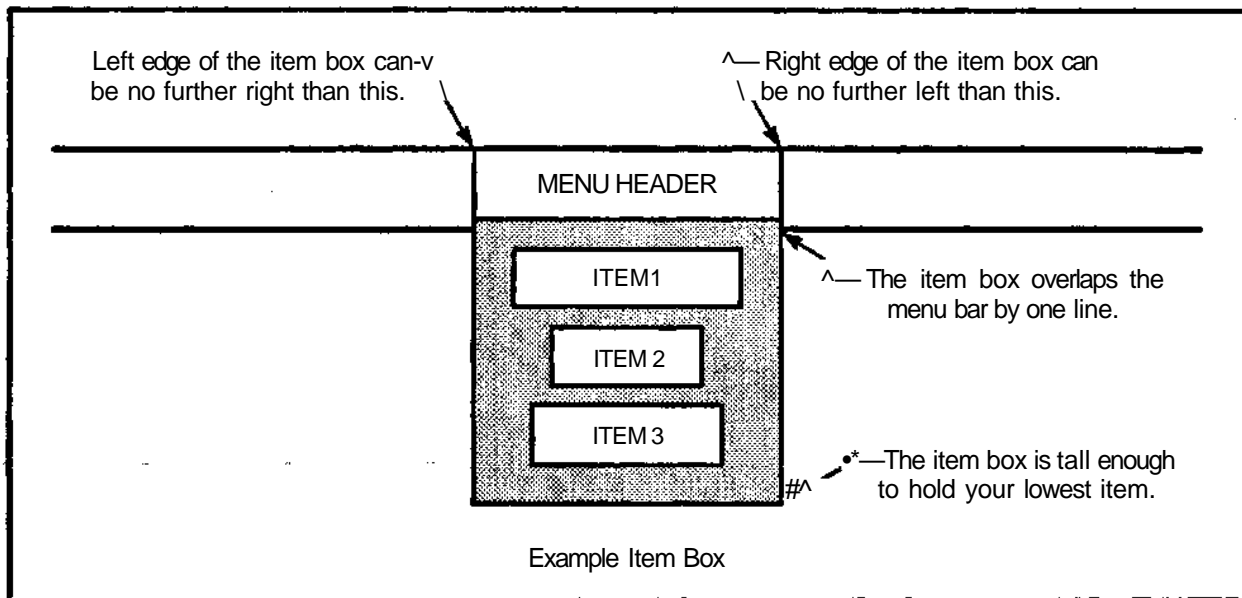


Figure 6-2: Example Item Box

The item box must start no further right than the leftmost position of the menu header's select box. It must end no further left than the rightmost position of the menu header's select box. The top edge of the menu box must overlap the screen's title bar by one line. The subitem box must overlap its item's select box somewhere.

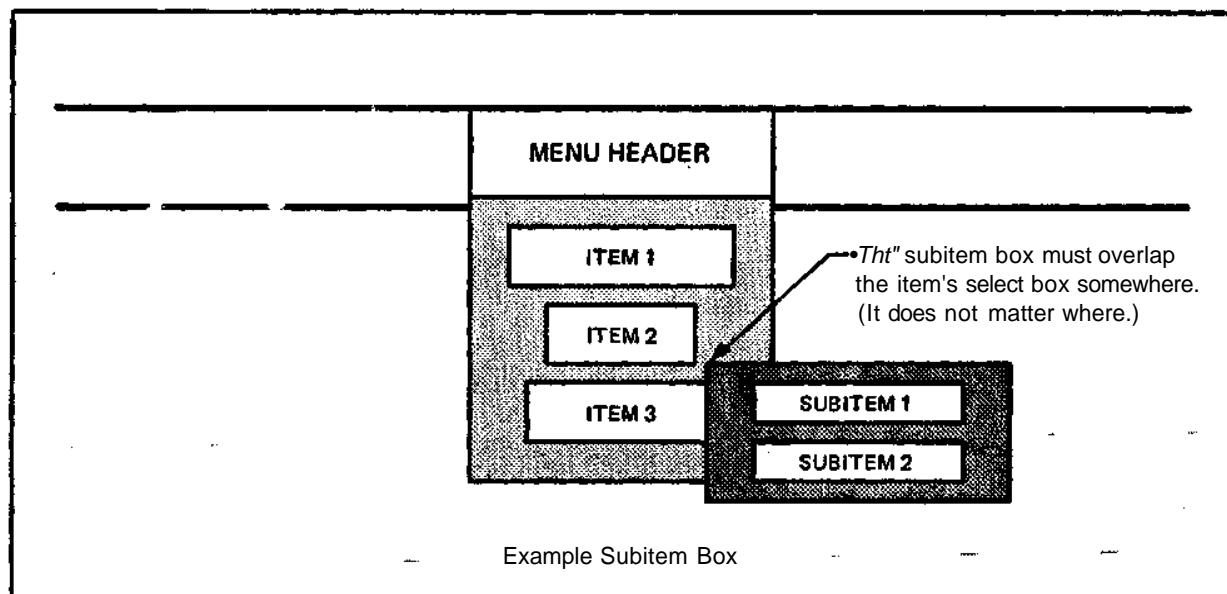


Figure 6-3: Example Subitem Box

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 Menu Item structure. If you set this flag, 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 (y/) or you can design your own and set a pointer to it in the **NewWindow** 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** amount of blank pixels 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, 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, 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, the **CHECKED** flag is set and the user cannot then reselect 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, 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 do not have the CHECKIT flag set has no effect.

It is 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, in which carrying the heavy sword excludes carrying any other items, the **MutualExclude** fields of the four items would look like this;

Heavy sword	0xFFFFE
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 IMAGERY

A *command-key sequence* is an event generated when 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 is associated with one of your menu items, Intuition will send the program 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, because 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 setting the COMMSEQ flag in the **Flags** variable of the MenuItem structure and by 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 a special AMIGA key icon rendered about one character span plus a few pixels from the right edge of the menu select box and 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 (see figure 6-4).

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.

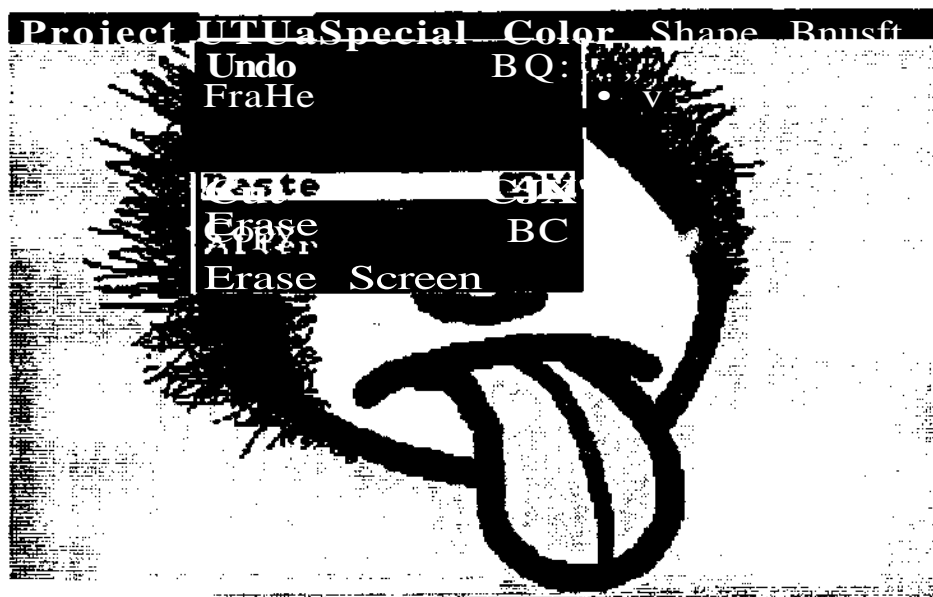


Figure 6-4: Menu Items with Command Key Shortcuts

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 the program disables a menu item that the user has already selected with extended select. The program will receive a `MENUPICK` message for that item, even though it thinks it has already disabled it. The program will have to ignore items that it knows are already disabled.

You use the routines **OnMenu()** and **OffMenuQ** to enable and disable individual subitems, items or whole menus. These routines check if the user is using the menus and whether the menus need to be redrawn to reflect the 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 **ClearMenuStripQ**. 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 **SetMenuStripQ** with the new menu.

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 **MENUPICK** telling which menu item has been selected. If one of your items has a subitem list, the menu number your program receives for that item includes some subitem 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, the program receives a message with the menu number equal to **MENUNULL**. In this way, the program 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 **Menuitem** data structure. After taking the appropriate action for the item selected by the user, the program should check the **NextSelect** field. If the number there is equal to the constant **MENUNULL**, there is no next selection. However, if it is not equal to **MENUNULL**, the user has selected another option after this one. The program should process the next item as well, by checking its **NextSelect** field, until it finds 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 MENU PICK 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 subitem 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 subitem 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 subitems attached to that item, SubNumber will equal NOSUB.

```

The menu number received by the program describes either MENUNULL or a valid menu selection. If it is 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, the program always gets one menu specifier and one menu item specifier. If a given menu item has a subitem, a subitem specifier will also be received. Just as it is not possible to select a menu, it is not possible to select a menu item that has a list of subitems. The user must select one of the options in the subitem list before the program ever hears 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 ItemAddressQ translates a menu number into an item address.

HOW MENU NUMBERS REALLY WORK

The following is a description of how menu numbers really work. It 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. To extract the item number from the variable **MenuNumber**, for example, call ITEMNUM(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 subitem 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

1

i

- > These bits are for the menu number.

II

> These bits are for the menu items within the menu.

i

> These bits are for the subitems within the menu items.

Thus, for each level of menu item and subitem, up to 31 pieces can be specified. There are 63 item pieces that you can build under each menu, which is a lot, especially with 31 subitems 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 1,953 items. You should not 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
|           |           |
|           I           > Menu number 0
|
> Menu item number 0x25 = 37

```

> Subitem 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 your program the opportunity to react before menu operations take place and, optionally, to cancel menu operations. The second, `RMBTRAP`, allows the program to trap right mouse button events for its own use.

