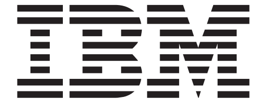
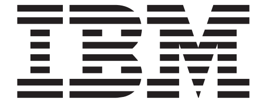
AIX 5L Version 5.2



AIXwindows Programming Guide

SC23-4870-02

AIX 5L Version 5.2



AIXwindows Programming Guide

SC23-4870-02

Note

Before using this information and the product it supports, read the information in Appendix F, “Notices,” on page 117.

Third Edition (April 2001)

This edition applies to AIX 5.1 and to the AIXwindows licensed program and to all subsequent releases of this product until otherwise indicated in new editions.

A reader’s comment form is provided at the back of this publication. If the form has been removed, address comments to Information Development, Department H6DS-905-6C006, 11501 Burnet Road, Austin, Texas 78758-3493. To send comments electronically, use this commercial Internet address: aix6kpub@austin.ibm.com. Any information that you supply may be used without incurring any obligation to you.

© Copyright International Business Machines Corporation 1997, 2002. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About This Book . . . v
Who Should Use This Book . . . v
Highlighting . . . v
Case-Sensitivity in AIX . . . v
ISO 9000 . . . v
Related Publications . . . v

Chapter 1. AIXwindows Overview for Programmers . . . 1

Chapter 2. AIXwindows Window Manager Overview . . . 3
Starting and Exiting X and the AIXwindows Window Manager . . . 3
Restoring Default Behavior . . . 3

Chapter 3. Using the AIXwindows Customizing Tool . . . 5
Using the Customizing Tool Introduction . . . 5
Related Information . . . 5
How to Start the Customizing Tool . . . 5
Using the Customizing Tool Main Window . . . 6
Using the Browsers . . . 10
Understanding the app-custom Files . . . 14

Chapter 4. AIXwindows Xlib Library . . . 19
Xlib Introduction . . . 19
Suggested Reading . . . 21
Using Display Functions in AIXwindows . . . 21
Graphical Overlay Planes . . . 23
AIXwindows National Language Support (NLS) . . . 25

Chapter 5. Using Extensions in AIXwindows . . . 27
Extension Functions . . . 27
Dynamically Loadable X Server Extensions . . . 35
AIXwindows Nonrectangular Window Shape Extension . . . 35
AIXwindows Screen Saver Extension . . . 38
Using AIXwindows Input Device Functions . . . 46
AIXwindows Input Extension Library . . . 51
AIXwindows Input Extension Protocol Specification . . . 63
AIXwindows Double Buffer Extension Specification . . . 69
AIXwindows XTest Extension . . . 79

Chapter 6. AIXwindows Font Enhancements . . . 83
ISO9241 Compliant Bitmap Fonts . . . 83
TrueType Rasterizer . . . 83

Appendix A. Bidirectional Support in Xm (Motif 1.2) Library . . . 85
Library Name . . . 85
Purpose . . . 85
Description . . . 85
Bidirectional Resources . . . 85
The Text Widget . . . 87
The Text Field Widget . . . 87
The Label and Gadget Widget . . . 87
The List Widget . . . 87
Effect of Layout Direction on Motif Widgets and Gadgets . . . 87

© Copyright IBM Corp. 1997, 2002

Example of Localizing a Motif Application for Bidirectional Support 88

Appendix B. Font Utility 91
Font Utility Introduction 91
Font Utility Limitations 92
How to Use the Font Utility 92
Using the Font Utility Window 94
Using the Reference Font Window 98
Using the Raster Editor Window 98
Using the Vector Editor Window. 102

Appendix C. Display Power Management 107

Appendix D. Setting Up X11 Graphic Input Devices (LPFKeys, Dials, Tablet, Spaceball) 109
Tablet 109
Dials and LPFKeys 109
Spaceball 110

Appendix E. The X Virtual Frame Buffer 111
Overview 111
Installing the XVFB 111
Starting the XVFB 112
Testing the XVFB 113
Implementing XVFB in Application Code. 113
Working with the XVFB 114
DirectSoft OpenGL and the XVFB 115
CATweb and the XVFB 116

Appendix F. Notices 117
Trademarks 118

Index 119

About This Book

This book contains information about user interfaces from a programming perspective.

Who Should Use This Book

This book is intended for users with programming experience who are interested in customizing their personal AIXwindows environment as well as for experienced C programmers who are designing user interfaces for user communities.

Highlighting

The following highlighting conventions are used in this book:

Bold
: Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.

Italics
: Identifies parameters whose actual names or values are to be supplied by the user.

`Monospace`
: Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

Case-Sensitivity in AIX

Everything in the AIX operating system is case-sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the **ls** command to list files. If you type `LS`, the system responds that the command is ″not found.″ Likewise, **FILEA**, **FiLea**, and **filea** are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always ensure that you use the correct case.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

Related Publications

The following book contains information about or related to programming the user interface:

- *The graPHIGS Programming Interface: Understanding Concepts*

© Copyright IBM Corp. 1997, 2002

Chapter 1. AIXwindows Overview for Programmers

This book describes AIXwindows, IBM's enhancements to X-Windows and Motif.

AIXwindows is a state-of-the-art graphical user interface environment that can be used by a wide range of end users and application developers to create graphical user interfaces, either on your system or across a network. The environment provides a graphical desktop that hides the low-level complexities of the operating system. The environment also supports Base 2-D as well as 3-D products.

The Base 2-D products are X-Windows and Motif. The application programming interfaces (APIs) that support 3-D are Graphics Library, and graPHIGS API. In addition, the AIX Common Desktop Environment allows you to organize your online work much as you would organize work on your office desk.

This overview summarizes the features of the AIXwindows environment comprised of:

- X-Windows (previously called Enhanced X-Windows), a network-transparent windowing system for creating and managing windows on bitmapped display screens. These windows enable you to show multiple work items simultaneously, much like working at an office desk. Because of its portability and network transparency, applications that run on AIXwindows look the same to the user and do not need to be rewritten by the programmer
- Motif Environment (previously called AIXwindows), a set of guidelines and tools that specify how a user interface for graphical computers should look and feel. These specifications focus on the design of the objects that make up the user interface: the menus, buttons, dialog boxes, text entry, and display areas.

X-Windows consists of three components: the server, client programs, and the communications channel as shown in the following figure.

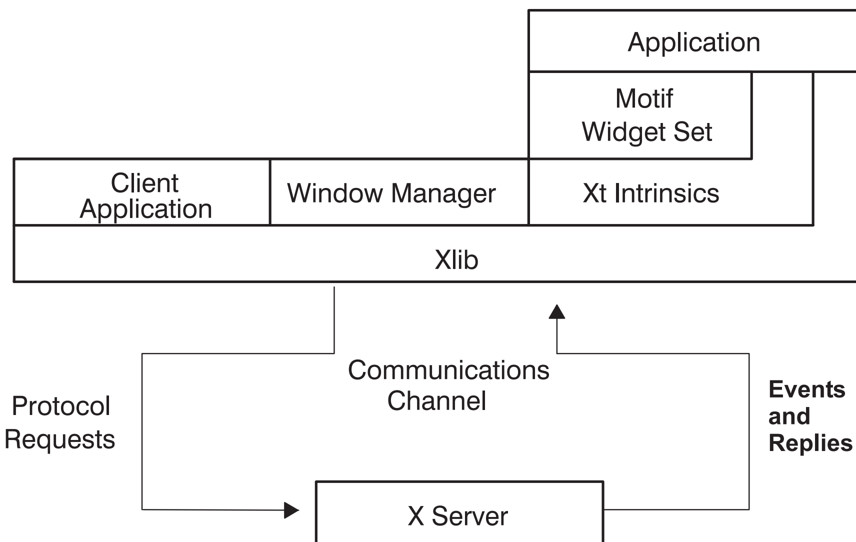


Figure 1. X-Windows Components

The server distributes user input to and accepts output requests from various client programs (applications) located either on the same machine or elsewhere on the network. These applications use the low-level C-language **Xlib** library to interface with the window system through the communications channel. Although a client usually runs on the same machine as the X server to which it is talking, this need not be the case.

© Copyright IBM Corp. 1997, 2002

The **Xlib** library is a layer on the X server and is a set of functions that are called by applications. These functions handle tasks such as adding a host, opening a display, manipulating windows, drawing, and handling events. **Xlib** calls are translated to protocol requests that are passed either to the local server or to another server across the network.

Applications are a layer on **Xlib**. Applications that run on **Xlib** can be used both by end-users and application developers. These applications can be the user's own or those provided with X-Windows. Beginning-to-experienced end-users can use the provided client programs to run multiple terminals, use desk accessories, set display and keyboard preferences, use font and graphic utilities, print applications, and get information on windows and displays. Application developers can use the Motif Toolkit provided to create and control user interfaces.

A Motif application consists of the X Intrinsics library and the Motif widget set. The X Intrinsics are a basic set of functions used to define, create, manage, and destroy user interface components. These user interface components are called widgets and make up the widget set. The Motif widget set is a layer on the X Intrinsics. Different interfaces can be provided by different widget sets, but they all use the X Intrinsics.

While the Motif Toolkit provides the building blocks for an interface, the window manager enables you to control the interface. The window manager acts as an application program. With the window manager, you can execute other application programs, move them on the screen, resize them, and so on.

X-Windows has standards and conventions that are applicable to all systems with bit-mapped display terminals. This enables application programmers to spend more time improving their programs and less time porting to new user-interface platforms. Standards and conventions are provided for the X protocol, atoms, the Inter-Client Communications Conventions Manual (ICCCM), display manager control protocol (XDMCP), logical font descriptions (XLFD), and compound text encoding.

Graphics Library is a set of graphics and utility functions that provide high-level and low-level support for graphics. Graphics Library has become the industry standard for 3-D application development.

The graPHIGS API application programming interface (API), an implementation of the PHIGS and proposed PHIGS PLUS standards, provides a set of device-independent programming tools that allow portability across many hardware platforms. The graPHIGS API offers over 500 high-level subroutines to define, modify, and display hierarchically organized graphics data. Advanced rendering capabilities, such as lighting and hidden line and hidden surface removal, are included.

The AIX Common Desktop Environment provides you with the ability to run several applications at once on your screen. Whether you are a beginner or an experienced user, the AIXwindows visual system helps you manage your work and use the operating system to perform various tasks.

The product you choose to use will depend on your needs. If you are sending 2-D graphics across the network, you should use X-Windows. However, if you are sending 3-D graphics across a network, you must use the distributed application processing capability of the graPHIGS API.

If you are working on one system and do not need to send information across a network, the fastest method would be to use the Graphics Library or graPHIGS API. Using these products would bypass the server and allow direct adapter access. Although Graphics Library applications typically operate in immediate mode and applications using the graPHIGS API have traditionally used structure storage, both products offer retained and nonretained graphics display as well as advanced rendering capabilities.

Chapter 2. AIXwindows Window Manager Overview

Starting and Exiting X and the AIXwindows Window Manager

Because different computer systems have different ways of starting X and AIXwindows, you should consult with your system administrator to learn how to get started. Usually, X and MWM are started from a shell script that runs automatically when you log in. You may, however, find that you need to start X or MWM or both.

If you log in and find that your display is functioning as a single terminal, with no windows displayed, you can start X by issuing the following command:

```
% xinit
```

If this command does not start X, check with your system administrator to ensure that the X11 directory containing executable programs is in your search path. The appropriate path may differ from one system to another.

If you log in and find one or more windows without frames, you can start MWM with the following command:

```
% mwm &
```

Before entering this command, make sure that the pointer rests within a window that has a system prompt.

Press the Ctrl-Alt-Backspace key sequence to exit AIXwindows.

Note: Before exiting AIXwindows, the recommended procedure is to exit any application programs and stop any commands that may be running in terminal windows. This avoids the possible loss of data due to improperly stopping a program.

When you exit a program, the command-line prompt returns to the terminal window. However, if you started the program automatically or from a menu, exiting also removes the terminal window. Selecting the **Close** option from the window menu immediately stops any program running in the window. Interrupting a program like this may cause it to lose data. However, you can close the clock, the load histogram, or an ″idle″ terminal window (one showing a command-line prompt) with no ill effect.

Restoring Default Behavior

Because AIXwindows permits a great deal of customization both by programmers writing AIXwindows applications and by users, you may find that mouse buttons or other functions do not operate as you might expect from reading this documentation. You can reset your AIXwindows environment to the default behavior by pressing the following four keys simultaneously:

Ctrl-Alt-Shift-!

You can return to the customized behavior by pressing this key sequence again. If your system does not permit this combination of keystrokes, you can also restore default behavior from the default root menu.

© Copyright IBM Corp. 1997, 2002

Chapter 3. Using the AIXwindows Customizing Tool

The AIXwindows customizing tool helps to change the look of a client application. It provides a simplified method for you to set the values of resources. *Resources* are items such as colors, fonts, and other attributes which can be customized.

For more information on the customizing tool, see the following:

- Using the Customizing Tool Introduction
- How to Start the Customizing Tool
- Using the Customizing Tool Main Window
- Using the Browsers
- Understanding the app-custom Files

Using the Customizing Tool Introduction

The customizing tool helps to change the look of an application. It provides a user-friendly way to set the values of resources. Resources allow you to customize a client application. Each application has its own unique set of resources. Files called **app-custom** files have been created for a set of popular applications. These files contain information in a special format that describes the resources available for the modification of an application and, in some cases, the possible resource values you can select.

The customizing tool has browsers to help you choose valid values for the resources. It can often apply these values to the application so you can see your changes immediately instead of having to restart your application. Once you like the look of your application, you can save your changes permanently.

Related Information

AIXwindows Overview provides general information on the environment.

Using the AIXwindows Window Manager allows you to move windows, change their size, change them into icons, and create new windows.

Bidirectional Support in Xm (Motif) Library provides a convenient way of creating Arabic/Hebrew graphical user interfaces.

How to Start the Customizing Tool

Prerequisite Condition

The customizing tool must be installed before you can start it. The **custom** command is installed in the **/usr/bin/X11** directory.

Basic Start Procedure

To start the customizing tool with no options, enter the following:

```
custom
```

No command-line options are required. If no application name is specified at the command line, you are prompted for it. Once the customizing tool start-up window is displayed, you may use one of the following three methods to choose an application:

- Click on the application with the mouse button.
- Select the application from the list of applications.
- Type the application name into the text field.

© Copyright IBM Corp. 1997, 2002

After you have chosen an application, the customizing tool main window is displayed with the resources that are valid for that application.

Detailed Start Procedure

To start the customizing tool using a command-line option, use the following syntax:

```
custom [-h | -e Browser | [-s ResourceFile] [Application]]
```

Command line help can be obtained using the **-h** option.

The **-e** option allows one of the standalone browsers to be called. Valid values for the *Browser* parameter are **color**, **font**, **cursor**, and **picture**.

Use the **-s** option to specify the resource file from which to load and save resource settings. If the **-s** option is not specified, the default is to load the values from the resource database stored in the **RESOURCE_MANAGER** property on the X server. If this database does not exist, then **$HOME/.Xdefaults** is loaded. See the **custom** command in *AIX 5L Version 5.2 Commands Reference* for more information.

Most standard X Toolkit command-line options are understood by the **custom** command. See the table in the **custom** command that lists the standard command-line options.

An application name can be specified on the command line. If no application name is specified on the command line, you will be prompted for it.

When more than one instance of an application is running, all are updated immediately when a resource value is changed. The same behavior occurs if you save these changes and restart your application.

The customizing tool searches for an **app-custom** file that describes the resources that can be set for the application you chose. If there is no **app-custom** file for the application you are customizing, the **DEFAULT app-custom** file is used. The **DEFAULT app-custom** file provides a set of resources that apply to most applications. An error message is displayed if either the **DEFAULT app-custom** file cannot be found or if there is an error in the **app-custom** file itself.

Related Information

AIXwindowsUsing the Customizing Tool Overview
Using the Customizing Tool Main Window
Using the Browsers
Understanding the app-custom Files

Using the Customizing Tool Main Window

This customizing tool has a graphical user interface that describes the resources available for modification of a particular application and their possible values. Tools are provided to assist in setting these values. When you complete modifications, you can save the changes, restore the defaults, start a new customizing session, or exit the customizing tool.

For more information on the customizing tool main window, see the following:

- Application Class
- Resource Category Button
- Scrolled Window Area
- File Pulldown Menu
- View Pulldown Menu
- Options Pulldown Menu

- Help Pulldown Menu

Application Class

The class of the application that you are customizing (for example, XCalc) is displayed in the area on the right beneath the ″Customizing″ label. The class of the application is hard-coded into the application itself and is generally different from the command that starts the application. By convention, the class usually begins with a capital letter.

Resource Category Button

Pressing and holding the left mouse button on the **Resource Category** button displays a list of resource categories. You can select a category by dragging the mouse over the list and releasing the mouse button on your choice. The panel inside the scrolled window area changes to present another set of resources to customize.

Scrolled Window Area

Inside the scrolled window are the resources that can be customized. Each line contains a description of the resource, a field to enter a value for the resource, and, in some cases, a button that starts a tool called a browser to assist in choosing a valid value.

On the left side of the scrolled window area are descriptive explanations of each resource you can customize. To see the actual resource strings, select the **Resources** option from the **View** pulldown menu.

To the right of the resource descriptions are fields for entering values for particular resources. To type in this field, press the left mouse button on this field so the keyboard focus is directed there; then enter the desired text.

You can directly type a value into the text field or, if there is a button to the right, you can press it to access a browser that assists you in choosing a valid value. The button is highlighted when the browser is called and not highlighted when the browser is exited.

When changing a resource involving a cursor you must use the cursor browser to choose a value for a cursor because you cannot type directly into the **Chosen Value** field for cursors.

When the `<-Number` label appears to the right of the text field, only numbers may be entered as valid values.

To select a value when the `<-Choice` label is present, press the button to the left and select a value from the displayed list. The first item in this menu is always `<default>`, which allows the default for that resource to be used and specifies that no value is written to your resource file when you save.

File Pulldown Menu

The **File** item on the menu bar has four options:

Open Starts the customizing session for a new application. You are presented with the Starting the Customizing Tool dialog. You can choose an application and press the **OK** button. If you decide not to start a new session, you can press the **Cancel** button.

Save Saves your changes to the current resource database. This is usually the same place from which the resources were initially loaded from showing the current values. The current resource database can be changed by using the **Save As . . .** option, but most users do not need to change it.

The default resource database initially used for loading and saving resources is the first of the following:

1. The file specified by the **-s** *ResourceFile* command-line option, if any.

2. The file specified by the **Custom*resourceFile** resource, if any.
3. The X server resource database, if it exists.
4. Otherwise, the **$HOME/.Xdefaults** file.

The X server resource database is usually created by the **xrdb** command, and it is stored in the RESOURCE_MANAGER property of the root window on screen zero.

When modifying an existing resource database, only the resource specifications that exactly match those defined in the **app-custom** file for the application are replaced or deleted. Comments and other settings are left intact.

Save As...

Allows you to save your changes to a location that you specify.

Once the **Save As . . .** option has been used, the **Save** option will continue to save to the new location. For most users, there is no need to specify a different resource database and using the **Save** option is sufficient.

You can specify that your settings be saved either to the resource database stored on the X server or to a file. The X server resource database is usually created by the **xrdb** command, and it is stored in the **RESOURCE_MANAGER** property of the root window on screen zero.

Reset Values

Erases all values you have chosen and reverts to values that were read in from the resource database at the beginning of the customizing session. You can cancel this operation. If you already saved your previous values, you must save again after resetting to erase those values and to accept the reset operation values.

Exit Exits the customizing tool editor. If you have made modifications but have not saved them, you are prompted to save and allowed to cancel the exit or exit.

View Pulldown Menu

The **View** item on the menu bar has two options: **Resources** and **Immediate Changes**. If you select the **Resources** option, the attribute labels change to display the actual resource names. To return to the descriptive strings, select the **Resources** option again.

The **Immediate Changes** toggle button controls the immediate updating of application resources. Previously, running applications had to be restarted for resource value changes to be applied. Using the customizing tool, many resources can be updated immediately when a value is changed. If you type a value directly in the text field of the resource, you must press the Enter key to update the client application.

The **Immediate Changes** button is desensitized (grayed out) until the customizing tool can communicate with the application. When communication is established you can use the **Immediate Changes** button as a toggle button to turn the immediate updating on or off. The default state for this button is on. If all instances of the application are exited anytime after startup, the button is desensitized again.

The **Immediate Changes** option is intended as a tool to view possible changes. Since it is possible to corrupt the interface of an application (for example, setting an unreadable font), it is best to use this option only when the application is in a noncritical state. There are some limitations, such as the following, to the immediate updating of applications:

- The **Immediate Changes** button is insensitive until communication with the application is established. The amount of time to wait for the application to contact the customizing tool is controlled by the **Custom*timeout** resource.
- Some undesired behavior may occur if you try to customize an application while it is starting. It is best to wait for your application to be completely displayed before attempting to customize it.

- To see immediate updating, the client that you are customizing must be run on a machine where the customizing tool has been installed. This is necessary because the shared **libXt.a** library contains a customizing tool enhancement.
- Some applications use low-level **libX11.a** calls that cannot be immediately updated by the customizing tool (for example, the **aixterm** and **mwm** commands).
- Also, some applications may hard code values so they can be updated immediately but do not take effect when the application is restarted. You can also bypass protection that the application writer intended, unintentionally corrupting the interface.
- An object on the interface might not resize correctly after immediate changes are applied. This is especially true for fonts.
- The customizing tool prefaces the resource with the application class. The resource value may not take effect because a more specific resource may be set in the customizing tool, in your **.Xdefaults** file, in the X server resource database or in the application **app-defaults** file. If you remove these resources or make them less specific, the customizing tool changes can then apply.

The **Immediate Changes** button becomes insensitive if the customizing tool cannot update the application. If you cannot update an application immediately, you can always save the changes to your resource file and restart your application to see the changes take effect.

Options Pulldown Menu

The **Options** item on the menu bar has the following option:

Reacquire Immediate Changes
The **Reacquire Immediate Changes** option allows you to regain communication with an application for the purpose of updating it immediately. You may wish to do this if the application was not already running when you started the customizing tool or if you restarted the application.

Help Pulldown Menu

The **Help** item on the menu bar has the following three options.

Help Facility

Brings up the help facility at the introduction level.

The help facility contains a Help Message window for viewing the help messages and an index to navigate through the facility. The index is hierarchical. Selecting an item in the index causes the corresponding help message to be displayed in the Help window. If the index item has additional help messages, selecting that item causes the index to descend to the next level. To return to the preceding level, press the **Go Back** button.

The **On Context** button provides access to context-sensitive help. See the **On Context** description that follows.

The help facility can remain active throughout your customizing session. To exit the help facility, press the **Cancel** button.

On Context
Changes the mouse cursor to the question mark cursor, prompting you to click on any object of interest. The index and help messages for that object are displayed in the help facility.

On Help
Brings up the help facility and displays the help message that describes how to use the facility.

Related Information

Using the AIXwindows Customizing Tool Overview
Using the Browsers
Understanding the app-custom Files
How to Start the Customizing Tool

Using the Browsers

The customizing tool provides useful browsers that assist you in changing the values of resources. The most useful of these browsers can also be started as standalone applications. This can be useful, for instance, if you want to locate a particular color but do not want to actually customize an application.

The following can be started as standalone browsers:

- Colors browser
- Fonts browser
- Pictures browser
- Cursors browser

The remaining tools are not available standalone:

- Choices browser
- Filenames browser

Colors Browser

The **Colors** browser assists you in choosing valid color values. The list of colors in the Colors browser allows color names from the **/usr/lib/X11/rgb.txt** file to be selected. If your X server is not using the **/usr/lib/X11/rgb.txt** file, the **Custom*rgbtxtPath** resource should be set to the path of the alternate **rgb.txt** file. Otherwise, the following error will occur if a color is selected that is not contained in the **rgb.txt** file in use by your X server:

```
The color name does not exist in /usr/lib/X11/rgb.txt
```

When you select a valid color from the list, the window directly below the list displays the chosen color, the RGB sliders are set to the RGB value of the chosen color, and the `Chosen Color` text field is set to the color name.

A color may also be set using the Red, Green, and Blue sliders, giving you finer control over color selection. When you move a slider bar, the window above the sliders reflects the color defined by the **RGB** slider values. The `Chosen Color` field is set to the RGB representation of the color, which begins with a `#` sign and is followed by the hexadecimal representation of the red, green, and blue values, respectively (for example, `#8456c8`).

After you position the **RGB** sliders to a desired color, you can press the **Match RGB to Closest Color Name** button. It finds the closest color name to the RGB value in the list of colors, highlights that color name in the list, and repositions the **RGB** sliders accordingly. The `Chosen Color` field is set to the value of this color.

When the **Select Color on Display with Mouse** button is pressed, the mouse cursor changes shape, prompting you to select any color that is displayed by clicking the left mouse button. The Colors browser then finds the RGB values for that color, repositions the slider bars, and also highlights the color name in the list of colors if it matches a valid color name.

You can press the **OK** button when you are satisfied and want to save the color choice, or you can press the **Cancel** button if you want to exit the Colors browser without saving. The help facility for the Colors browser is called by pressing the **Help** button. To exit the help facility, press the **Cancel** button.

Fonts Browser

The **Fonts** browser assists you in choosing font values. The fonts in the list describe fonts available in your system X server. There are six filters to narrow down font choices in the list. A specific family, weight, slant, style, spacing, or size can be chosen. All of these filters offer the option of **Any**, which lists all items of that category.

Some of the values that appear in these filters may not apply to the fonts that are available on your X server. If you select a value from one of these filter lists that does not apply to any of the fonts that are available, the browser displays an empty list of fonts and the following message:

```
0 fonts match the specified filters.
```

This situation may occur if your X server font path does not include paths to additional fonts installed on the system. You can add font paths using the following command:

xset +fp

When you choose values for the filters, the list is refiltered to show only the fonts conforming to the chosen values.

When you select a font from the list of fonts, it is displayed in the `Chosen Font` field. Your selection is also reflected in the `Sample` field.

The **Size** filter lets you specify the size of a font in points, either by selecting a size from the list or by typing a size into the `text` field beneath the list and pressing the Enter key. The list contains all the sizes of bitmap fonts that have been installed on your X server. In addition, your X server may have scalable fonts. These are fonts that are specially designed so that they can be drawn at any size you specify. To see a list of the scalable fonts available, type a size into the text field located below the **Size** filter that does not appear in the list and press the **Enter** key. A list of the scalable fonts scaled to the size you specified is displayed. When you select a size from the list of sizes or when you enter a size that also appears in the list, scaled fonts are included with the bitmap fonts in the list of fonts.