Menu-verify

Menu-verify is one of the Intuition verification functions. These functions allow you to make sure that your program is prepared for some event before it takes place. Using menu-verify, 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 the program is doing something special to the display of a custom screen, and you want to make sure it is completed before menus are rendered.

Any window can access the menu-verify feature by setting the `MENUVERIFY` flag in the **NewWindow** structure when opening the window. When your program gets a message of class `MENUVERIFY`, menu operations will not proceed until the program replies to the message.

The active window gets special menu-verify treatment. It is allowed to see the menu-verify message before any other window and has the option of canceling 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.

Your program can tell whether or not it is in the active window by examining the code field of the MENUVERIFY message. If the code field is equal to MENUWAITING, **your** window is not the active one and Intuition is simply waiting for you to verify that menu operations may continue. However, if the code field is equal to MENUHOT, your window is the active one and it gets to decide whether or not menu operations should proceed. If the program does *not* want them to proceed, it should change the code field of the message to MENCANCEL 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 do not want any menu operations at all for your window. Whenever the user presses the right button while your program's window is active, the program 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:

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

To create the menu structures, you need to choose:

- 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 Which items have subitems. For the subitems, you make the same decisions as for the menu items.

Menu strips are constructed of three components: menus, menu items, and subitems. They use two data types: Menu and MenuItem. Subitems 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 The 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 subitems (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 subitem, uses the same data structure as the menu item. Subitems are identical to menu items except that the subitem's location is relative to its menu item's select box and the subitem's subitem link is ignored.

MENU STRUCTURES

The specifications for the menu structures are given below. **Menus** are the headers that show in the menu bar, and **Menuitems** are the items and subitems 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 **TopBorder** 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

This flag indicates 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 disabled by calling **OnMenuQ** or **OfiMenu()**.

MIDRAWN

This flag indicates whether or not this Menu's items are currently displayed to the user.

Menu Name

This is a pointer to a null-term'mated character string that is printed on the screen's title bar starting at the **LeftEdge** of this Menu's select box and at the **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 subitems):

```
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

This field is a 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 the 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 does not mean that you cannot 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 an instance of an **Image** structure with image data. Your program tells 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 subitem list, this variable should point to the first subitem in the list. Note that if this item is a subitem, 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, no other items

were selected; otherwise, there is 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 a checkmark to precede this item 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; you 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 OffMenuQ. If this item has subitems, all of the subitems 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

This complements¹ all of the bits contained by this item's select box.

HIGHBOX

This draws a box outside this item's select box.

HIGHIMAGE

This displays the alternate imagery in **SelectFill** (textual or image).

HIGHNONE

This specifies no highlighting.

The following two flags are used by Intuition:

ISDRAWN

Intuition sets this flag when this item's subitems 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 subitems.

- 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 subitems.
- o To enable or disable a *single item* and all subitems attached to that item, set the item component of the menu number to your item's ordinal number. If your item has a subitem list, set the subitem component of the menu number to NOSUB. If your item has no subitem list, the subitem component of the menu number is ignored.
- o To enable or disable a *single subitem*, set the item and subitem components appropriately.

OnMenu(Window, MenuNumber)

This function enables the given menu or menu item.

OfiMenu(Window, MenuNumber)

This function 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.

Chapter 7

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 are 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.

About Requesters

Requesters (see figure 7-1) are like menus in that 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 are that requesters *require* a response from the user and that while the requester is in the window, the window locks out all user input. The requirement of a user response is virtually the only restriction placed on your program's use of requesters.

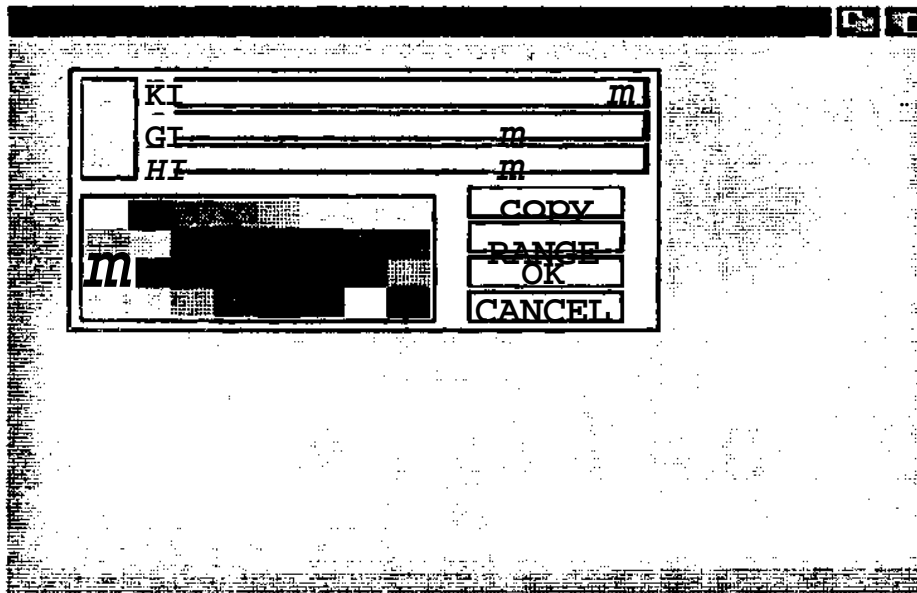


Figure 7-1: Requester Deluxe

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 is 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 made up of 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 use caution, however, since the requester obscures part of the display memory of the window. Fortunately, there are several ways to monitor the comings and goings of requesters, which your program can use to ensure that it 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 submitting the requester to Intuition for display.

M

You can select a requester rendered by Intuition in two ways. 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. Alternatively, if the requester requires only a simple yes or no answer from the user, you can use the special `AutoRequestQ` 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 to which the user does not really need to respond. 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 `DisplayBeepQ` in chapter 11, "Other Features."

RENDERING REQUESTERS

There are two ways of having complex requesters rendered—you can supply Intuition with enough information to do the rendering for you, or you can supply your own completely customized 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 gadget 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.

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.

NOTE: The current release of Intuition does not support the POINTREL feature.

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 exactly like other requesters with one exception: it is displayed only when the user double-clicks the mouse menu button. You give the user the ability to bring up a double-menu requester by calling `SetDMRequestQ`. 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 throw it away), you can unlink it from the window by calling `ClearDMRequestQ`.

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 **Activation** field. 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 must supply a list of gadgets, because you must still define the select box, highlighting, and gadget type for each gadget even though the gadget image information is ignored.

EDCMP 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 the program receives a `REQVERIFY` message, the requester **will not** be rendered **until** the program replies to the message.

DOUBLE-MENU REQUESTERS

A double-menu requester is exactly like other requesters with one exception: it is displayed only when the user double-clicks the mouse menu button. You give the user the ability to bring up a double-menu requester by calling `SetDMRequestQ`. 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 throw it away), you can unlink it from the window by calling `ClearDMRequestQ`.

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 `Activation` field. 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 must supply a list of gadgets, because you must still define the select box, highlighting, and gadget type for each gadget even though the gadget image information is ignored.

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 the program receives a `REQVERIFY` message, the requester will not be rendered until the program replies to the message.

REQSET

With this flag set, your program will receive a message when the first requester opens in your window.

REQCLEAR

With this flag set, your program will receive a message when the last requester is cleared from the window.

You set these flags when you call **ModifyIDCMP()** or create a **NewWindow** structure. See chapter 8, "Input and Output Methods," for further information about these IDCMP flags.

A SIMPLE, AUTOMATIC REQUESTER

For a simple requester that prompts the user for a positive or negative response, you can use the **AutoRequestQ** function (see figure 7-2). 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's choice of "Yes," "True," "Retry," and similar responses. Likewise, the negative text is associated with the user's 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.

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.

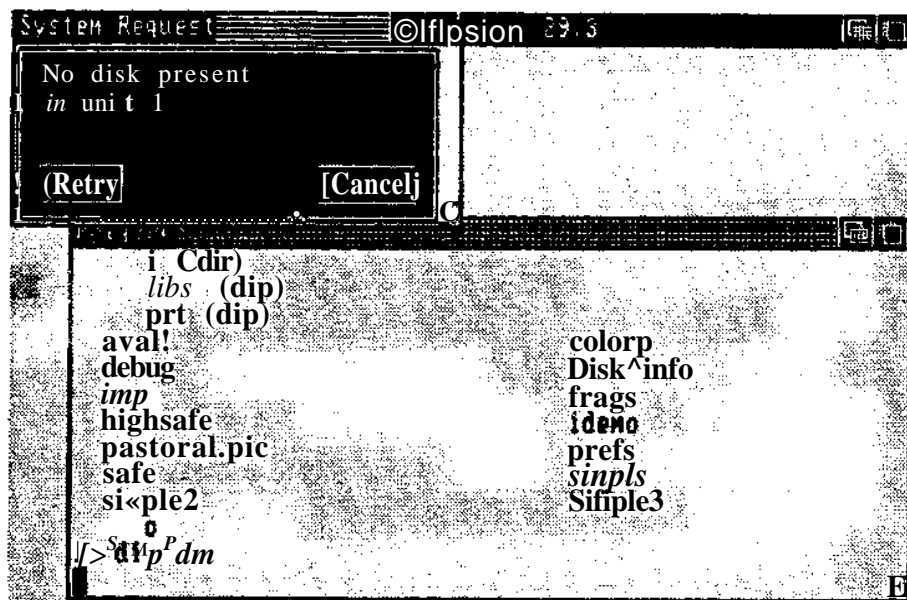


Figure 7-2: A Simple Requester Made with AutoRequestQ

The AutoRequestQ function calls BuildSysRequestQ to construct the simple requester. Your program can call BuildSysRequestQ directly if you want the program to use the simple requester and to monitor the requester itself. All gadgets created by BuildSysRequestQ have the following gadget flags set:

BOOLGADGET

It is a Boolean TRUE or FALSE gadget.

RELVERIFY

The program receives a broadcast if this gadget is activated.

REQGADGET

This flag specifies that this is a requester gadget.

TOGGLESELECT

This flag specifies that this is a toggle-select type of gadget.

Using Requesters

To create and use a requester, follow these steps:

1. Declare or allocate a Requester structure.
2. Fill out the Requester with your specifications for gadgets, text, borders, and imagery.
3. If you are using the IDCMP for input, decide whether to use the special functions provided.
4. Display the requester by calling either RequestQ or SetDMRequestQ.

REQUESTER STRUCTURE

To create a requester structure, follow these steps:

1. Fill in the values you need in the structure.
2. Set up a gadget list.
3. Supply a BitMap structure if this is a custom requester.

The specification for a Requester structure follows.