The **Toggle** button immediately above the `Chosen Font` field lets you specify that the character set encoding to be used is determined from the locale in effect when the application that you are customizing is run. This field is not a filter. Instead, it controls whether the chosen font is written in a language-independent way with the `*:` notation replacing the character set. The `*:` notation is of most concern if your language requires multiple fonts. Not all applications support the `*:` notation, but all AIXwindows applications do support it.

Press the **OK** button to write the value to the resource text field. Press the **Cancel** button to exit the browser without saving the value. Help for the **Fonts** browser is provided. To exit the help facility, press the **OK** button.

ISO 9241.3 Fonts

You can use the **Fonts** browser to help you choose fonts that meet the requirements of the ISO 9241, part 3, standard. This standard sets ergonomic guidelines for fonts so that text is easier to read.

For this feature of the **Fonts** browser to function correctly, you must first inform the X server of the physical dimensions of your display screen. You only need to do this set-up procedure one time. To set the physical size of your display, first use a ruler to measure the width and height of the image area of your display in millimeters (mm). Measure only the area of the screen that is actually used. Then log on as a user with superuser privileges and enter the following command:

smit chdispsz

Select the name of the display you are using from the list that is presented. Enter the width and height into the fields provided and press the **Enter** key to make the changes. Finally, exit all applications running on your system, and then shut down and reboot the system using the following command:

shutdown -Fr

When the system comes back up, the display is configured with the new size you set.

Choosing ISO 9241.3 Fonts

The fonts that have been specially designed to meet the ISO 9241.3 standard have iso9241 as the `ADD_STYLE_NAME` field of their XLFD names. You can list these fonts using the **Fonts** browser of the customizing tool by selecting the **ISO 9241.3** style filter.

However, a font that meets the standard on one display may not be satisfactory on another display. For example, a font that is fine on one display may appear too small to read on a display with higher resolution.

After selecting a font from the list, look for a message beneath the sample. If the characters in the font are large enough when shown on the display screen you are currently using, the following message is displayed:

```
When used on this display screen, this font meets
the requirements of the ISO 9241.3 standard.
```

Note: The ISO 9241.3 standard has guidelines for hardware display technology as well as the design of fonts. The Customizing Tool is not able to detect whether the display being used has the necessary characteristics. See the documentation that came with your display to see if it meets the requirements of ISO 9241.3 standard.

Pictures Browser

The **Pictures** browser allows bitmaps and pixmaps to be chosen and viewed. A Filter selection box, in which the path name is set to **/usr/lpp/X11/bitmaps/***, is provided to browse the pictures. Use the Pictures browser to change directories, view files, and select a file that contains a valid bitmap or pixmap. You can view the bitmap or pixmap by pressing the **View Picture** button on the **Pictures** browser window.

You can edit the bitmap or pixmap by pressing the **Edit Picture** button on the Pictures browser window. The editor is a separate application that exists on your system. It is called on your behalf. The **Custom*pictureEditor*editor** resource determines which editor commands to choose from. This resource accepts a list of commands separated by \n's (backslash `n's). The first command that identifies an existing program that the user has permission to execute is used. The file name in the `Chosen Picture` text field is passed as a parameter to the editor when it is invoked. The default setting for this resource is:

Custom*pictureEditor*editor:/usr/dt/bin/dticon -f \n\ /usr/lib/X11/bitmap

Note: The default editor, **/usr/dt/bin/dticon** only exists if the Common Desktop Environment (CDE) is installed. It edits both bitmaps (monochrome images) and pixmaps (color images). The **dticon** command accepts bitmaps stored in either the X Pixmap Version 2 Enhanced (XPM2) format which was used by the X Desktop (**xdt**) application shipped in AIXwindows Version 1.2.5, or X Pixmap Version 3 (XPM3) - a new XPG3 compliant format used by CDE. However, it requires pixmap images be stored in the XPM3 format. CDE has documented tools that can be used to convert pixmaps from the XPM2 to the XPM3 format.

The **/usr/bin/X11/bitmap** command is an unsupported sample program that accepts bitmaps in either the XPM2 or XPM3 formats. It does not support pixmap editing. Be sure that the Bitmap app-defaults file has been installed in the **/usr/lib/X11/app-defaults** directory before invoking the **bitmap** command. If not, issue the following commands in the **/usr/lpp/X11/Xamples/clients/bitmap** directory:

```
xmkmf;
make install
```

Error messages are displayed above the Chosen Picture text field. An informational message is usually displayed stating the type of picture (pixmap or bitmap) and its size.

Press the **OK** button to write the value to the resource text field. Press the **Filter** button to switch the file system. Press the **Cancel** button if you want to exit with no changes. Help is available for this editor.

Cursors Browser

The **Cursors** browser brings up a graphical list of cursors from which to choose. Applications use the cursor font for the picture on the mouse cursor. The cursor font names and indexes can be found in the **/usr/include/X11/cursorfont.h** file. A cursor can be selected from the list with the mouse button and is displayed in the `Chosen Cursor` field. You can press the **OK** button to write the value to the resource text field. The **Cancel** button to exit the editor without saving the value. Help for the cursor editor is provided.

Choices Browser

You can select several items from the list in the **Choices** Browser. The items are then joined and displayed in the `Chosen Items` field as they appear when saved to the resource database. For example, if the items in the list are translated into French, the value at the bottom still shows in an English specification. Press the **OK** button to write the value to the resource text field. Press the **Cancel** button to exit the browser without saving the value. Help for the **Choices** browser is available.

Filenames Browser

The **Filenames** browser is used to change directories, to view files, and to select files.

The `Filter` text field lets you display and edit a directory filter that is used to select the files to be displayed. The directory filter must be a string specifying the base directory to be examined and a search pattern.

The `Directories` field displays the subdirectories of the base directory, the base directory itself, and its parent directory. To change directories, double click on an item in this list.

The `Files` field displays all files in the base directory that match the search pattern.

The `Chosen Filename` field contains the current file name selection. It is updated whenever you select a file name from the `Files` field list.

If you press the **Filter** button, the `Files` field list is filtered to display all files and subdirectories in the base directory that match the search pattern.

Press the **OK** button to write the value to the resource text field. Press the **Cancel** button to exit the editor without saving the value. Help for the **Filenames** browser is provided.

Related Information

Using the AIXwindows Customizing Tool Overview
Using the Customizing Tool Main Window
Understanding the app-custom Files
How to Start the Customizing Tool

Understanding the app-custom Files

The resources that are important to customize should be listed in an **app-custom** file. The **app-custom** file contains all of the necessary information to generate a customizing tool interface that is unique for each application. For more information, see the following:

- Location of the app-custom Files
- Format of the app-custom Files
- Guidelines for Writing app-custom Files

Location of the app-custom Files

The customizing tool uses the application class to name **app-custom** files. Applications have two names: their application name, which is usually the name used to start the application, and their application class, which is permanently coded into the application.

By default, the customizing tool searches for the **app-custom** file in the following places, in the order listed. The first file found is used. (**$HOME** is the user's home directory. *Locale* is the locale in which **custom** is running. *Class* is the class of the application to be customized.)

1. **$HOME/***Locale***/app-custom/***Class*
2. **$HOME/app-custom/***Class*
3. **/usr/lib/X11/***Locale***/app-custom/***Class*
4. **/usr/lib/X11/app-custom/***Class*
5. **$HOME/***Locale***/app-custom/DEFAULT**
6. **$HOME/app-custom/DEFAULT**
7. **/usr/lib/X11/***Locale***/app-custom/DEFAULT**
8. **/usr/lib/X11/app-custom/DEFAULT**

The place where the customizing tool looks for the **app-custom** file can be changed by specifying the **Custom.appCustomPath** resource.

Format of the app-custom Files

Each line of the resource panel description file has the following format:

```
Group   Type    Resource
  Description       [Values];
```

Following is an explanation of these parameters:

Group Groups are used to organize similar resources. For example, scrollbar color, scrollbar on/off, and number of scrolled lines to save are all different types of resources, but they are related. The *Group* parameter can be any character string in any language you choose. If more than one word is used for the group or if it contains any double-byte character strings, it must be enclosed in quotation marks. You are limited to a maximum of 20 groups. The groups are listed on the **Resource Category** button on the Customizing Tool main window.

Type The *Type* parameter specifies whether the possible values of a resource should be limited to colors, fonts, or other types. The customizing tool provides graphical tools that assist in setting a valid value for the resource type. The type of a resource must be one of the predefined customizing tool types.

Resource
This is the actual resource string (for example, ***background**). The complete resource string must be listed exactly as it would be in a resource database; it *cannot* be prefaced by the application name or class and it *must* begin with an * (asterisk) or a . (period). This is required because the customizing tool attaches the application class to the beginning of each resource string prior to writing the string to the resource database.

The **custom** command also accepts the following syntax, which is intended for the Window Manager client-specific resources:

```
$MWM%% Resource
```

where each `%` is either an `*` (asterisk) or a `.`(period).

The customizing tool attempts to determine whether a window manager that is compatible with the mwm window manager is currently running. If so, it substitutes the name or class of the window manager for **$MWM** and inserts the class name of the application between the `**` (asterisks), `..` (periods), `*.` (asterisk, period), or `.*` (period, asterisk), represented by `%%` above.

If the customizing tool does not detect that a window manager known to be compatible with mwm is running, then all resource description statements containing **$MWM** are ignored.

For example, if `$MWM**clientDecoration` appears in the **app-custom** file for XClock and the **dtwm** window manager is currently running, the customizing tool expands this syntax to:

```
Dtwm*XClock*clientDecoration
```

If the **mwm** window manager is currently running, this syntax expands to:

```
mwm*XClock*clientDecoration
```

If no window manager compatible with **mwm** is running, then this entire statement (from the *Group* name to the **;** (semicolon)) is ignored, and this resource is not shown on the user interface.

Description

This is a descriptive explanation of each resource that you can customize. This string is chosen to represent the resource on the panel. It can be any descriptive text but should be brief because space is limited on the custom interface.

Although the resource string must be in English, the description can be in any language, which implies that language-specific **app-custom** files are localized files that are encoded in the current native locale at run time. It is also necessary to enclose any double-byte character string in double quotes.

The `\n` characters are recognized in description strings as a new-line character and can be used to break the description into multiple lines.

Values Some data types require additional data from which to create the interface.

The following provides a more detailed description of individual data type **app-custom** file syntax:

- The color, font, cursor, file name, number, and string data types all use the following syntax:

  ```
  Group   Type    Resource    Description;
  ```

  The following is an example of a color resource **app-custom** file syntax:

  ```
  "Menu Bar" Color *menubar*background "menu bar background";
  ```

- A picture data type can be specified in the **app-custom** file using the following syntax:

  ```
  Group   Picture Resource     Description PictureType;
  ```

  The following is an example of the **app-custom** file syntax:

  ```
  Icon    Picture *appIcon     "application icon"  bitmap;
  ```

  The *PictureType* value can be one of three types: **bitmap**, **pixmap**, or **all**. If **bitmap** is set, only bitmaps can be browsed; **pixmap** is similar. However, if **all** is set, both bitmaps and pixmaps are browsed.

  Bitmaps are black and white images where file names end with **.bm** or **.xbm**; *pixmaps* are multicolored images where file extensions are **.pm** or **.px**.

- The selectmany data type allows many values to be selected from a list of choices. It is limited to 20 items. A selectmany data type can be specified in the **app-custom** file using the following syntax:

```
Group   SelectMany       Resource        Description
        "Separator"      Item1 [= String] Item2 [= String]... ;
```

The *Separator* parameter specifies one or more characters that are used to separate each of the items the user selected when the resource definition is written to the **.Xdefaults** file. Following is an example of the **app-custom** file syntax:

```
Foods   SelectMany *iceCream  "Ice Cream Choices"
        " " vanilla chocolate strawberry ;
```

If you want to see more descriptive names for the choices, you can attach a string to each value with an = (equal sign). This is especially useful for translating values into other languages.

The following is an example:

```
Foods   SelectMany  *iceCream   "Ice Cream Choices"
        ", "
        vanilla = "Vanilla Flavor"
        chocolate = Fudge
        strawberry ;
```

- The selectone type is a generic data type for choosing one value from among a list of choices. It is limited to 20 items. A selectone data type can be specified in the **app-custom** file using the following syntax:

```
Group   SelectOne   Resource     Description
        Item1 [= String] Item2 [= String]... ;
```

Following is an example of the **app-custom** file syntax:

```
Foods   SelectOne   iceCream    "Ice Cream Choices"
                          vanilla chocolate strawberry ;
```

The selectone data type is represented by an **Option** menu button to switch between the values. The first option is always `<default>`. This is consistent with the other data types because no values are written to the resource database unless you have specifically chosen one.

You can select an item by pressing the **Option** button, dragging the mouse over the list, and releasing the mouse button on your choice.

If you want to see more descriptive names for the choices, you can attach a string to each value with an = (equal sign). This is especially useful for translating values to other languages.

The following is an example:

```
Foods   SelectOne   *iceCream  "Ice Cream Choices"
        vanilla = "Vanilla Flavor"
        chocolate = Fudge
        strawberry ;
```

- Comments must begin with an ! (exclamation point) as the first character on the line.

A brief example of an **app-custom** file follows:

```
! XClock app-custom file
Size      Number     *width        "width of clock";
Size      Number     *height       "height of clock";
Color     Color      *foreground   "foreground";
Color     Color      *hands        "analog hands";
Color     Color      *highlight    "highlight analog hands";
Color     SelectOne  *reverseVideo "reverse video"
          true false;
Font      Font       *font         "digital clock font";
Behavior  Number     *update       "interval of updates/n (sec)";
Behavior  SelectOne  *analog       "type of clock"
          true = analog false = digital;
Behavior  SelectOne  *chime        "chime every half hour"
          true false;
Behavior  Number     *padding      "internal padding (pixels)";
```

Guidelines for Writing app-custom Files

The following are guidelines for writing **app-custom** files:

- Creating app-custom Files
- Organizing Resource Categories
- Choosing Resources

Creating app-custom Files

If you already have resources in your **.Xdefaults** file for an application, you can load your resource values onto the customizing tool interface, remove the old resources from your **.Xdefaults** file, and use the customizing tool to customize the application in the future.

If the **app-custom** file does not contain all the resource strings you need, you can create a subdirectory named **app-custom** in your **$HOME** directory, copy the **app-custom** file from the **/usr/lib/X11/app-custom** directory into it, and add the needed resource strings to this file. To create a new **app-custom** file, copy the **DEFAULT app-custom** file and use it as a template.

If you want to create a new **app-custom** file, you should add the **app-custom** file and the application name to the **Custom.listOfApps** resource. This resource is used to display the application names on the starting dialog. The application name and corresponding **app-custom** file must be listed in pairs with the following syntax:

```
Application:app-custom,
```

For example:

```
Custom.listOfApps:
xclock:XClock,custom:Custom,msmit:Msmit
```

You can specify a maximum of 100 applications.

Organizing Resource Categories

Group resources into the following recommended categories:

- Colors
- Fonts
- Size and position
- Icon
- Graphics
- Behaviors

If the application resources do not fall into one of these categories, you can create additional categories. The new categories can be added after the other categories. You can create a new category if any of the standard categories contain too many resources (more than 15 or 20). The new category needs to be added after the category whose limit was exceeded.

Choosing Resources

The following are some general guidelines for selecting which resources to list in **app-custom** files:

- The resources in each category should be those resources the application writer decides are commonly used or those that may be important to customize.
- Do not list the same resource more than once in a given **app-custom** file. Doing so can lead to unexpected results.
- Although colors and fonts can be set on each object or groups of objects, try to keep it simple. For example, `*background` sets the background color of the application to one color.

- You should append an * (asterisk) to the name of window manager client resources. This convention tells the user that the window manager has to be restarted for changes to these resources to take effect. For example,

```
$MWM**clientDecoration    "client decoration *"
```

- Resource strings should not be prefaced with the application name or class because the customizing tool prepends the class name to the resource string when saving to the resource file.
- The descriptions of the resource should be no more than 20 to 30 characters long. If longer descriptions are needed, the \n (new-line) character can be used to break the description into multiple lines.
- The maximum number of bytes in any quoted string is 500.

Once you create your **app-custom** file, you can check the syntax by running the customizing tool.

Related Information

Using the AIXwindows Customizing Tool Overview

Using the Customizing Tool Main Window

Using the Browsers

How to Start the Customizing Tool

The **xset** command

Chapter 4. AIXwindows Xlib Library

The following major sections describe the **Xlib** library and its functions:
- AIXwindows Xlib Introduction
- Using Displays
- Graphical Overlay Planes
- National Language Support (NLS) Reference Overview
- Suggested Reading

Xlib Introduction

When users enter input on a terminal, an X server distributes that input to client programs that are on the same system or elsewhere in the network. The X server then returns the requested actions, such as refreshing the screen or starting an application. The windowing interface is consistent across platforms and remains consistent because the **Xlib** library controls system calls. The **Xlib** library is a C language function library that client programs use to communicate with the windowing system. Calls to the client are sent through the **Xlib** library, and return information passes back through the **Xlib** library, which translates information to a standard X-Window language.

As a programmer, the application program you create is the client of the client-server relationship; you write the program and the X server gives it hardware independence. There is one X server for each virtual terminal that runs AIXwindows.

A *window* is a rectangular area on the screen that displays graphical output. Client applications can display overlapping and nested windows on one or more screens that are driven by X servers on one or more systems. There can be one or more physical screens each containing several windows. (The term *screen* refers to a physical monitor, which can be color or black and white, and associated hardware.)

Compiling X Programs with AIXwindows

The following compiler command can be used to build your program:

```
cc {Compiler Options} -oSample Sample.c -lX11
```

In this example, *Sample*.c is the name of your C language source program, *Sample* is the name of your executable C program and `-lX11` indicates the function library (**/usr/lib/libX11.a**). The `-l` is the flag and `X11` indicates the library.

The compiler definitions for structures, parameters, error codes, and data types are located in the **/usr/include/X11/Xlib.h** and **/usr/include/X11/X.h** files. You must include these files in any program that uses these functions. To include these files, enter the following statement early in your program:

```
#include <X11/Xlib.h> /* also includes X11/X.h */
```

Window Relationships

The windows in an X server are arranged in a strict hierarchy. At the top of the hierarchy is the root window, which covers the entire display screen. The root window is partially or completely covered by lower-level child windows. The root window is parent to its child windows, and all windows except the root have parents. Child windows can, in turn, have their own children, which allows an application program to create an organizational tree on the screen. There is usually at least one window per application.

A child window can be larger than its parent, and part or all of the child window can extend beyond the boundaries of the parent. However, moving a child window outside the parent clips it; the only visible part of the window is that part within the parent's boundaries.

© Copyright IBM Corp. 1997, 2002

If several children of a window have overlapping locations, one of the children is considered to be on top of or raised over the others, obscuring them. Output to areas covered by other windows is not displayed; it is suppressed by the window system.

Window Attributes

A window has a border that can be any pattern or solid color. A window usually has a background pattern that the window system repaints when the window is uncovered. (To keep track of objects within, each window has its own coordinate system.) Child windows obscure their parents unless the child has no background. Graphic operations in the parent window are usually clipped by the child windows.

Input takes the form of *events*. Events can be side effects of a command (for example, restacking windows generates exposure events) or they can be completely asynchronous (keyboard entries). AIXwindows never sends events that a client application did not request; the client application must request to be informed of events.

AIXwindows does not take responsibility for the contents of windows. When part or all of a window is hidden and then brought back onto the screen, its contents may be lost. The client program is then notified by an exposure event that part or all of the window needs to be repainted. Your programs must be prepared to regenerate the contents of windows on demand.

AIXwindows provides off-screen storage of graphics objects, called *pixmaps*. Keeping graphics off screen allows the system to redraw windows more quickly because the information within them is still in memory. Single plane (depth 1) pixmaps are sometimes referred to as *bitmaps*. Bitmaps can be used in most graphics functions interchangeably with windows and are used in various graphics operations to define patterns, also called *tiles*. (Windows and pixmaps together are referred to as *drawables*.)

Window Caveats

Most of the functions in the **Xlib** library only add requests to an output buffer. These requests run later (and asynchronously) on the X server. Functions that attempt to return values stored in the X server do not return; these functions block until the X server sends an explicit reply or an error occurs. If a nonblocking call results in an error, the error is generally not reported by an optional error handler until a later blocking call is made.

If AIXwindows does not process a request asynchronously, a client can follow the request with a call to the **XSync** function. The **XSync** function blocks until all previously buffered asynchronous events are sent and acted upon. The output buffer is always erased by a call to any function that returns a value or waits for input (for example, the **XPending**, **XNextEvent**, **XWindowEvent**, **XFlush**, or **XSynchronize** functions).

Many **Xlib** functions return an integer resource ID that allows you to refer to objects stored on the X server. These objects can be of type **Window**, **Font**, **Pixmap**, **Bitmap**, **Cursor**, and **GContext**, as defined in the file **<X11/X.h>**. (The < > is defined by the **#include** statement of the C compiler and is a file relative to a well-known directory. In the AIX Operating System this directory is **/usr/include**.)

Some functions return *Status*, which is an integer error indication. If the function fails, the *Status* is 0 (zero). A status of 0 indicates the function did not update the return parameters. Because the C language does not provide multiple return values, many functions must return their results by writing to client-passed storage. A pointer that is used to return a value is designated by the *Return* suffix as part of its name. All other pointers passed to these functions are used for reading only. By default, errors are handled either by a standard library function or by a library function that you provide. Functions that return pointers to strings return NULL pointers if the string does not exist.

Input events (for example, key press events or mouse activity events) arrive asynchronously from the server and are queued until they are requested by a call to the **XNextEvent** or **XWindowEvent** functions. In addition, some of the library functions (such as the **XResizeWindow** and **XRaiseWindow** functions) generate exposure events or requests to repaint sections of a window that do not have valid contents.

These events also arrive asynchronously, but the client can wait for them by explicitly calling the **XSync** function after calling a function that generates exposure events.

Suggested Reading

AIXwindows Overview provides a conceptual introduction to using AIXwindows functions, macros, protocols, extensions, and events. The AIXwindows program is a tool designed to help enhance the usability of the overall application processing environment.

Using Display Functions in AIXwindows

Before a program can use a display, it must establish a connection to the X server driving the display. Once users establish a connection, they can use the **Xlib** macros and functions to return information about this display, such as image format size and depth.

The following information describes how to open windows. The two major sections are as follows:

- Opening a Display
- Shared Memory Transport (SMT)

Opening a Display

To open a connection to the X server controlling a specified display, use the **XOpenDisplay** function. This function returns a display structure that serves as the connection to the X server. This display structure contains information about the X server and connects the specified hardware display to the server through Transmission Control Protocol (TCP) or UNIX domain sockets. If the host name is a host system name and a : (colon) separates the host name and display number, the **XOpenDisplay** function connects the host and the display using TCP sockets. If the host name does not exist, or if `unix` and a **:** (colon) separates it from the display number, the **XOpenDisplay** function connects the host and the display using UNIX domain sockets.

A single server can support any or all of these transport mechanisms simultaneously.

The display name or **DISPLAY** environment variable is a string with the following format:

```
HostName:Number.Screen
```

HostName
: Specifies the name of the host system where the display is physically attached. The host name should be followed by a : (colon).

Number
: Specifies the ID number of the display server on that host machine. The display number can be followed by a . (period).

Screen
: Specifies the number of the screen on that host server. Multiple screens can be connected to or controlled by a single X server. The screen sets an internal variable that can be accessed with the **DefaultScreen** macro or the **XDefaultScreen** function.

For example, you can use the following format to specify screen `0` display `2` on the system named `Dave`**:**

```
Dave:2.0
```

Applications should not directly modify any part of the **Display** and **Screen** data structures. These structures should be considered read-only by the user, but they can be changed by specified functions or display macros.

Note: Communication from the Xlib to the X server is handled in one of the following three ways:

- UNIX domain sockets

- TCP domain sockets
- Shared Memory Transport Sockets

See Shared Memory Transport (SMT) for more information.

Shared Memory Transport (SMT)

Communication from **Xlib** to the X windows server is handled by UNIX domain sockets, TCP domain sockets, and Shared Memory Transport sockets. UNIX domain sockets constitute a mechanism that was an inherited mechanism for protocol transport. TCP sockets are used on a system when the application (client) running is on a remote machine. Shared Memory Transport sockets constitute a newly developed mechanism that is specifically designed for the X server.

The Shared Memory Transport sockets mechanism was developed to increase performance of the transfer data from the client application (through **Xlib**) to the X server. As shown in the following illustration, data is passed to and from the X server through one of the three transport mechanisms:

- UNIX Domain Sockets.
- TCP DOmain Sockets.
- Shared Memory Transport Sockets.

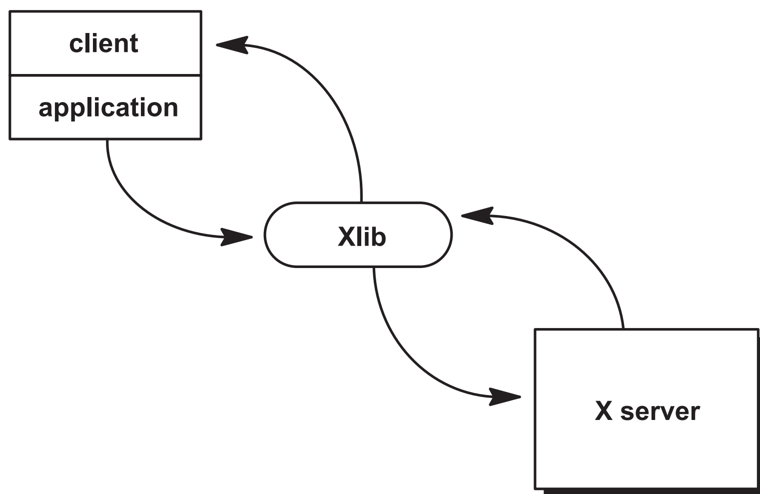


Figure 2. Xlib Data Flow. This diagram shows data passing from the client application through the Xlib to the X server and back.

The default protocol transport method is a convention inherited from the Massachusetts Institute of Technology (MIT) which is the fastest transport method available.

Data has shown that the communications overhead for the local X applications can be significant and can be improved by using SMT sockets rather than UNIX or TCP sockets as the transport mechanism. The standard communication mechanism for local X clients is UNIX domain sockets. This design provides a set of services to facilitate the use of shared memory as a transport mechanism for local clients.

Goals of the Shared Memory Transport Design

The Shared Memory Transport (SMT) design assists in eliminating unnecessary data movement. When UNIX domain sockets are used for communication, the data must be copied twice to move it between processes. This data is copied into the kernel on ″sends″ and out of the kernel on ″receives.″ By using a shared memory segment for the data, both the client and server processes can access the same data. This saves one or more copies of the data.