```
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 Layer *ReqLayer;
    UBYTEReqPad1[32]
    struct BitMap *ImageBMap;
    struct Window *RWindow;
    UBYTE ReqPad2[36]
```


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

These fields 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

This field is 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

This field is a pointer to an optional **Border** structure for the drawing lines around and within your requester.

ReqText

This field is 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 but is positioned off-window.

REQACTIVE

This flag is set or cleared by Intuition based on whether or not this requester is currently being used.

SYSREQUEST

This flag is 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.

ReqLayer

This contains the address of the **Layer** structure used in rendering the requester.

ImageBMap

This flag is a pointer to the custom bit-map for this requester. If you are not supplying a custom bit-map for this requester, Intuition ignores this variable.

If you are supplying a custom bit-map, the **PREDRAWN** flag must be set.

RWindow

This is a system variable.

ReqPad1, ReqPad2

These are reserved for system use.

The following sections describe the differences in the **Requester** structure between requesters rendered by Intuition and custom-bit-map requesters.

Requesters Rendered by Intuition

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 drawing takes place.

For example, the following **Requester** structure 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,                /* ReqLayer */
    {NULL},              /* pad */
    {NULL},              /* BitMap */
    NULL,                /* RWindow */
    {NULL},              /* pad */
};
```

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 a 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 **AutoRequestQ** for the automatic, simple Boolean requester that Intuition will build for you:

Window

This is a pointer to the window in which the requester is to appear.

BodyText

This is a pointer to an **IntuiText** structure that explains the purpose of the requester.

PositiveText

This is a pointer to the **IntuiText** structure containing the positive response text.

This field can be NULL if there is no positive response.

Negative Text

This is a pointer to the **IntuiText** structure containing the negative response text.

PositiveFlags

These are flags for the IDCMP for positive external events that will satisfy the request.

NegativeFlags

These are flags for the IDCMP for negative external events that will satisfy the request.

Width, Height

These specify the size of the rectangle enclosing the requester.

REQUESTER FUNCTIONS

A brief rundown of the requester functions follows.

Submitting a Requester for Display

The following function submits regular requesters to Intuition for display:

Request(Requester, Window)

This function displays a requester in the specified window.

Double-menu Requesters

The following functions affect double-menu requesters:

SetDMRequest(Window, Requester)

This function attaches a requester to the double click of the mouse menu button.

ClearDMRequest(Window, Requester)

This function unlinks the requester from the window and stops the user from bringing it **up**.

Removing a Requester from the Display**EndRequest(Requester, Window)**

This function erases a requester invoked by the user or application and resets the window. It removes only the one requester named.

The Easy Yes-or-No Requester

The following function automatically builds, displays, and gets a negative or positive response from a requester:

**AutoRequest (Window, BodyText, PositiveText, NegativeText
PositiveFlags, NegativeFlags, Width, Height)**

This function 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 in situations that require the user to take some immediate action; for instance, when the application has experienced a fatal error or the system has or is about to crash. System alerts are managed entirely by Intuition. (See figure 7-3 for an example of an alert.)

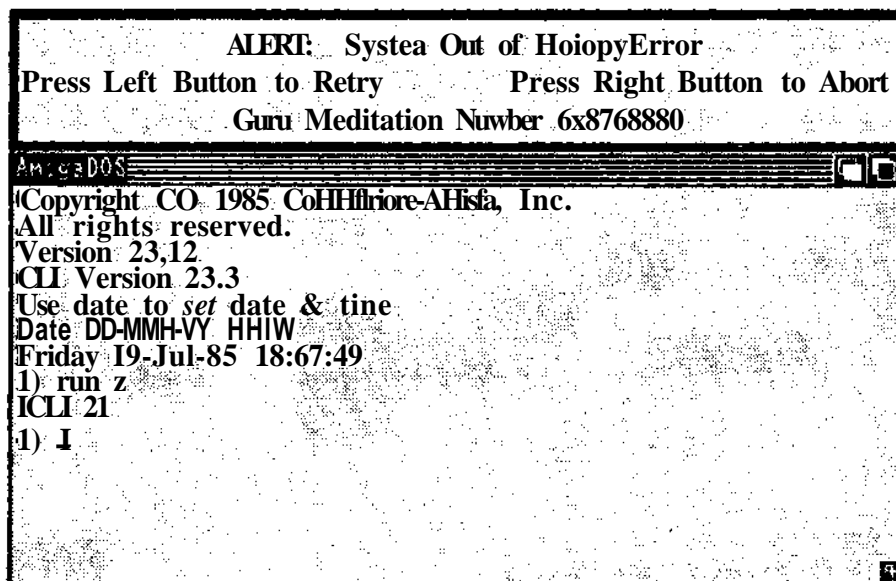


Figure 7-3: The "Out of Memory" Alert

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 is 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 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. **Display Alert ()** 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 that is made up of one or more substrings. Each substring contains the following:
 - o 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.
 - o The second component is the text itself. The string must be null-terminated (it ends with a zero byte).
 - o The last component is the continuation byte. If this byte is zero, this is the last substring in the message. If this byte is non-zero, there is another substring in this alert message.
- o The last argument, **Height**, tells Intuition how many display lines are required for your alert display.

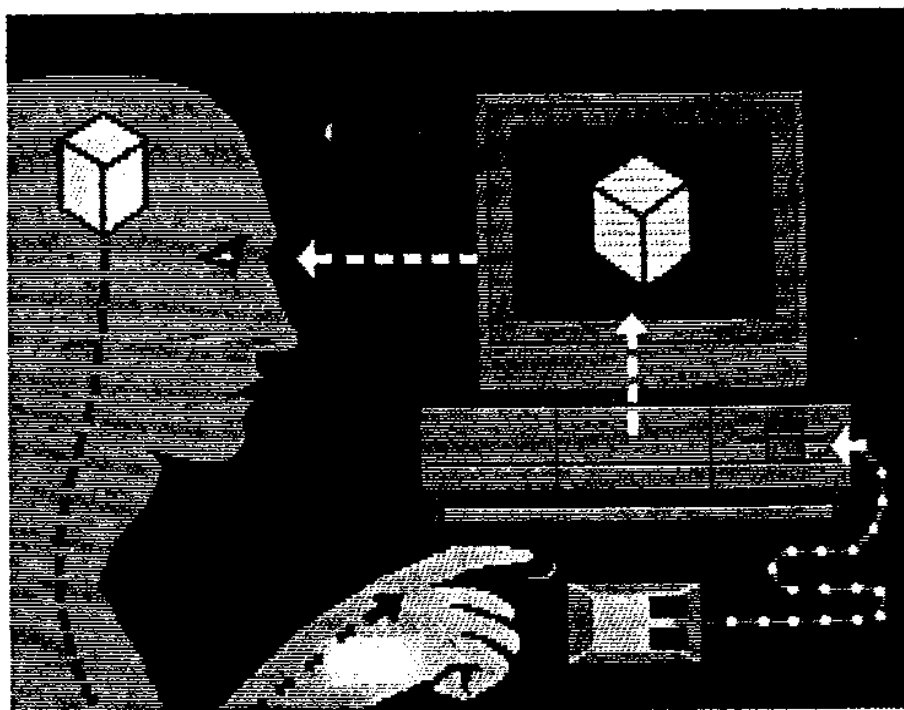
Chapter 8

INPUT AND OUTPUT METHODS

An Overview of Input and Output

From the Intuition point of view, information flows through the system in the following steps (see figure 8-1):

- 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 such as keyboards, mice, graphics tablets, and light pens. These input devices create input signals that enter the Amiga through several different ports.
- o Inside, these input signals are merged 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 often involves creating 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 that are transmitted to the brain, thus completing the cycle.



% Figure 8-1: Watching the Stream

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 one of the mouse buttons, or types on the keyboard, the input device detects it and constructs an `InputEvent` (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, 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). All of these events are merged 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, such as "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, such as 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, such as 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, 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 the program has no 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 preprocessing of the data. The other can supply you with standard terminal input functions, buffers, and data representations. The paths are explained below:

- o Intuition's Direct Communications Message Ports system (IDCMP) makes standard Amiga Exec message communications easily available for you and 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 gives you "cooked"* input data, including key-code conversions to ASCII and conversions to ANSI escape sequences (Intuition-generated events, such as CLOSE WINDOW, will be translated into escape sequences).

When you want your program 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. Also, the image functions provide a new method of object-oriented rendering.
- o Outputting text via the console device, which formats text with special text primitives such as `ClearEndOfLineQ` and text functions such as automatic 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 both as a source for input and as a mechanism for output. It is 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, bit-maps, 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 do not 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 following description of how I/O flow works with (and around) your program is actually a super-simplified model of how system-wide I/O really works, but it is a true representation of I/O at the microcosmic level of your program.

In the illustrations that follow, the input device is found at the top of the diagram. In this device mouse, keyboard, and other input events are merged into a single stream of input events, which is then submitted to Intuition for further processing.