When multiple applications are running on a single server, little data may be transferred to or from a client before another client requires service. In this type of scenario, the server spends much of its time in the [select] system call in order to switch to another process.

The select system call contains much overhead that the X server does not need. The SMT design implements a proprietary select called smselect. This SMT system call eliminates much of the unnecessary overhead, minimizing the process switching time between the applications, and improving performance.

Environment Variables and Shared Memory Transport

The X server uses SMT by default, which is transparent to the user (except for performance differences). The user can, however, make use of two environment variables that can change the way the transport layer works. These environment variables are: **DISPLAY** and **X_SHM_SIZE**.

DISPLAY is an existing environment variable that is commonly manipulated by X server users. This variable is used to specify the connection to a specific X server. It can specify any number of either local or remote X servers. By default, the **DISPLAY** environment variable is set to `:screen#`. This convention specifies that the user would like to connect to the X server specified by `screen#` and use the fastest available transport method. If the **DISPLAY** variable is set to `unix:screen#` then UNIX domain sockets are used. If the variable is set to `hostname:screen#` then TCP sockets are used to transport the data to the remote X server.

X_SHM_SIZE is an environment variable to X11R5 1.2.3 (Release 3.2.4). This variable can be used to change the size of the client-to-server and server-to-client buffer sizes. The default buffer size is 64K bytes, the minimum size is 16K bytes, and the maximum size is 252K bytes. The user may wish to change the size of the buffers proportionally to the specific application needs. For example, large buffers for large data sets, and small buffers for small data sets.

Other Shared Memory Mechanisms

It is important to note that the Shared Memory Transport mechanism is not the same as the Shared Memory Extension shipped by MIT. The extension operates solely on pixmaps and images and requires that the client application have detailed knowledge of the transport mechanism. In this scenario, the client makes specific calls to the server-side extension to gain addressability to a shared memory segment simultaneously with the server.

Shared Memory Transport, on the other hand, utilizes the shared memory segment to transport all data (not just pixmaps and images) to and from the X server. The client is not required to do any special processing. The **Xlib** program handles the transport mechanism for the client application.

Graphical Overlay Planes

Graphical overlay planes are made up of additional memory positioned logically on top of the frame buffer (thus the name overlay). Typically, graphics applications would use this extra hardware for creating windows that would not disturb the contents of existing windows in the frame buffer.

The primary goals for implementing this feature are:

- To reduce the need for redrawing underlying complex images while the user is navigating through the user interface.
- To allow simple images, such as annotations, to be written over underlying images without overwriting the images.

At X server startup/reset time, the following property is created on the root window of any screen that has overlay capabilities:

```
property_name:  SERVER_OVERLAY_VISUALS
property_type:  SERVER_OVERLAY_VISUALS
format:         32
```

The contents of the property is a list of each visual that resides in the overlays. Each element of the list contains information in the following format:

```
SERVER_OVERLAY_VISUALS {
        overlay_visual:         VISUALID
        transparent_type:       {None, TransparentPixel,
                                TransparentMask}
        value:                  CARD32
        layer:                  CARD32
    }
```

Visuals in layer 0 (zero) are nonoverlay visuals, and would not appear in the list. Overlay visuals start in layer 1 (one). (See the *-layer* flag of the **X** command for information concerning how to specify the default visual.)

Drawing into a window created with an overlay visual (an *overlay window*) does not disturb the contents of windows in layers different from the layer containing the overlay window. Additionally, there is a set (perhaps empty) of transparent pixel values such that when written into an overlay window, pixels of windows in the next lower-numbered layer show through. If these pixel values also happen to be transparent, the process is applied until a window containing a nontransparent pixel value is found. Since visuals in layer 0 are not displayed in the list, they cannot have any transparent pixel values. The `transparent_type` and `value` fields specify the set of transparent pixel values as follows:

transparent_type	**Meaning of Value Field**
0 - None	Ignore the value field; there are no transparent pixels.
1 - TransparentPixel	The value field explicitly names a transparent pixel.
2 - TransparentMask	Any pixel value that has at least the same bits on as the `value` field is transparent. A pixel is transparent if and only if (value & pixel) == value.

The same `overlay_visual` element may appear more than once in the list. If it does:

- The union of the pixel values described by the `transparent_type` and `value` fields should all be transparent.
- The value of the `layer` field is the same across all instances of a given `overlay_visual` element. (A given `overlay_visual` element exists in one and only one layer.)

Note: There is no restriction on the parent-child relationships of windows with respect to the visual with which the windows were created. A window can have combinations of children in any layer on, below, or above its own layer.

GXT_OVERLAYS

On the GXT150 and GXT155 families of graphics adapters (except for the GXT150M), the **GXT_OVERLAYS** environment variable must be set to a value of `TRUE` to enable 4-bit overlays.

- If the **GXT_OVERLAYS** environment variable is set to a value of `TRUE`:
  - On low resolution monitors (1024x768), overlays and multiple hardware color maps are enabled.
  - On high resolution monitors, only overlays are enabled. The 4-bit overlays take over the video memory used to enable multiple hardware colormaps.
  - A command line argument of `-layer 1` or `-d 4` causes the X server to place the root window and default visual in the 4-bit overlay planes.
- If the **GXT_OVERLAYS** environment variable is *not* set to a value of `TRUE`:
  - On both low and high resolution monitors, only multiple hardware color maps are enabled. The 4-bit overlays are not available.
  - A command line argument of `-layer 1` is ignored by the X server.

– A command line argument of `-d 4` is considered invalid by the X server.

AIXwindows National Language Support (NLS)

Overview

Internationalization is a method of application development that enables an application to be used in a variety of *locales*. The concept of a locale is used to encompass the characteristics and requirements of a given language. An internationalized application requires that any language-dependent or custom-dependent information be stored external to the application program. A locale can be characterized by several components. For example, character sets, sort order, text direction, and formats for data (such as date and currency) are used to describe the unique characteristics of a locale. For each different locale, information (such as menu items, help information, and user prompts) is defined and stored separately. The information is localized, which means that it has been tailored for a specific language and country.

An application designed to run in different locales examines certain resources and environment variables in order to determine which language to use when run. An application establishes file search paths for resource files and other language-dependent information. There is a wide variety of information that can be localized. Localized information is stored in files that reside in different directories. AIXwindows uses an underlying Xt Intrinsics mechanism (**XtResolvePathname**) to select and locate the appropriate files, depending on the locale.

An internationalized application can set its locale at run time, typically using an internal procedure called a language procedure. The language procedure processes data in your user environment and sets the application's locale. This feature enables you to modify your user environment and rerun an application in a different locale. An application can use a default language procedure or supply its own procedure. As a result, two applications can process the same user environment data with different results. Refer to the user documentation for the application you intend to use for detailed instructions.

An *input method* is the underlying mechanism that takes keyboard input and displays the locale's corresponding characters on the screen. An input method interprets the user's keystrokes based on the conventions supplied by the input method. The input method used by an application based on an alphabetic language is generally not visible. However, ideographic languages such as Chinese or Japanese may use an input method that composes keystrokes in a separate window called a pre-edit window. For example, you type a phonetic representation of a spoken word and the input method determines the ideographic character that is pronounced in that way. When more than one character meets the criteria, the input method displays a list of characters from which to select. Once you confirm a selection in the pre-edit area, the information is passed to the application.

Input methods can be defined by the platform vendor, an application, or a user. Information about available input methods and their features can be found in the documentation for the system or application that you are using.

An internationalized application can dynamically set its language environment when it is run. The design of the application defines what information is expected and how it is used to set the language environment. Typically, you can modify the language environment used by an application by specifying a language resource or a language environment variable. The exact method that you use to set the language environment for an application is defined by the application. Refer to the documentation for the application for specific information.

Exceptions

The AIXwindows system supports the standard X11R6 with the following exceptions:

Input Method Values

- **XNVisiblePosition** argument indicates whether the visible position masks of **XIMFeedback** in **XIMText** are available.
- **XNR6PreeditCallbackBehavior** is obsolete and its use is not recommended. The argument originally indicated whether the behavior of preedit callbacks regarding **XIMPreeditDrawCallbackStruct** values followed Released 5 or Release 6 semantics.

Input Context Values

- **String Conversion Callback** argument speficies a string conversion callback. This callback is not required for correct operation of either the input method or the client.
- **String Conversion** argument specifies the string to be converted by an input method. This argument is not required for correct operation of either the input method or the client.

Chapter 5. Using Extensions in AIXwindows

Because AIXwindows can evolve by extensions to the core protocol, it is important that extensions not be given lesser importance.

To avoid initializing extensions explicitly in application programs, it is important that extensions perform ″lazy evaluations″ and automatically initialize themselves when called the first time.

The following sections describe techniques for writing extensions to the **Xlib** library that have the same performance rate as Core protocol requests.
- Extension Functions
- Dynamically Loadable X Server Extensions

Note: Because an AIXwindows extension is expected to consist of multiple requests, defining 10 new features as 10 separate extensions is not a good practice. Rather, package new features into a single extension and use minor opcodes to distinguish between the features.

The following extension protocols are included in this chapter:
- AIXwindows Nonrectangular Window Shape Extension
- AIXwindows Screen Saver Extension
- AIXwindows Input Device Functions
- AIXwindows Input Extension Library
- AIXwindows Input Extension Protocol
- AIXwindows Double Buffer Extension
- AIXwindows XTest Extension

Extension Functions

The following are extensions function topics:
- Basic Extension Functions
- Hooking into the Xlib Library
- Graphics Context (GC) Caching
- Graphics Batching
- Lock Data Structures
- Define Extension Stubs, Requests, and Replies
- Define Request Formats
- Define Reply Formats
- Send Protocol Request and Arguments
- Variable Length Arguments
- Synchronous Calling
- Allocating and Deallocating Memory
- Deriving the Correct Extension Opcode

Basic Extension Functions

The basic protocol requests for extensions are the **XQueryExtension** extension function, the **XListExtensions** extension function, and the **XFreeExtensionList** extension function.

© Copyright IBM Corp. 1997, 2002

Hooking into the Xlib Library

Hooking routines sink a connecting hook into the library. These routines normally are not used by application programmers but, instead, by programmers who need to extend the Core AIXwindows protocol and the AIXwindows library interface. Hooking routines, which generate protocol requests for AIXwindows, are called stubs.

In extensions, stubs first check to see if they have initialized themselves on a connection. If the stubs have not been initialized, they should call the **XInitExtension** function.

The wire-formatted structure **XEvent** is in the **<X11/Xproto.h>** header file, and the host-formatted structure **XEvent** is in the **<X11/Xlib.h>** header file.

The **XExtCodes** data structure returns the information from the **XQueryExtension** extension function and is defined in the **<X11/Xlib.h>** header file.

The **XInitExtension** function calls the **XQueryExtension** function to see if the extension exists. Then, it allocates storage for maintaining the information about the extension on the connection. It chains this to the extension list for the connection, and returns the information that the stub implementor needs to access the extension. If the extension does not exist, the **XInitExtension** function returns NULL.

The extension number returned in the **XExtCodes** data structure is used in the other calls that follow. This extension number is unique to a single connection only.

```
XExtCodes *XAddExtension(DisplayPtr)
        Display *DisplayPtr;
```

For local **Xlib** library extensions, the **XAddExtension** function allocates the **XExtCodes** data structure, bumps the extension number count, and chains the extension onto the extension list. (This allows extensions access to the **Xlib** library without requiring server extensions.)

The types of functions and associated functions that hook into the AIXwindows library include the following:

- Creating a new graphics context for a connection (the **XESetCloseDisplay** extension function and the **XESetCreateGC** extension function).
- Copying a graphics context (the **XESetCopyGC** extension function).
- Freeing a graphics context (the **XESetFreeGC** extension function).
- Creating and freeing fonts (the **XESetCreateFont** extension function and the **XESetFreeFont** extension function).
- Converting events defined by extensions to and from wire format (the **XESetWireToEvent** function and the **XESetEventToWire** extension function).
- Handling errors (the **XESetError** extension function, and the **XESetErrorString** extension function.

Use these routines to define procedures to be called under certain circumstances. All these routines return the previous routine defined for this extension.

Various **Xlib** library data structures have provisions for extension functions to chain extension-supplied data onto a list. Because the list pointer is always the first member of the structure, a single set of functions can be used to manipulate the data in these lists.

The **XExtData** data structure is used in these functions, and is defined in the **<X11/Xlib.h>** header file.

Graphics Context (GC) Caching

GCs are cached by the library so that independent change requests can be merged into a single protocol request. This cache is called a write back cache. Any extension function whose behavior depends on the contents of a GC must erase the GC cache to make sure the server has up-to-date contents in its GC.

If you extend the GC to add additional resource ID components, you should ensure that the library stub immediately sends the change request. Since a client can free a resource immediately after using it, storing the value in the cache without forcing a protocol request can destroy the resource before it is set into the GC.

The **_XFlushGCCache** procedure forces the cache to be erased.

Graphics Batching

If you extend AIXwindows to add more poly-graphics primitives, you might be able to take advantage of facilities in the library to allow back-to-back single calls to be transformed into poly-requests. The display structure has a pointer to an **xReq** called last_req, which is the last request being processed. By checking that the last request type, drawable, GC, and other options are the same as the new one, and that there is enough space left in the buffer, you might be able to extend the previous graphics request by extending the length field of the request and appending the data to the buffer.

For example, the following is the source for the **XDrawPoint** stub:

```
#include <X11/Xlibint.h>
/* precompute the max size of batching request allowed */
   static int size = sizeof(xPolyPointReq) + EPERBATCH
   * sizeof(xPoint);
XDrawPoint(dpy, d, gc, x, y)
   register Display *dpy;
   Drawable d;
   GC gc;
   int x, y;          /* INT16 */
{
   xPoint *point;
   LockDisplay(dpy);
   FlushGC(dpy, gc);
   {
   register xPolyPointReq *req = (xPolyPointReq *)
   dpy->last_req;
   /* if same as previous request, with same drawable,
   batch requests */
   if (
      (req->reqType == X_PolyPoint)
      && (req->drawable == d)
      && (req->gc == gc->gid)
      && (req->coordMode == CoordModeOrigin)
      && ((dpy->bufptr + sizeof (xPoint)) <=dpy->bufmax)
      && (((char *)dpy->bufptr - (char *)req) < size)) {
       point = (xPoint *) dpy->bufptr;
       req->length += sizeof (xPoint) >> 2;
       dpy->bufptr += sizeof (xPoint);
       }
   else {
      GetReqExtra(PolyPoint, 4, req ); /* 1 point = 4 bytes */
      req->drawable = d;
      req->gc = gc->gid;
      req->coordMode = CoordModeOrigin;
      point = (xPoint *) (req + 1);
      }
   point->x = x;
   point->y = y;
```

```
   }
   UnlockDisplay(dpy:);
   SyncHandle();
}
```

To keep clients from generating long requests that might monopolize the server, there is a limit of **EPERBATCH** defined in **<X11/Xlib.h>** on the number of requests batched. Note that the **FlushGC** macro is called before picking up the value of the `last_req` field, since it may modify this field.

Lock Data Structures

To lock the display structure for systems that want to support multithreaded access to a single display connection or to support asynchronous input, each stub needs to lock its critical section. Generally, this section is the point immediately prior to the appropriate **GetReq** call when all arguments to the call have been stored into the request. Two calls generally implemented as macros are:

```
LockDisplay(Display)
         Display *Display;
UnlockDisplay(Display)
         Display *Display;
```

The *Display* parameter specifies a pointer to the display structure of the display to be locked or unlocked.

Define Extension Stubs, Requests, and Replies

The **<X11/Xproto.h>** header file contains the following three sets of definitions:

- Request names
- Request structures
- Reply structures

X server requests contain the length, expressed as a 16-bit quantity of 32-bits, of the request. Therefore, a single request can be no more than 256k long. Some servers may not support single requests of such a length. The value of `display ->max_request_size` contains the maximum length as defined by the server implementation.

An **Xlib** library stub routine should start as follows:

```
#include <X11/Xlibint.h>
        XDoSomething (arguments, ...) /* argument declarations */
        {
/* variable declarations, if any */
```

If the protocol request has a reply, then the variable declarations should include the reply structure for the request. The following is an example:

```
xDoSomethingReply rep;
```

Generate a file equivalent to the **<X11/Xproto.h>** header file for your extension and include it in your stub routine. Each stub routine also must include the **<X11/Xlibint.h>** header file.

The identifiers are deliberately chosen in such a way that if the request is called `X_DoSomething`, then its request structure is `xDoSomethingReq` and its reply is `xDoSomethingReply`. The **GetReq** family of macros, defined in the **<X11/Xlibint.h>** header file, takes advantage of this naming scheme.

For each X request, there is a definition in the **<X11/Xproto.h>** that looks similar to the following:

```
#define X_DoSomething 42
```

Note: In your extension header file, this is a minor opcode instead of a major opcode.

Define Request Formats

Every request contains an 8-bit major opcode and a 16-bit length field expressed in units of 4 bytes. Every request consists of a 4-byte header (containing the major opcode, the length field, and a data byte) followed by a 0 (zero) or additional bytes of data. The length field defines the total length of the request, including the header. The length field in a request must equal the minimum length required to contain the request. If the specified length is smaller or larger than the required length, the extension should generate a **BadLength** error. Unused bytes in a request are not required to be the value of 0.

The **XMaxRequestSize** extension function returns the maximum request size (4-byte units) supported by the server.

Major opcodes 128 through 255 are reserved for extensions. Extensions are for holding multiple requests, so extension requests typically have an additional minor opcode encoded in the spare data byte in the request header. But the placement and interpretation of this minor opcode, as well as all other fields in extension requests, are not defined by the Core protocol. Every request is implicitly assigned a sequence number (starting with one) used in replies, errors, and events. The **B16** and **B32** macros have been defined so that they can become bit-field specifications on some machines.

Most protocol requests have a corresponding structure **typedef** in the **<X11/Xproto.h>** header file. The following is an example of the **xResourceReq** typedef structure:

```
typedef struct _ResourceReq{
   CARD8 reqType;       /* the request type, X_DoSomething */
   BYTE pad;            /* not used */
   CARD16 length;       /* 2 (= total number of bytes in
                           request, divided by 4)          */
   CARD32 id;           /* the window, drawable, font, or
                           gcontext, for example           */
} xResourceReq;
```

reqType
: Identifies the type of the request, such as the **X_MapWindow** value or the **X_CreatePixmap** value.

length
: Identifies how long (in units of 4 bytes) the request is. It includes both the request structure and any variable length data, such as strings or lists, that follow the request structure. Request structures come in different sizes, but all requests are padded to be a multiple of 4-bytes long.

If a Core protocol request has a single 32-bit argument, you do not need to declare a request structure in your extension header file. Instead, such requests use the **xResourceReq** data structure in the **<X11/Xproto.h>** header file. This structure is used for any request whose single argument is *Window*, *Pixmap*, *Drawable*, *GContext*, *Font*, *Cursor*, *Colormap*, *Atom*, or *VisualID*.

```
typedef struct _DoSomethingReq{
   CARD8 reqType;    /* X_DoSomething                     */
   CARD8 someDatum;  /* used differently in different requests*/
   CARD16 length;    /* total number of bytes in request,
                        divided by 4                      */
   ....              /* request-specific data             */
   ...
} xDoSomethingReq;
```

reqType
: Identifies the type of the request, such as the **X_MapWindow** value or the **X_CreatePixmap** value.

length
: Identifies how long (in units of 4 bytes) the request is. It includes both the request structure and any variable length data, such as strings or lists, that follow the request structure. Request structures come in different sizes, but all requests are padded to be a multiple of 4-bytes long.

You can do something similar in your extension header file.

A few protocol requests take no arguments at all. Instead, they use the **xReq** data structure, which contains only a request type and a length (and a pad byte), in the **<X11/Xproto.h>** header file.

Define Reply Formats

If the protocol request requires a reply, then the **<Xproto.h>** header file also contains a reply structure typedef.

```
typedef struct _DoSomethingReply {
   BYTE type;             /* always X_Reply               */
   BYTE someDatum         /* used differently in different
                              requests                    */
   CARD16 sequenceNumber; /* number of requests sent so far*/
   CARD32 length;         /* number of additional bytes,
                              divided by 4                */
   ....
                          /* request-specific data        */
   ....
} xDoSomethingReply;
```

Most of these reply structures are 32 bytes long. If the reply value is less than 32 bytes, the reply structure contains a sufficient number of pad fields to bring them up to 32 bytes.

The *length* is the total number of bytes in the request minus 32, divided by 4. This field is not the value of 0 if:

- The reply structure is followed by variable length data, such as a list or string.
- The reply structure is longer than 32 bytes.

The only Core protocol exceptions that have reply structures longer than 32 bytes are as follows:

- **GetWindowAttributes** protocol request
- **QueryFont** protocol request
- **QueryKeymap** protocol request
- **GetKeyboardControl** protocol request

A few protocol requests return replies that contain no data. The **<X11/Xproto.h>** header file does not define reply structures for these. Instead, these protocol requests use the **xGenericReply** structure, which contains only a type, length, and sequence number (and sufficient padding to make it 32-bytes long).

Send Protocol Requests and Arguments

After the variable declarations, a stub routine should call one of the following four macros defined in the **Xlibint.h** file:

- **GetReq**
- **GetReqExtra**
- **GetResReq**
- **GetEmptyReq**

These macros take the name of the protocol request as declared in the **<X11/Xproto.h>** header file without the **X_** as their first argument. Each macro declares a **Display** structure pointer, called *dpy* and a pointer to a request structure, called *req*, which is of the appropriate type. The macro then appends the request structure to the output buffer, fills in the `type` and `length` field, and sets the *req* variable to point to it.

If the protocol request, such as **GrabServer**, has no arguments, use the **GetEmptyReq** protocol request as in the following example:

```
GetEmptyReq (DoSomething, req);
```

If the protocol request has a single 32-bit argument (such as a *Pixmap*, *Window*, *Drawable*, or *Atom*), use the **GetResReq** macro.

The second argument to this macro is the 32-bit object. The **X_MapWindow** request type is a good example of the **GetResReq** macro:

```
GetResReq (DoSomething, rid, req);
```

The *rid* argument is the **Pixmap** or **Window** value, or other resource ID.

If the protocol request takes any other argument list, then call the **GetReq** macro. After the **GetReq** macro, set all the other fields in the request structure, usually from arguments to the stub routine.

```
GetReq (DoSomething, req);
/* fill in arguments here */
req->arg1 = arg1;
req->arg2 = arg2;
```

A few stub routines, such as the **XCreateGC** function and the **XCreatePixmap** function, return a resource ID to the caller but pass a resource ID as an argument to the protocol request. These stub routines use the **XAllocID** macro to allocate a resource ID from the range of IDs that were assigned to this client when it opened the connection. The following is an example of the **XAllocID** macro:

```
rid = req->rid = XAllocID();
return (rid);
```

Finally, some stub routines transmit a fixed amount of variable-length data after the request. Typically, these routines, such as the **XMoveWindow** function and the **XSetBackground** function, are special cases of more general routines like the **XMoveResizeWindow** function and the **XChangeGC** function. In these cases, the **GetReqExtra** macro, which is like the **GetReq** macro with an additional argument, is used. The additional argument is the number of extra bytes (a multiple of 4) allocated in the output buffer after the request structure.

Variable Length Arguments

Some protocol requests take additional variable length data that follow the **xDoSomethingReq** structure. The format of this data varies from one request to another. Some require a sequence of 8-bit bytes, others a sequence of 16-bit or 32-bit entities, and still others a sequence of structures.

The length of any variable length data must be added to the `length` field of the request structure. The `length` field is in units of 32-bit long words. If the data is a string or other sequence of 8-bit bytes, then round up the length and shift it before adding. For example:

```
req->length += (nbytes+3)>>2;
```

To transmit the variable length data, use the **Data** macros. If the data fits into the output buffer, then this macro copies it to the buffer. If it does not fit, however, the **Data** macro calls the **_XSend**, which first transmits the contents of the buffer and then transmits your data. The **Data** macro takes three arguments: the display, a pointer to the beginning of the data, and the number of bytes to be sent.

- Data (*display,* (char*) *data, nbytes*);
- Data16 *(display,* (short*) *data, nbytes*);
- Data32 (*display, (*long*) data,nbytes*);

Data, Data16, and **Data32** are macros that can use their last argument more than once, so that argument should be a variable rather than an expression such as *nitems*sizeof(item)*. You should do that kind of computation in a separate statement before calling them. Use the appropriate macro when sending byte, short, or long data.

If the protocol request requires a reply, then call the procedure **_XSend** instead of the **Data** macro. **_XSend** takes the same arguments, but because it sends your data immediately instead of copying it into the output buffer (which would later be erased anyway by the following call on **_XReply**), it is faster.

The following is an example of the **Data** macro:

```
Data(display, (char *) data, nbytes);
```

If the protocol request has a reply, use the **_XReply** function after dealing with all the fixed and variable length arguments.

Synchronous Calling

To ease debugging, each routine should have a call to a routine immediately prior to returning to the user. This routine is called **SyncHandle()** and is generally implemented as a macro. If the synchronous mode is enabled with the **XSynchronize** function, the request is sent immediately. The library, however, waits until any generated error has been handled.

Allocating and Deallocating Memory

To support the possible re-entry of these routines, several conventions should be observed when allocating and deallocating memory. This is appropriate especially when the user does not know the size of the data that is being returned. (The standard C language library routines on many systems are not protected against signals or other multithreaded use.) The analogies to standard I/O library routines are defined as follows:

Xmalloc()
Replaces the **malloc()** routine.

Xfree()
Replaces the **free()** routine.

Xcalloc()
Replaces the **calloc()** routine.

These routines should be used in place of any calls made to the normal C language library routines. For example, if you need a single scratch buffer inside a critical section to pack and unpack data to and from wire protocol, the general memory allocators may be too expensive to use (particularly in output routines, which are performance critical). Use the **_XAllocScratch** function to return a scratch buffer. This storage must only be used inside the critical section of your stub.

Deriving the Correct Extension Opcode

When writing an extension stub routine map from the call to the proper major and minor opcodes, you can use the following suggested strategy:

1. Declare an array of pointers. The length of this array, **_NFILE** long (normally found in the **<stdio.h>** header file), is the number of file descriptors supported on the system of type **XExtCodes**. These descriptors should be initialized to the value of NULL.
2. When your stub is entered, your initialization test should use the display pointer to access the file descriptor and an index into the array. If the entry is the value of NULL, then this is the first time you are entering the routine for this display. Call your initialization routine and pass the display pointer.
3. Once in your initialization routine, call the **XInitExtension** function. If it succeeds, store the pointer returned into this array. Establish a close display handler to allow you to zero the entry. Perform any other initialization your extension requires. (For example, install event handlers.) Your initialization routine normally returns a pointer to the **XExtCodes** data structure for this extension, which you normally find in your array of pointers.
4. After the initialization routine, the stub routine can continue normally, since its major opcode is safely in the **XExtCodes** data structure.