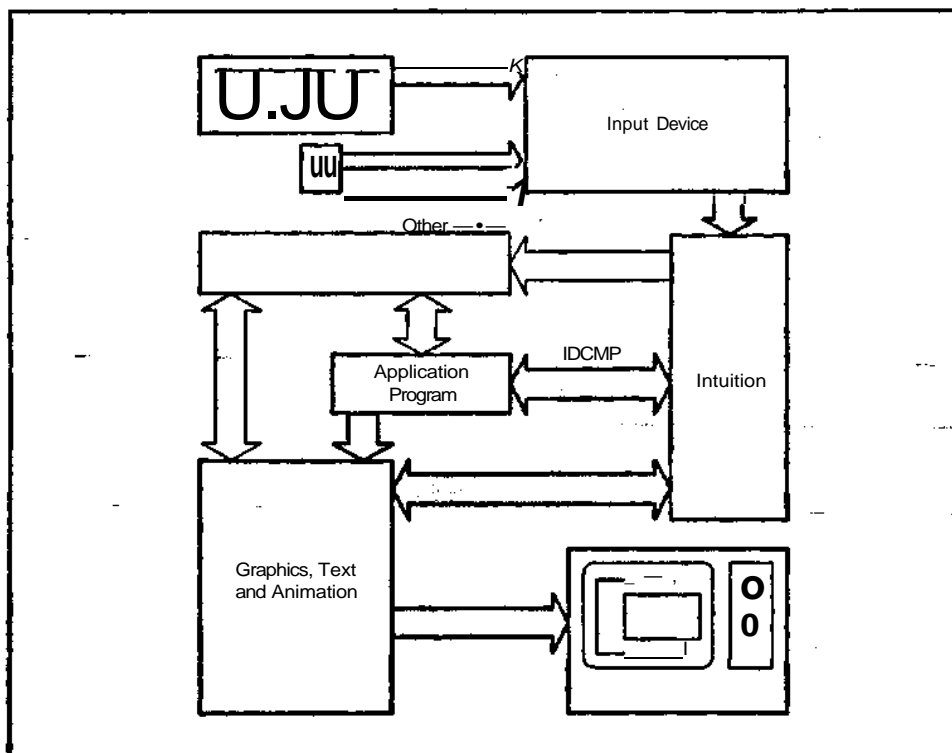


Figure 8-2: Input from the IDCMP, Output through the Graphics Primitives

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. Your program can, for instance, learn about gadget events and get notification that it should stop writing to its window (the IDCMP flags `SIZEVERIFY` and `REQVERIFY`), but the program may not want to learn about other mouse or keyboard events. If you set up the program to learn about raw keyboard events through the IDCMP, note that the key codes received come straight from the keyboard to the program. These keycodes are as raw as they get, although the IDCMP also provides the special **Qualifier** field to assist your translations. Alternatively, you can receive keyboard events translated into ASCII (or some other standard). 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 8-3 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.

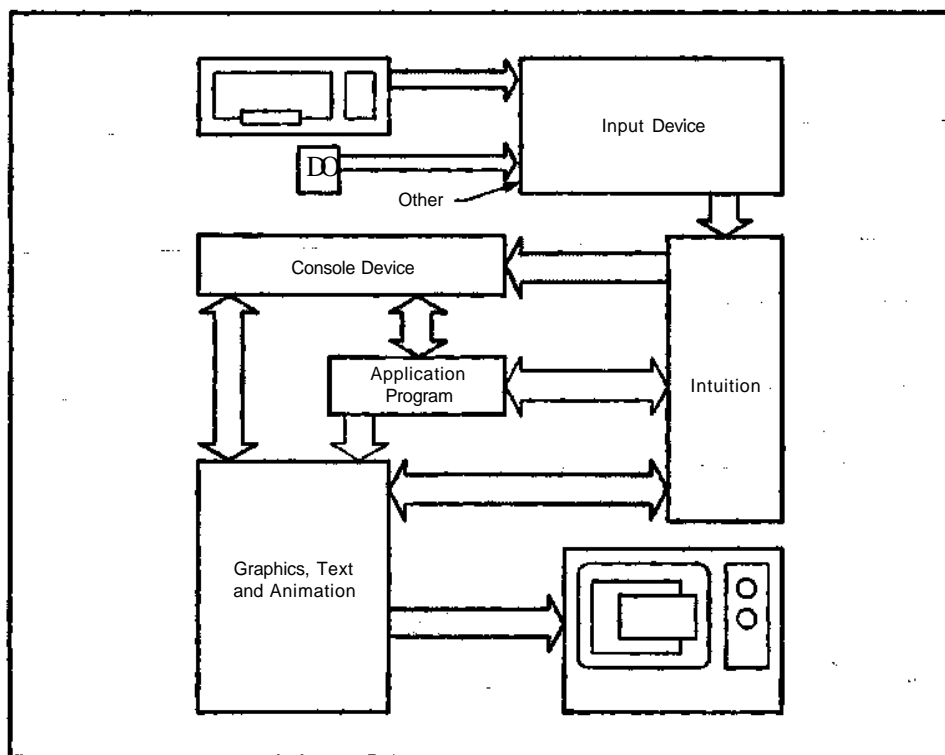


Figure 8-3: Input and Output through the Console Device

Figure 8-4 shows a complex program that needs the features of both the console device and the IDCMP. An example might be 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).

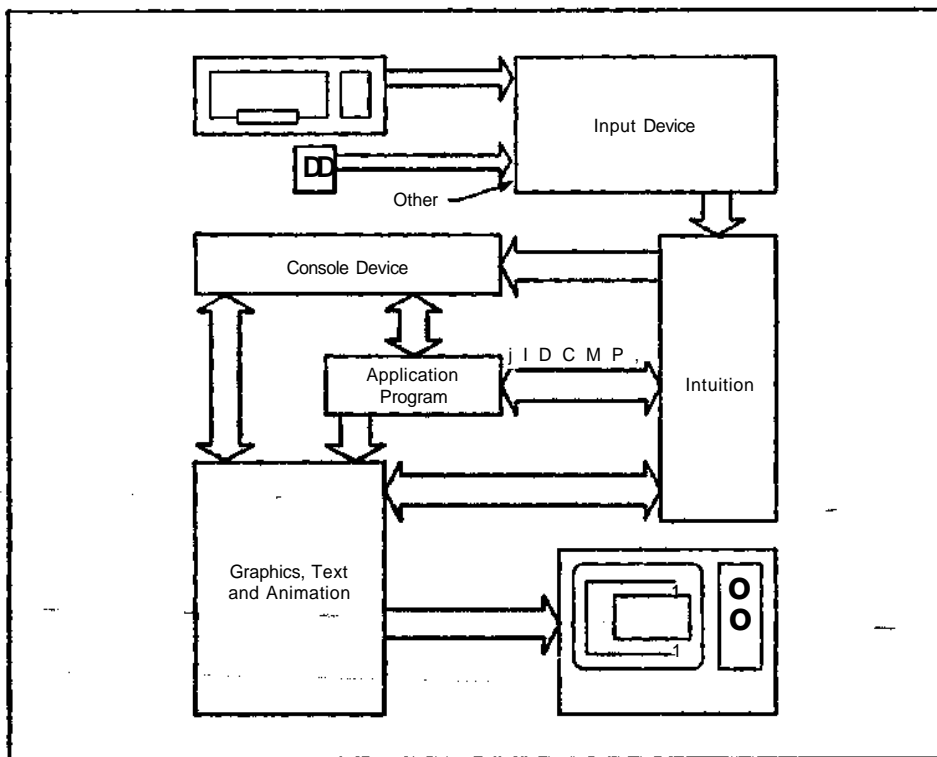


Figure 8-4: Full-system Input and Output (a Busy Program)

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) that 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.

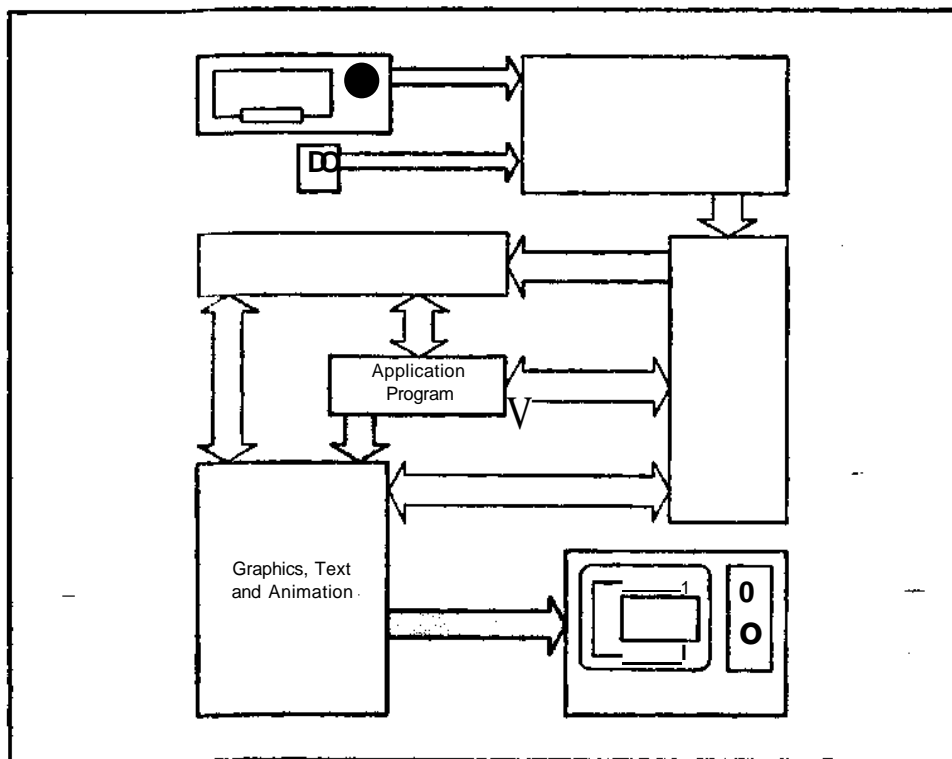


Figure 8-5: Output Only

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, certain useful Intuition features, most notably the verification functions (described under "IDCMP Flags" below), 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 interprocess communications in the Amiga multitasking environment. To open these ports automatically, you set IDCMP flags in the **NewWindow** structure. To open or close them later, you call **Modify ID CMP** (), which allocates or deallocates message ports or changes 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. 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.

If you have a message port that you have already created, you can have Intuition use that port to communicate with you. This is described below.

CAUTION: If you attempt to close the IDCMP, either by calling **ModifyIDCMFQ** 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. When you attempt to **ReplyQ** after the close, you will get to watch the Amiga FIREWORKSJDISPLAY 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 has been extended to include Intuition-specific information. The **ExecMessage** field in the **IntuiMessage** is used by Exec to manage the transmission of the message. The Intuition extensions of the **IntuiMessage** are used to transmit all sorts of information to your program. Here is 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 data in this field is maintained by Exec. It is used for linking the message into the system and broadcasting it to a message port.

Class

This is a ULONG variable whose bits correspond directly with the IDCMP flags.

Code

This is a USHORT variable whose bits contain special values, such as 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 is no special meaning for the code field, and it is merely a copy of the code of the **InputEvent** initially sent to Intuition by the input device. When this message is of class RAWKEY, this field has the raw key code generated by the keyboard device. When this message is of class VANILLAKEY, this field has the translated character.

Qualifier

This contains a copy of the **ie_Qualifier** field that is transmitted to Intuition by the input device. This field is useful if your program handles raw key codes, since the **Qualifier** tells the program, 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. The 32 bits allocated to the **Seconds** variable means that the Amiga clock can run for 139 years before wrapping around to zero again.

IAAddress

This 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.

IDCMP Window

This contains the address of the window to which this message pertains.

SpecialLink

This is 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 opening 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

This flag causes reports about mouse-button up and down events to be sent to you, if these transitions do not mean something to Intuition. When your program receives a **MOUSEBUTTONS** class of event, it 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 your program does not hear about it. Also, the only way your program 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 your program 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 if some gadget is selected with the **FOLLOWMOUSE** flag set.

DELTAMOVE

When this flag is set, mouse movements are reported as deltas (amount of change from the last position) rather than as absolute positions. This flag works in conjunction with the **MOUSEMOVE** flag. Note that delta mouse movements are reported even after the Intuition pointer has reached the limits of the display.

Gadget flags:

GADGETDOWN

Your program will receive a message of this class, when the user selects a gadget that was created with the GADGIMMEDIATE flag set.

GADGETUP

When the user releases a gadget that was created with the flag REL VERIFY set, your program will receive a message of this class.

CLOSEWINDOW

If the user has selected your window's close gadget, the message telling the program about it will be of this class.

Menu flags:

MENUPICK

This flag indicates that 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 your program to verify that it has 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 so 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 you want your application to make sure that other rendering to its window has ceased before a requester is rendered in 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 then WaitQs for the application to Reply() before drawing the requester in 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 multitasking environment.

See the "Verification Functions" section below for some things to consider when using this flag.

Window flags:

NEWSIZE

Intuition sends your program a message after the user has resized the window. After receiving this, the program can examine the size variables 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 for which the SIMPLEL.REFRESH or SMARTJtEFRESH type of refresh has been selected.

SIZEVERIFY

You set this flag if your program is drawing to the window in such a way that the drawing must be finished 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 the program replies.

See the "Verification Functions" section below for some things to consider when using this flag.

ACTIVIEWINDOW and INACTIVIEWINDOW

Set these flags to discover when your window becomes activated or inactivated.

Other flags:

VANILLAKEY

This is the raw keycode RAWKEY event translated into the current default character keymap of the console device. In the USA, the default keymap is ASCII characters. When you set this flag, you will get IntuiMessages with the Code field containing a character representing the key struck on the keyboard.

RAWKEY

Keycodes from the keyboard are sent in the Code field. They are raw keycodes, so you may want the program to process them.

The Qualifier field contains the information generated by the input device about this key.

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 make sure your program finds out about it by setting this flag.

When your program gets a message of class NEWPREFS, it can call the procedure GetPrefs() to get the new Preferences.

NOTE: Everyone who sets this flag will learn about these events, not just the active window.

DISKINSERTED and DISKREMOVED

When the user inserts or ejects any disk with any drive, the program will be told about the event if either or both of these flags are set.

NOTE: Everyone who sets these flags will learn about these events, not just the active window.

INTUITICKS

This gives you simple timer events from Intuition when your window is the active one; it may help you avoid opening and managing the timer device. With this flag set, you will get only one queued-up INTUITICKS message at a time. If Intuition notices that you've been sent an INTUITICKS message and haven't replied to it, another message will *not* be sent.

Intuition receives timer events ten times a second (approximately).

Verification Functions

SIZEVERIFY, **REQVERIFY**, and **MENUVERIFY** are exceptional in that Intuition sends an **IntuiMessage** and then waits, by calling the Exec message port function **WaitQ**, for the application to reply that it is all right to proceed. The application replies by calling the Exec message passing function **ReplyMsg()**.

The implication is that the user requested some operation but the operation will not happen immediately and, in fact, will not happen at all until your application says it is safe. Because this delay can be frustrating and intimidating, you should strive to make the delay as short as possible. Your program should always reply to a verification message as immediately as possible.

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 when you define a window with IDCMP flags or call **ModifyIDCMP()**. If the **WindowPort** is not already opened when one of these functions is called, it will be allocated and initialized. The **UserPort** is checked separately to see whether it is already opened. Intuition will send messages to your program via the **UserPort** and will receive replies 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 **AllocSignalQ** to get a signal bit. Since your task called **OpenWindowQ**, 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. You might do this in an environment in which your program is going to open several windows and you want the program 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 **ModifyIDCMPQ** with the flags set as you wish. When Intuition sees that the **UserPort** 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 **CloseWindowQ**, 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. You can receive and respond to 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;
Reply Msg(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

The following discussion is a 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 open it directly via a call to OpenDeviceQ. 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 simple and convenient. Also, you get special line-edit capabilities when opening an AmigaDOS console.

Opening a console as an AmigaDOS file has, however, two limitations. File I/O requires more processing overhead than going straight to the console device. Also, your program must be in an AmigaDOS environment (AmigaDOS must be active), which will not be the case for those of you who want your applications 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 do not have the special line-edit capabilities.

USING THE AMIGADOS CONSOLE

Two sorts of input can be obtained 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 **OpenQ**, 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 **WriteQ**. When you have finished with console I/O, you should call **CloseQ** 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 **OpenDeviceQ**, which opens the console device and attaches it to your window. The call to **OpenDeviceQ** 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 **ioJData** field to point to your *buffer*. A buffer is a block of memory that will be used to receive the characters from the console device.
- o Set the **ioJLength** 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 **SendIOQ** or **DoIOQ** function to read in any characters that are waiting to be read. The difference between **SendIOQ** and **DoIOQ** is that **SendIOQ** is *asynchronous*, which means that while the console device monitors the keyboard, the program does other processing and checks later to see whether or not the user has typed something. **DoIOQ**, on the other hand, is *synchronous*, which means that when **DoIOQ** is called control does not return to the program until the user has typed something.

After the call to one of the input routines, your program 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 is 0x9B in hex). This character is also known as the *control sequence introducer*, or **CSI**.

When you want to write to the console device, you use the same **IOStdReq** data structure information that was created by the call to **OpenDeviceQ**, 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. Alternatively, 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 does not 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 that allows the devices to behave like virtual terminals. When this feature is present, if a character to be written will not 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, such as **LINE FEED**, **CLEARJEND_OF_LINE**, and cursor movements. The complete list of control functions available from the console device is quite long; 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 keycodes 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, your program 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. 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 producing graphics images, lines, and text in displays. For quick and easy rendering, you can use Intuition's high-level data structures and functions. You are also free to use all of the lower-level Amiga graphics, animation, and text primitives.

This chapter shows you how to use the Intuition structures and functions, but the Amiga primitives are a large topic in themselves and the discussion here can only point the way. *You* will find instructions for using the primitives in the *Amiga ROM Kernel Manual*.

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 have learned one you have 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 reduces the number 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 drawn for you.
- o In windows or screens, you draw the illustration types directly into the display element by using one of the functions **DrawImageQ**, **DrawBorderQ**, or **PrintIntText()**.

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 drawn 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 drawing the illustration types directly into screens and windows, however, you use the Intuition functions **DrawBorderQ**, **DrawImage()**, and **PrintTextQj**. 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 drawing 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 is 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 is called a border because that is 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:
 - o A color for the lines.
 - o A color that can be used for background areas enclosed within lines,
 - o One of several drawing modes.
- o An optional pointer to another instance of **Border**.

Border Coordinates

Intuition draws lines between points that you specify 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 is another coordinate pair, the starting point of the next line.

Here is an example. 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.

The (**LeftEdge**, **TopEdge**) coordinate pair defines 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), the next point is at $(20+15, 15+0)$, or (35,15). A second line segment is drawn from (35,20) to (35,15).

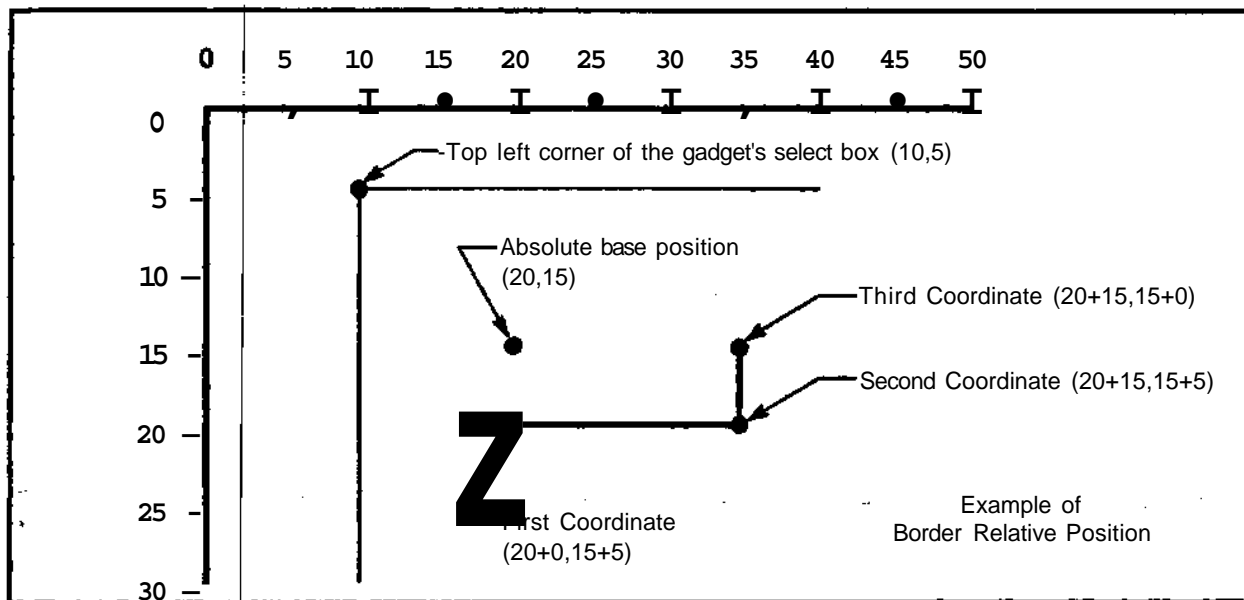


Figure 9-1: Example of Border Relative Position

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 five 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: **JAMI**, and **XOR**. To draw the line in your choice of color, use **JAMI**. 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, 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 all the 0 bits and 1 bits in the binary representation of the color register number. In a three-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-drawn 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

This field gives the 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

This field contains the number of pixels from the left edge of the containing element.