Dynamically Loadable X Server Extensions

Certain server extensions can be loaded by the X server during runtime when the server initializes. This can be done in two ways:

1. If the user wants to load the extension every time the server starts, the extension's nickname and the path name of its loadable module must be present in the file **/usr/bin/X11/static_ext**.
2. If the user wants to selectively load the extension in each invocation of the X server, the command line flag **-x** and the extension's nickname can be used. In order to use this method, the extension's nickname and the path name of its loadable module must be present in the file **/usr/bin/X11/dynamic_ext**. Usage of the command line flag is shown below:

   ```
   X -x mbx
   ```

   This loads the Multi-Buffer Extension into the server. Multiple instances of the **-x** flag are allowed.

The format of both **static_ext** and **dynamic_ext** files is the same, it contains two fields, separated by white spaces. The first field is the extension's nickname used to identify the module to be loaded, the nickname cannot be longer than 27 characters; the second field is the path name of the extension's loadable module. The total length of one entry, that is, the two fields and white spaces, cannot be longer than 160 characters. The following is an example of the format:

```
mbx    /usr/lpp/X11/bin/loadMbx
```

The server keeps track of nicknames of the extensions loaded, therefore, the same extension is only loaded once, even if it has multiple entries in files **static_ext** and **dynamic_ext** and command line flag **-x**.

If the X server cannot find or load the specified extension module, it is considered a fatal error. The X server uses the **loadquery** subroutine to query the error messages when the load system call fails. It then executes **execerror** to display these error messages, and exits the process.

AIXwindows Nonrectangular Window Shape Extension

This extension provides arbitrary window and border shapes within the X11 protocol.

The restriction of rectangular windows within the X protocol can be a significant limitation. For example, many transient windows would like to display a drop shadow to give the illusion of 3 dimensions. Also, some user interface style guides call for buttons with rounded corners; the full simulation of a nonrectangular shape, particularly with respect to event distribution and cursor shape, is not possible within the Core X protocol. Finally, round clocks and nonrectangular icons are pleasant additions to the desktop.

This extension provides mechanisms for changing the visible shape of a window to an arbitrary, nonrectangular form. This extension supplements the existing semantics and does not replace them. In particular, clients that are unaware of the extension need to be able to cope with oddly-shaped windows. For example, window managers should still be able to negotiate screen real estate in rectangular pieces. Toward this end, any shape specified for a window is clipped by the bounding rectangle for the window as specified by the window's geometry in the core protocol. An expected convention would be that client programs expand their shape to fill the area offered by the window manager.

For additional information, refer to the following topics:

- Description
- Types
- C Language Binding

Description

Each window (even with no shapes specified) is defined by two regions: the bounding region and the clip region. The *bounding region* is the area of the parent window which the window will occupy (including border). The *clip region* is the subset of the bounding region which is available for subwindows and graphics. The area between the bounding region and the clip region is defined to be the border of the window.

A nonshaped window will have a bounding region that is a rectangle spanning the window (including its border); the clip region will be a rectangle filling the inside dimensions (not including the border). In this document, these areas are referred to as the default bounding region and the default clip region. For a window with inside size of *width* by *height* and border width **width**, the default bounding and clip regions are the rectangles (relative to the window origin):

bounding.x = **width**

bounding.y = **width**

bounding.width = **width** + 2 * **width**

bounding.height = **height** + 2 * **width**

clip.x = 0

clip.y = 0

clip.width = **width**

clip.height = **height**

This extension allows a client to modify either or both of the bounding or clip regions by specifying new regions that combine with the default regions. These new regions are called the client bounding region and the client clip region. They are specified relative to the origin of the window and are always defined by offsets relative to the window origin (that is, region adjustments are not required when the window is moved). Three mechanisms for specifying regions are provided: a list of rectangles, a bitmap, and an existing bounding or clip region from a window. This is modeled on the specification of regions in graphics contexts in the Core protocol and allows a variety of different uses of the extension.

When using an existing window shape as an operand in specifying a new shape, the client region is used, unless none has been set in which case the default region is used instead.

The effective bounding region of a window is defined as the intersection of the client bounding region with the default bounding region. Any portion of the client bounding region that is not included in the default bounding region is not included in the effective bounding region on the screen. This means that window managers (or other geometry managers) accustomed to dealing with rectangular client windows can constrain the client to a rectangular area of the screen.

Construction of the effective bounding region is dynamic; the client bounding region is not mutated to obtain the effective bounding region. If a client bounding region is specified that extends beyond the current default bounding region, and the window is later enlarged, the effective bounding region will be enlarged to include more of the client bounding region.

The effective clip region of a window is defined to be the intersection of the client clip region with both the default clip region and the client bounding region. Any portion of the client clip region which is not included in both the default clip region and the client bounding region will not be included in the effective clip region on the screen.

Construction of the effective clip region is dynamic; the client clip region is not mutated to obtain the effective clip region. If a client clip region is specified, which extends beyond the current default clip region, and the window or its bounding region is later enlarged, the effective clip region will be enlarged to include more of the client clip region if it is included in the effective bounding region.

The border of a window is defined to be the difference between the effective bounding region and the effective clip region. If this region is empty, no border is displayed. If this region is not empty, the border is filled using the border tile or border pixel of the window as specified in the Core protocol. Note that a window with a nonzero border width will never be able to draw beyond the default clip region of the window. Also note that a zero border width does not prevent a window from having a border, because the clip shape can still be made smaller than the bounding shape.

All output to the window, and visible regions of any subwindows, will be clipped to the effective clip region. The server must not retain window contents beyond the effective bounding region with backing store. The window's origin (for graphics operations, background tiling, and subwindow placement) is not affected by the existence of a bounding region or clip region.

Areas that are inside the default bounding region but outside the effective bounding region are not part of the window; these areas of the screen will be occupied by other windows. Input events that occur within the default bounding region but outside the effective bounding region will be delivered as if the window was not occluding the event position. Events which occur in a nonrectangular border of a window will be delivered to that window, just as for events which occur in a normal rectangular border.

An InputOnly window can have its bounding region set, but it is a **Match** error to attempt to set a clip region on an InputOnly window or to specify its clip region as a source to a request in this extension.

The server must accept changes to the clip region of a root window, but the server is permitted to ignore requested changes to the bounding region of a root window. If the server accepts bounding region changes, the contents of the screen outside the bounding region are implementation-dependent.

Types

The following types are used in the request and event definitions in subsequent sections:

```
SHAPE_KIND:
        {Bounding,
        Clip}
SHAPE_OP:
        {Set,
        Union,
        Intersect,
        Subtract,
        Invert}
```

Set Indicates that the region specified as an explicit source in the request is stored unaltered as the new destination client region.

Union Indicates that the source and destination regions are connected to produce the new destination client region.

Intersect
Indicates that the source and destination regions are intersected together to produce the new destination client region.

Subtract
Indicates that the source region is subtracted from the destination region to produce the new destination region.

Invert Indicates that the destination region is subtracted from the source region to produce the new destination region.

C Language Binding

The C Language routines provide direct access to the protocol and add no additional semantics.

All **XShape** Extension functions and procedures, and all manifest constants and macros, start with the string ″XShape″. All routines that have return type *Status* will return non-zero for *success* and zero for *failure*. Even if the **XShape** Extension is supported, the server may withdraw such facilities arbitrarily; in which case they will subsequently return zero.

The **XShape** Extension includes the following C Language routines:

- **XShapeQueryExtension**
- **XShapeQueryVersion**
- **XShapeCombineRegion**
- **XShapeCombineRectangles**
- **XShapeCombineMask**
- **XShapeCombineShape**
- **XShapeOffsetShape**
- **XShapeQueryExtents**
- **XShapeSelectInput**
- **XShapeInputSelected**
- **XShapeGetRectangles**

AIXwindows Screen Saver Extension

AIXwindows provides support for changing the image on a display screen after a user-defined period of inactivity to avoid burning the cathode ray tube phosphors. However, no interfaces are provided for the user to control the image that is drawn. This extension allows an external ``screen saver'' client to detect when the alternate image is to be displayed and to provide the graphics.

Current X server implementations typically provide at least one form of ``screen saver'' image. Historically, this has been a copy of the X logo drawn against the root background pattern. However, many users have asked for the mechanism to allow them to write screen saver programs that provide capabilities similar to those provided by other window systems. In particular, such users often wish to be able to display corporate logos, instructions on how to reactivate the screen, and automatic screen-locking utilities. This extension provides a means for writing such clients.

The core **SetScreenSaver** Protocol Request can be used to set the length of time without activity on any input devices after which the screen saver should ``activate'' and alter the image on the screen. This image periodically ``cycles'' to reduce the length of time that any particular pixel is illuminated. Finally, the screen saver is ``deactivated'' in response to activity on any of the input devices or particular X requests.

Screen saving is typically done by disabling video output to the display tube or by drawing a changing pattern onto the display. If the server chooses the latter approach, a window with a special identifier is created and mapped at the top of the stacking order where it remains until the screen saver deactivates. At this time, the window is unmapped and is not accessible to any client requests.

The server's default mechanism is referred to as the internal screen saver. An external screen saver client requires a means of determining the window identifier for the screen saver window and setting the attributes (that is, size, location, visual, colormap) to be used when the window is mapped. These requirements form the basis of this extension.

Assumptions

This extension exports the notion of a special screen saver window that is mapped above all other windows on a display. This window has the override-redirect attribute set so that it is not subject to manipulation by the window manager. Furthermore, the X identifier for the window is never returned by **QueryTree** protocol requests on the root window, so it is typically not visible to other clients.

Types

In addition to the common types described in the core protocol, the following type is used in the request and event definitions in subsequent sections.

Name	Value
SCREENSAVEREVENT	{ScreenSaverNotify, ScreenSaverCycle}

Errors

The Screen Saver extension adds no errors beyond the core protocol.

Requests

The Screen Saver extension adds the following requests:

```
ScreenSaverQueryVersion

   client-major-version: CARD8
   client-minor-version: CARD8
 ->
   server-major-version: CARD8
   server-minor-version: CARD8
```

This request allows the client and server to determine which version of the protocol should be used. The client sends the version that it prefers; if the server understands that version, it returns the same values and interprets subsequent requests for this extension according to the specified version. Otherwise, the server returns the closest version of the protocol that it can support and interprets subsequent requests according to that version. This document describes major version 1, minor version 0; the major and minor revision numbers should only be increased in response to incompatible and compatible changes, respectively.

```
ScreenSaverQueryInfo
   drawable: DRAWABLE
 ->
   saver-window: WINDOW
   state: {Disabled, Off, On}
   kind: {Blanked, Internal, External}
   til-or-since: CARD32
   idle: CARD32
   event-mask: SETofSCREENSAVEREVENT
   Errors: Drawable
```

This request returns information about the state of the screen saver on the screen associated with drawable. The saver-window is the XID type that is associated with the screen saver window. This window is not guaranteed to exist except when external screen saver is active. Although it is a child of the root, this window is not returned by **QueryTree** protocol requests on the root. Whenever this window is mapped, it is always above any of its siblings in the stacking order.

The state field specifies whether or not the screen saver is currently active and how the til-or-since value should be interpreted:

Off The screen is not currently being saved; til-or-since specifies the number of milliseconds until the screen saver is expected to activate.

On The screen is currently being saved; til-or-since specifies the number of milliseconds since the screen saver activated.

Disabled
The screen saver is currently disabled; til-or-since is zero.

The `kind` field specifies the mechanism that either is currently being used or would have been were the screen being saved:

Blanked
The video signal to the display monitor was disabled.

Internal
A server-dependent, built-in screen saver image was displayed; either no client had set the screen saver window attributes or a different client had the server grabbed when the screen saver activated.

External
The screen saver window was mapped with attributes set by a client using the ScreenSaverSetAttributes request.

The `idle` field specifies the number of milliseconds since the last input was received from the user on any of the input devices.

The `event-mask` field specifies which, if any, screen saver events this client has requested using the **ScreenSaverSelectInput** function.

If drawable is not a valid drawable identifier, a **Drawable** error is returned and the request is ignored.

```
ScreenSaverSelectInput

   drawable: DRAWABLE
   event-mask: SETofSCREENSAVEREVENT

   Errors: Drawable, Match
```

This request specifies which Screen Saver extension events on the screen associated with drawable should be generated for this client. If no bits are set in the `eventmask` field, then no events are generated. Otherwise, any combination of the following bits may be set:

ScreenSaverNotify
If this bit is set, **ScreenSaverNotify** events are generated whenever the screen saver is activated or deactivated.

ScreenSaverCycle
If this bit is set, **ScreenSaverNotify** events are generated whenever the screen saver cycle interval passes.