TopEdge

This field specifies the number of lines from the top line of the 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:

JAM!

This specification uses **FrontPen** to draw the line and makes no change in the background.

XOR

This specification changes the background beneath the line to its binary complement.

NextBorder

This field is a 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

This field is a pointer to an array of coordinate pairs, one pair for each line.

Count

This field specifies the number of pairs in the array of coordinate pairs; the 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 handy in creating menus.

To define and display **IntuiText**, you specify the following:

- o Colons for the text and, optionally, for the text's background.
- o One of three drawing modes.
- o The starting location for the text.

- o The default font or your own special font.
- o A 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 five bit-planes, 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, but not the character background area, will 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 the 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 drawn. **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 three-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 create several distinct groups of characters with one stroke; each group has 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 drawing 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

This field specifies 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

This field specifies the starting position for the text as an offset from the left corner of the containing element.

The field contains an integer or constant, which is the number of pixels from left edge of containing element.

TopEdge

This field specifies the starting position for the text as an offset from the top line of the display element.

The field contains an integer or constant, which is the number of lines from the top line of the containing element.

TextAttr

This field is a pointer to a **TextAttr** structure containing your own font description. Set this to NULL if you want the default font.

IText

This field is a pointer to null-terminated text.

NextText

This field is a pointer to another instance of **IntuiText**, if this text is part of a linked list of text strings.

Set this field to NULL if this text is not part of a list or if it 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 The location of the image within the display element.
- o The width and height of the image and the data to create it.
- o The depth of the image that is, how many bit-planes are used to define it.
- o The 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 Is and Os 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 six 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 how many bit-planes there are in the image data and 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 that 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:

- Wherever there is a 0 bit in the image data, the color in color register 0 is displayed.
- 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 1s and bit-plane 1 contains 0s and 1s, 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, figure 9-2 shows the layout for the system sizing gadget, which is a one-bit-plane image.

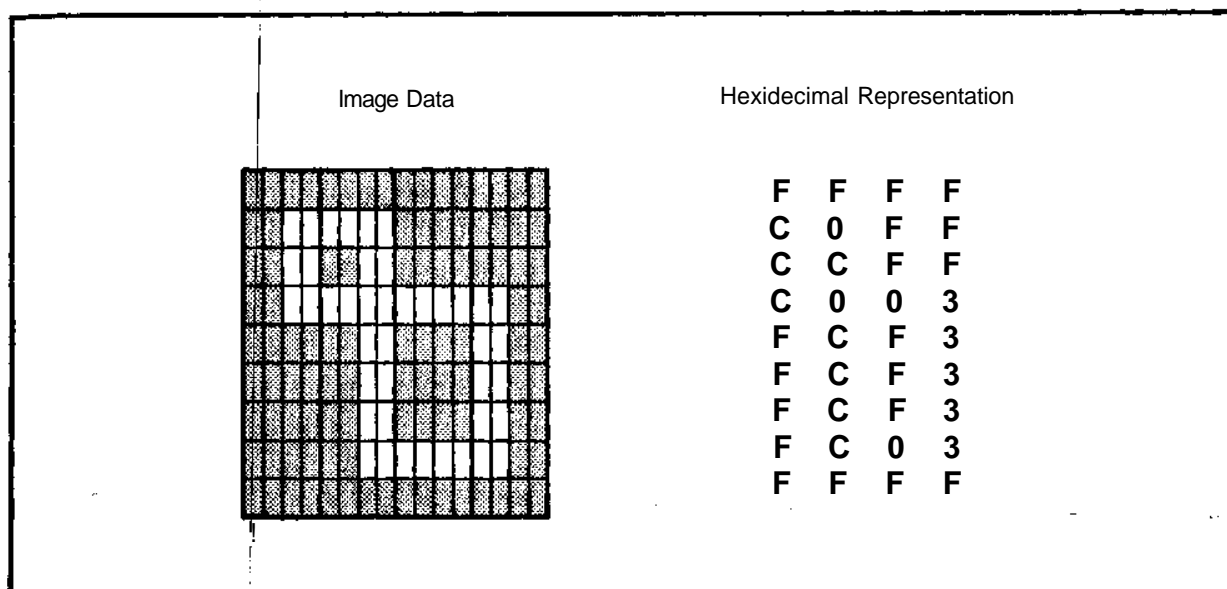


Figure 9-2: Intuition's High-resolution Sizing Gadget Image

In hex notation, the data words of the sizing gadget image are defined as follows:

```
USHOjit SizeDatafl =
{
  OXPJFFF,
  OxCJOFF,
  OxCJJCHF,
  0x<}003,
  0xF|CF3,
  0xF CF3,
  0xltCF3,
  0xFC03,
  Ox^FFF,
};
```

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, whereas 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

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 SizerImage =
{
    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 have designed it properly).
- 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 *j* 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 0s or 1s. 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 three 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 spring gadget shown above has only one bit-plane of data. To display this gadget in plane 0 of a four-bitplane 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.

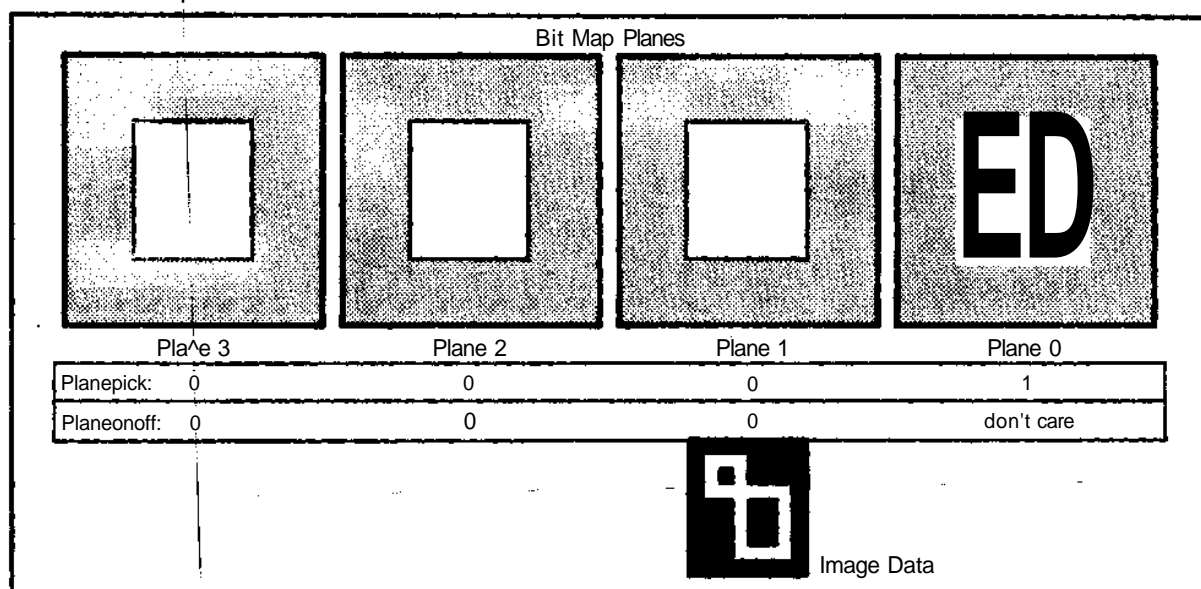


Figure 9-3: Example of PlanePick and PlaneOnOff

If you want the sizing gadget to be drawn in color 2 and its background drawn 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 1s. 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 need not 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;
    feHORT Width, Height, Depth;
    SHORT *ImageData;
    UBYTE PlanePick, PlaneOnOff;
    struct Image *NextImage;
};
```

The meanings of the fields in the **Image** structure are:

LeftEdge, TopEdge

These are offsets from the top left of the display element.

These fields contain integers or constants:

LeftEdge

This field contains the number of pixels from the left edge of the display element.

TopEdge

This field contains the number of lines from the top line of the display element.

Width

This field contains the width of the actual image in pixels.

The field contains an integer or constant.

Height, Depth

These fields specify the height of the image in pixels and the number of bit-planes needed to define the image.

These fields contain integers or constants.

ImageData*.

This field is a 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

A more complex example of an image is presented below. The image shown in figure 9-4 belongs to one of the system depth-arrangement gadgets (the front gadget, which brings a window or screen to the front of the display).

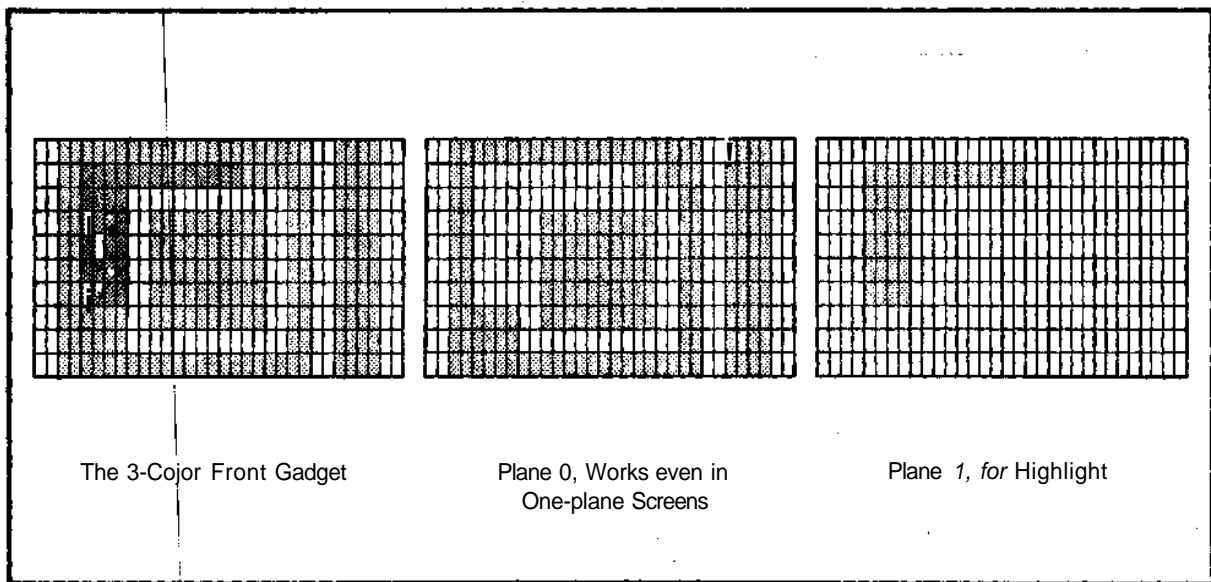


Figure 9-4: Example Image — the Front Gadget

Its data structure and data definition look like this:

```
USHORT UpFrontDatafl =
{
    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. 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 displayed. The resulting colors are 0 for the background and 1 and 2 for the imagery are

Image Memory

An extra requirement is imposed on image data (and on sprite data). It must be located in chip memory, which is memory that can be accessed by the special Amiga hardware chips. Chip memory is in the lower 512 Kbytes of RAM. In expanded machines (the Amiga can be expanded up to 8,000 Kbytes), the Amiga chips still cannot address memory locations greater than the 512-Kbyte limit. In hexadecimal notation, 512 K spans memory addresses \$00000 to \$7FFFF.

To write a program that will survive in any possible configuration of Amiga hardware, you are obliged to ensure that your image and sprite data resides in this chip memory. You can make sure that your data is in chip memory by using the ATOM tool on the file containing the data. The loader will then automatically load that portion of your program into chip memory. See the *AmigaDOS User's Manual* for information about ATOM and the loader. TM~

As of the time of this writing, the only way to check whether your data is in chip memory is by comparing its load address after it has been loaded into Amiga memory. If the address of the end of your data is less than \$80000, you are safe. If the address is equal to or greater than \$80000, you must allocate chip memory and copy your data into the new location. To allocate chip memory, call the Exec function AllocMem() with MEMF_CHIP as the requirements argument.

FOR GRAPHICS FUNCTIONS

Following are brief descriptions of the Intuition functions that relate to the use of illustration data types and the Amiga graphics primitives.

Drawing Images, Lines, or Text in a Window or Screen

DrawImage (RPort, Image, LeftOffset, TopOffset)

This function moves the **Image** data into the **RastPort** window or screen.

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)

This function 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 coordinates.

TopOffset = offset added to each vector's y coordinates.

PrintText (RPort, IText, LeftOffset, TopOffset)

This function 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 coordinates.

TopOffset = offset added to **IntuiText** y coordinate.

Obtaining the Width of a Text String

IntuiTextLength (IText)

This function returns the width of an **IntuiText** in pixels. **IText** is a pointer to an instance of an **IntuiText** structure.

Obtaining the Address of a View or ViewPort

ViewAddressQ

I This function returns the address of the Intuition **View** structure for any
I graphics, text, or animation primitive that requires a pointer to a **View**.

ViewPortAddress (window)

I This function returns the address of the screen **ViewPort** associated with
I the specified window for any graphics, text, or animation primitive that
I 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 drawing into windows and custom screens with the general-purpose Amiga graphics routines and for drawing borders, images, and text into windows and screens with the Intuition routines. This section also has some cautionary advice about using drawing 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—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 those below:

o Pointers to window **RastPort** and **ViewPort**:

```
struct Window *MyWindow;  
struct RastPort *MyRPort;  
struct ViewPort *MyVPort;  
struct View *BigView;
```

```
My Window = OpenWindow(...);  
MyRPort = My Window->RPort;  
MyVPort = ViewPortAddress(MyWindow);  
BigView = ViewAddressQ;
```

o Pointers to screen **RastPort** and ViewPort:

```
struct Screen *My Screen;  
struct RastPort *MyRPort;  
struct ViewPort *MyVPort;  
struct View *BigView;
```

```
MyScreen = OpenScreen(...);  
MyRPort = &My Screen-> RastPort;  
MyVPort = &MyScreen-> ViewPort;  
BigView = ViewAddress();
```

The Intuition function `ViewPortAddressQ` 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 `ViewAddressQ` returns the address of the Intuition View structure.

When you use graphics primitives to draw 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 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 0) 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 a button	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 a button	Positioning the pointer and quickly pressing and releasing one of the mouse buttons.
Double-clicking a button	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 is impossible to imagine all the uses you will 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. Alternatively, 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 through which a complex exchange of information can take place. Because the requester is used for the transfer of information, it is 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 is 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. To move an object on the Workbench screen, the user places the pointer within the object's icon, 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 in **IntuiMessages** as the event class **MOUSEBUTTONS**, with the codes **SELECTDOWN**, **SELECTUP**, **MENUDGWN**, 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 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 interlaced screen.

To get mouse movement reported as deltas (amount of change from the last position) instead of as 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
<i>control</i>	CTRL	The associated Qualifier flag is the CONTROL flag.
<i>alternate</i>	ALT	Please note that there are two separate ALT keys, one on each side of the space bar. These can be treated distinctly. Your program can detect which one was pressed by examining the LALT and RALT commands for the Left ALT and Right ALT keys respectively
<i>escape</i>	ESC	When this key is struck, its keycode is entered into the input stream as an actual keystroke.
<i>function</i>	F1 to F10	Shortcut methods for entering command-key sequences starting with the ESC key.
<i>AMIGA</i>	Fancy A	There are two Amiga keys, one on each side of the space bar. These, like the ALT keys, are distinctly identifiable. The Left AMIGA key is recognized by the Qualifier Sag 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 section.

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 selections. This gives the user a shortcut method to select often-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.

It is recommended that you abide by certain command-key standards to provide a consistent interface for Amiga users. Chapter 12, "Style," contains a complete list of the recommended standards.

Chapter 11

OTHER FEATURES

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 `FreeRememberQ`. They use a data type called **Remember**.

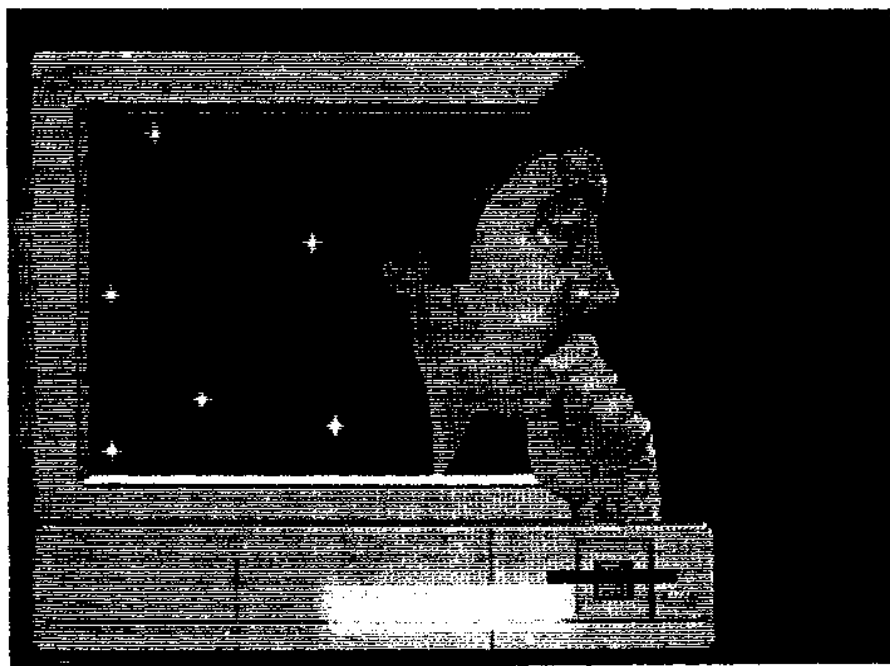


Figure 11-1: Intuition Remembering

INTUITION HELPS YOU REMEMBER

The **AllocRememberQ** routine calls the Exec **AllocMemQ** function to do your memory allocation for you. It 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 **FreeRememberQ** 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 in one of two ways: You can free both the memory blocks you've allocated and the link nodes that Intuition allocates, or, after you have 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 **AllocRememberQ**. 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 **FreeRememberQ** does the job (see figure 11-1).

- The other use is to do a series of memory allocations and abandon it in midstream easily, if you must. 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, such as "out of memory." When aborting, you should free up any memory that you have 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 have 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 is a pointer to the data structure **Remember** and initializing that pointer to NULL. This variable 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 is linked into the master list to which your **RememberKey** points. Then, to free up any memory that has 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

The **Remember** structure is as follows:

```
struct Remember
{
    struct Remember *NextRemember;
    ULONG RememberSize;
    UBYTE *Memory;
};
```

The **Remember** variables are explained below.

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 Interface (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 your program can call `GetPrefsQ` to find out what settings the user has made. In a system in which the user does not use Preferences, you can call `GetDefPrefsQ` 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, your program will receive an `IntuiMessage` telling it that there is a new set of Preferences for it to examine. To get the new settings, the program then calls `GetPrefsQ`.

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 **GetDefPrefsQ** 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 start-up 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.

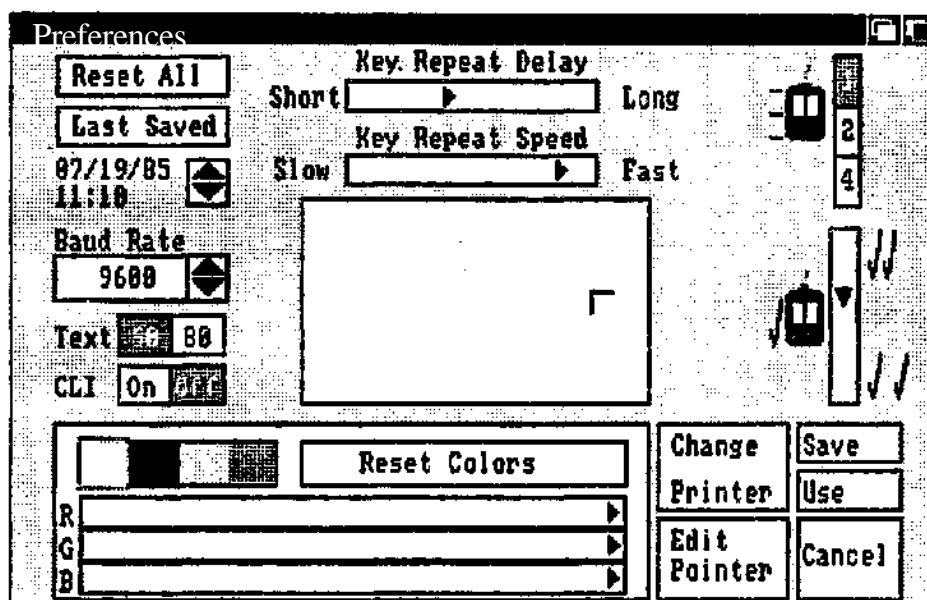


Figure 11-2: Preferences Display

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 do not 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 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 Interface 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, determines 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:

FontHeighi

This variable will contain one of **two** constants: TOPAZJSIXTY or TOPAZ JEIGHTY. 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 the time between repeats of the key.

DoubleClick

This is a timeval structure that 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.

PointerMatrixpOINTERSIZE]

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 advised. For instance, if **PointerTicks** is set to 32768, 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.

colorO, color1, color2, color3

These are the Workbench colors.

ViewXOffset, ViewYOffset

These describe the offset of the **View** from its initial start-up 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

Your program 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(PrefBuSfer, *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—**RethinkDisplayQ** and **RemakeDisplayQ**. The **MakeScreen()** function works only with the Copper lists of your custom screen.

RetHnkDisplay() reworks Intuition's internal state data, rethinks the relationship of all of the screen ViewPorts to one another and reconstructs the entire Intuition display by calling the graphics primitives **MrgCopQ** and **LoadViewQ**. 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 the *Amiga Hardware Reference Manual*.

RethinkDisplayQ makes calls to the graphics primitives **MrgCopQ** 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). **LoadViewQ** uses this merged Copper instruction list to create the display. Before

calling `RethinkDisplayQ`, you may wish to call `MakeScreenQ` to create the Copper instruction list for your own custom screens.

Note that `RethinkDisplayQ` can take several milliseconds to run, and it locks out all other tasks while it runs. This can seriously degrade the performance of the multitasking Executive, so do not use this routine lightly.

The function `RemakeDisplay` reconstructs the entire Intuition display. It calls `MakeScreenQ` for every screen in the system and then calls `RethinkDisplayQ`. As with `RethinkDisplayQ`, `RemakeDisplayQ` can take several milliseconds to run, and it locks out all other tasks while it runs. This can seriously degrade the performance of the multitasking Executive, so do not use this routine lightly.

To remake the Copper lists of your custom screen, call `MakeScreenQ`. The only difference between `MakeScreenQ` and the graphics library routine `MakeVPortQ` is that Intuition synchronizes your call to `MakeVPortQ` with any calls that it needs to make.

Current Time Values

The function `CurrentTimeQ` 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, the screen-flashing 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 independently of the Intuition display. In particular:

- o Sprites cannot 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 **DrawGListQ** or **MoveSpriteQ**).

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). Thus, you can look at any routine, start from register AO if the routine's arguments start with an address, and start from DO when the routine's arguments become data values. As an added mnemonic, even the register names are in alphabetical order—AO precedes DO. The register names for each function are given in appendix A.

Unfortunately for assembly programmers, many of you will have to use assemblers that do not give you macros to declare and reference structure elements. If this is the case, you should use the include file called *intuition.i*, in which 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 OSMenuQ when an item becomes meaningless or non-functional. Do not 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 allow the user to select which project to work with, you should create a "Project" menu. For consistency, it is suggested that everyone create their menu strips with the Project menu as the leftmost menu. This menu 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, it is suggested 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 is not obvious to users which gadget they are selecting. Unless you are very careful, all sorts of weird things can happen and the gadgets will behave in unusual ways.

As with menus, use OffGadgetQ to remove a gadget when it becomes meaningless or nonfunctional.

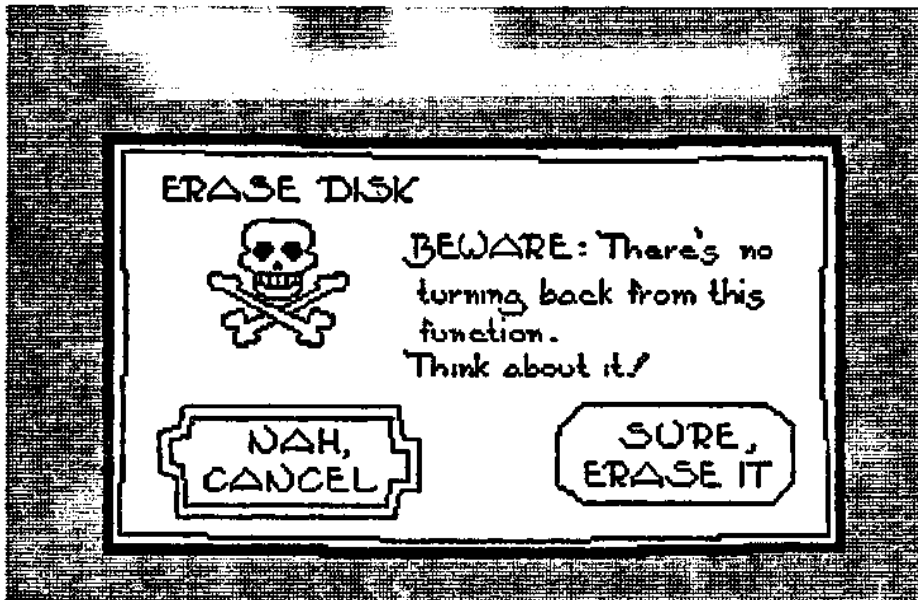


Figure 12-1: The Dreaded Erase-Disk Requester

Requester Style

This is easily the *most important* rule about requesters: always make sure that there is a safe way to exit from any requester. In figure II < requester can be cancelled; 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 extremely important. We cannot emphasize this point strongly enough.

When you design a requester with your own Map 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 enable a shortcut, users should be able to hold down the AMIGA key with the little finger on one hand, and press one of the keys they would normally press with the other hand. This will help touch typists as well as prevent that clumsy feeling that everyone experiences.

Table 12-3 shows recommendations for standard selection shortcuts (using the left AMIGA key to emulate usage of the left button of a mouse):

Table 12-3: Selection Shortcuts

Key Pressed with Left AMIGA Key	Function
A	"Select a small piece to the right of the cursor," such as the next word
O	"Select a bigger piece to the right of the cursor," such as the next sentence
p	"Select an even bigger piece to the right of the cursor," such as the next paragraph
J	"Select a small piece to the left of the cursor," such as the previous word
K	"Select a bigger piece to the left of the cursor," such as the previous sentence
L	"Select an even bigger piece to the left of the cursor," such as 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 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

Key Pressed with Right AMIGA Key	Function v
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

Mouse Style

Intuition uses the left mouse button for *selection* and the right mouse button for *information transfer*. To help the user understand and remember the elements of every application, you are encouraged to follow this model.

When the user presses the left button, Intuition examines the state of the system and the position of the pointer and uses this information to decide if 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. 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.

Most often, Intuition uses the right button 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. Because this requester is used for the transfer of information, it is appropriate to use the right mouse button.

If you are planning to handle mouse button events directly, you should follow the selection/information model described above.

The Sides of Good and Bad

Whenever the user is presented with a pair of choices that could be characterized as positive/negative options, the positive option should always appear on the left and the negative option on the right. For example, if you are designing a requester with "OK!" and "CANCEL" options, "OK!" should appear on the left and "CANCEL" on the right. If your options are "RETRY" and "ABORT," you should render "RETRY" on the left and "ABORT" on the right.

Intuition's `AutoRequestQ` requester and `DisplayAlertQ` alert both use this scheme. If all programs mimic this design, the user will come to feel secure in knowing that the right button can always be used to abort some dire sequence of events while the left button selects the normal, placid continuation of events. Refer to figures 1-4, 7-2, and 12-1 for examples.

Miscellaneous Style Notes

Remember, exiting programs should always make a call to `OpenWorkBenchQ`, even if you did not call `CloseWorkBenchQ`. Workbench should be open as much as possible. If Workbench was closed and your departure has freed up enough memory for Workbench to reopen, it is preferable that it be reopened. `OpenWorkBench()` will not necessarily work (if there is no memory for the display, it will not open). But if every program calls `OpenWorkBenchQ`, 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 have 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. Do not 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 SetRGB4() function on the Viewport of any screen. An example of this follows:

```
struct Screen *MyScreen;

SetRGB4(&MyScreen->ViewPort, 17, Red17, Green17, Blue17);
SetRGB4(&MyScreen->ViewPort, 18, Red18, Green18, Blue18);
SetRGB4(&MyScreen->ViewPort, 19, Red19, Green19, Blue19);
```

A Final Note on Style

Design beautiful Gadgets, Menus, Requesters. Think simplicity and elegance. Always remember the fourth grader, the sophisticated user, and the poor soul who is terrified of breaking the machine.

Dare to be gorgeous and unique. But don't ever be cryptic or otherwise unfathomable. Make it unforgettably great.