If drawable is not a valid drawable identifier, a **Drawable** error is returned. If any undefined bits are set in `event-mask` field, a **Value** error is returned. If an error is returned, the request is ignored.

```
ScreenSaverSetAttributes

   drawable: DRAWABLE
   class: {InputOutput, InputOnly, CopyFromParent}
   depth: CARD8
   visual: VISUALID or CopyFromParent
   x, y: INT16
   width, height, border-width: CARD16
   value-mask: BITMASK
```

```
   value-list: LISTofVALUE

   Errors: Access, Window, Pixmap, Colormap, Cursor,  Match,
   Value, Alloc
```

This request sets the attributes that this client would like to see used in creating the screen saver window on the screen associated with drawable. If another client currently has the attributes set, an Access error is generated and the request is ignored.

Otherwise, the specified window attributes are checked as if they were used in a core **CreateWindow** protocol request whose parent is the root. The `overrideredirect` field is ignored as it is implicitly set to a value of True. If the window attributes result in an error according to the rules for the **CreateWindow** protocol request, the request is ignored.

Otherwise, the attributes are stored and take effect on the next activation that occurs when the server is not grabbed by another client. Any resources specified for the **background-pixmap** or **cursor** attributes may be freed immediately. The server is free to copy the **background-pixmap** or **cursor** resources or to use them in place; therefore, the effect of changing the contents of those resources is undefined. If the specified colormap no longer exists when the screen saver activates, the parent's colormap is used instead. If no errors are generated by this request, any previous screen saver window attributes set by this client are released.

When the screen saver next activates and the server is not grabbed by another client, the screen saver window is created, if necessary, and set to the specified attributes and events are generated as usual. The colormap associated with the screen saver window is installed. Finally, the screen saver window is mapped.

The window remains mapped and at the top of the stacking order until the screen saver is deactivated in response to activity on any of the user input devices, a **ForceScreenSaver** protocol request with a value of Reset, or any request that would cause the window to be unmapped.

If the screen saver activates while the server is grabbed by another client, the internal saver mechanism is used. The **ForceScreenSaver** request may be used with a value of Active to deactivate the internal saver and activate the external saver.

If the screen saver client's connection to the server is broken while the screen saver is activated and the client's close down mode has not been Retain Permanent or Retain Temporary, the current screen saver is deactivated and the internal screen saver is immediately activated.

When the screen saver deactivates, the screen saver window's colormap is uninstalled and the window is unmapped (except as described below). The screen saver XID is disassociated with the window and the server may, but is not required to, destroy the window along with any children. When the screen saver is being deactivated and then immediately reactivated (such as when switching screen savers), the server may leave the screen saver window mapped (typically to avoid generating exposures).

```
ScreenSaverUnsetAttributes
   drawable: DRAWABLE
   Errors: Drawable
```

This request notifies the server that this client no longer wishes to control the screen saver window. Any screen saver attributes set by this client and any descendents of the screen saver window created by this client should be released immediately if the screen saver is not active, else upon deactivation.

This request is ignored if the client has not previously set the screen saver window attributes.

Events

The Screen Saver extension adds one event:

```
ScreenSaverNotify

   root: WINDOW
   window: WINDOW
   state: {Off, On, Cycle}
   kind: {Blanked, Internal, External}
   forced: BOOL
   time: TIMESTAMP
```

This event is delivered to clients that have requested **ScreenSaverNotify** events using the **ScreenSaverSelectInput** protocol request whenever the screen saver activates or deactivates.

The `root` field specifies root window of the screen for which the event was generated. The `window` field specifies the value that is returned by the **ScreenSaverQueryInfo** function as the identifier for the screen saver window. This window is not required to exist if the external screen saver is not active.

The `state` field specifies the cause of the event:

Off The screen saver deactivated; this event is sent if the client has set the ScreenSaverNotify bit in its event mask.

On The screen saver activated. This event is sent if the client has set the ScreenSaverNotify bit in its event mask.

Cycle The cycle interval passed and the client is expected to change the image on the screen. This event is sent if the client has set the ScreenSaverCycle bit in its event mask.

If the `state` field is set to `On` or `Off` then forced indicates whether or not activation or deactivation was caused by a core **ForceScreenSaver** protocol request; otherwise, forced is set to a value of False. The `kind` field specifies mechanism that was used to save the screen when the screen saver was activated, as described in the **ScreenSaverQueryInfo** function.

The `time` field indicates the server time when the event was generated.

Common Types

```
SETofSCREENSAVEREVENT
       #x00000001ScreenSaverNotifyMask
       #x00000002ScreenSaverCycleMask
```

Requests

```
ScreenSaverQueryVersion
   1    CARD8               screen saver opcode
   1    0                   minor opcode
   2    2                   request length
   1    CARD8               client major version
   1    CARD8               client minor version
   2                        unused

 ->
   1    1                   Reply
   1                        unused
   2    CARD16              sequence number
   4    0                   reply length
   1    CARD8               server major version
   1    CARD8               server minor version
   22                       unused
ScreenSaverQueryInfo
   1    CARD8               screen saver opcode
   1    1                   minor opcode
   2    2                   request length
   4    DRAWABLE            drawable associated with screen
```

```
->
  1    1                  Reply
  1    CARD8              state
       0         Off
       1         On
       3         Disabled
  2    CARD16             sequence number
  4    0                  reply length
  4    WINDOW             saver window
  4    CARD32             milliseconds until saver or since saver
  4    CARD32             milliseconds since last user device input
  4    SETofSCREENSAVEREVENTevent mask
  1    CARD8              kind
       0         Blanked
       1         Internal
       2         External
  10                      unused
```

```
ScreenSaverSelectInput
  1    CARD8              screen saver opcode
  1    2                  minor opcode
  2    3                  request length
  4    DRAWABLE           drawable associated with screen
  4    SETofSCREENSAVEREVENTevent mask
```

```
ScreenSaverSetAttributes
  1    CARD8              screen saver opcode
  1    3                  minor opcode
  2    6+n                request length
  4    DRAWABLE           drawable associated with screen
  2    INT16              x
  2    INT16              y
  2    CARD16             width
  2    CARD16             height
  2    CARD16             border-width
  1                       class
       0         CopyFromParent
       1         InputOutput
       2         InputOnly
  1    CARD8              depth
  4    VISUALID           visual
       0         CopyFromParent
  4    BITMASK            value-mask (has n bits set to 1)
       encodings are the same as for core CreateWindow
  4n   LISTofVALUE        value-list
       encodings are the same as for core CreateWindow
```

```
ScreenSaverUnsetAttributes
  1    CARD8              screen saver opcode
  1    4                  minor opcode
  2    3                  request length
  4    DRAWABLE           drawable associated with screen
```

Events

```
ScreenSaverNotify
  1    CARD8              code assigned by core
  1    CARD8              state
       0         Off
       1         On
       2         Cycle
  2    CARD16             sequence number
  4    TIMESTAMP          time
  4    WINDOW             root
  4    WINDOW             screen saver window
  1    CARD8              kind
       0         Blanked
```

```
      1        Internal
      2        External
   1    BOOL              forced
   14                     unused
```

Inter-Client Communications Conventions

Screen saver clients should create at least one resource value whose identifier can be stored in a property named **_SCREEN_SAVER_ID** on the root of each screen it is managing. This property should have one 32-bit value corresponding to the resource identifier; the type of the property should indicate the type of the resource and should be one of the following: **WINDOW**, **PIXMAP**, **CURSOR**, **FONT**, **COLORMAP**.

C Language Binding

The C binding for this extension simply provide access to the protocol; they add no semantics beyond what is described above.

The include file for this extension is `X11/extensions/scrnsaver.h`.

```
Bool
XScreenSaverQueryExtension (Display,EventBase,ErrorBase)
     Display *display;
     int *event_base; /* RETURN */
     int *error_base; /* RETURN */
```

This routine returns a value of True if the specified display supports the SCREEN-SAVER extension; otherwise it returns False value. If the extension is supported, the event number for **ScreenSaverNotify** event functions is returned in the value pointed to by the *EventBase* parameter. Since no additional errors are defined by this extension, the results of the *ErrorBase* parameter are not defined.

```
Status
XScreenSaverQueryVersion (Display, Major, Minor)
     Display *display;
     int *major; /* RETURN */
     int *minor; /* RETURN */
```

If the specified display supports the extension, the version numbers of the protocol expected by the server are returned in *Major* and *Minor* parameters and a nonzero value is returned. Otherwise, the arguments are not set and 0 (zero) is returned.

```
XScreenSaverInfo *
XScreenSaverAllocInfo ()
```

This routine allocates and returns an **XScreenSaverInfo** structure for use in calls to the **XScreenSaverQueryInfo** Function. All fields in the structure are initialized to zero. If insufficient memory is available, a Null value is returned. The results of this routine can be released using the **XFree** function.

```
Status
XScreenSaverQueryInfo (display, drawable, saver_info)
     Display *display;
     Drawable drawable;
     XScreenSaverInfo *saver_info; /* RETURN */
```

If the specified display supports the extension, information about the current state of the screen server is returned in saver_info and a **non-zero** value is returned. The **XScreenSaverInfo** structure is defined as follows:

```
typedef struct {
   Window window;            /* screen saver window */
   int state;                /*
ScreenSaver{Off,On,Disabled} */
   int              /* ScreenSaver{Blanked,Internal,External} */
```

```
   unsigned long til_or_since; /*  milliseconds */
   unsigned long idle;         /*  milliseconds */
   unsigned long event_mask;   /* events */
} XScreenSaverInfo;
```

See the **ScreenSaverQueryInfo** protocol request for a description of the fields. If the extension is not supported, the **saver_info** structure variable is not changed and 0 is returned.

```
void
XScreenSaverSelectInput (display, drawable, event_mask)
     Display *display;
     Drawable drawable;
     unsigned long event_mask;
```

If the specified display supports the extension, this routine asks that events related to the screen saver be generated for this client. The format of the events generated is:

```
typedef struct {
    int type;               /* of event */
    unsigned long serial;   /* # of last request */
                            /* processed by server */
    Bool send_event;        /* true if this came from a */
                            /* SendEvent request */
    Display *display;       /* Display the event was read from */
    Window window;          /* screen saver window */
    Window  root;           /*  root window of event screen */
    int state;              /* ScreenSaver{Off,On,Cycle}   */
    int kind;               /*
                        ScreenSaver{Blanked,Internal,External} */
    Bool  forced;           /* extents of new region */
    Time time;              /* event timestamp */
           } XScreenSaverNotifyEvent;
```

See the definition of the **ScreenSaverSelectInput** protocol request for descriptions of the allowed event masks.

```
void
XScreenSaverSetAttributes (display, drawable, x, y, width, height, border_width,
depth, class, visual,
                 valuemask, attributes)
     Display *dpy;
     Drawable drawable;
     int x;
     int y;
     unsigned int width;
     unsigned int height;
     unsigned int border_width;
     int depth;
     unsigned int class;
     Visual *visual;
     unsigned long valuemask;
     XSetWindowAttributes *attributes;
```

If the specified display supports the extension, this routine sets the attributes to be used the next time the external screen saver is activated. See the definition of the **ScreenSaverSetAttributes** protocol request for a description of each of the arguments.

```
void
XScreenSaverUnsetAttributes (display, drawable)
     Display *display;
     Drawable drawable;
```

If the specified display supports the extension, this routine instructs the server to discard any previous screen saver window attributes set by this client.

```
Status
XScreenSaverRegister (display, screen, xid, type)
     Display *display;
     int screen;
     XID xid;
     Atom type;
```

This routine stores the given XID in the **_SCREEN_SAVER_ID** property (of the given type) on the root window of the specified screen. It returns 0 (zero) if an error is encountered and the property is not changed, otherwise it returns nonzero.

```
Status
XScreenSaverUnregister (display, screen)
     Display *display;
     int screen;
```

This routine removes any **_SCREEN_SAVER_ID** property from the root window of the specified screen. It returns 0 (zero) if an error is encountered and the property is changed, otherwise it returns nonzero.

```
Status
XScreenSaverGetRegistered (display, screen, xid, type)
     Display *display;
     int screen;
     XID *xid; /* RETURN */
     Atom *type; /* RETURN */
```

This routine returns the XID and type stored in the **_SCREEN_SAVER_ID** property on the root window of the specified screen. It returns 0 (zero) if an error is encountered or if the property does not exist or is not of the correct format; otherwise it returns nonzero value.

Using AIXwindows Input Device Functions

You can use the Xlib input device functions to accomplish the following:

- Grab the pointer and individual buttons on the pointer.
- Grab the keyboard and individual keys on the keyboard.
- Move the pointer.
- Set the input focus.
- Manipulate the keyboard and pointer settings.
- Manipulate the keyboard encoding.

Pointer Grabbing

Xlib provides functions that you can use to control input from the pointer, which usually is a mouse. Usually, as soon as keyboard and mouse events occur, the X server delivers them to the appropriate client, which is determined by the window and input focus. The X server provides sufficient control over event delivery to allow window managers to support mouse ahead and various other styles of user interface. Many of these user interfaces depend upon synchronous delivery of events. The delivery of pointer and keyboard events can be controlled independently.

When mouse buttons or keyboard keys are grabbed, events will be sent to the grabbing client rather than the normal client who would have received the event. If the keyboard or pointer is in asynchronous mode, further mouse and keyboard events will continue to be processed. If the keyboard or pointer is in synchronous mode, no further events are processed until the grabbing client allows them (see **XAllowEvents**). The keyboard or pointer is considered frozen during this interval. The event that triggered the grab can also be replayed.

Note: The logical state of a device (as seen by client applications) may lag the physical state if device event processing is frozen.

There are two kinds of grabs: active and passive. An active grab occurs when a single client grabs the keyboard and/or pointer explicitly (see the **XGrabPointer** and **XGrabKeyboard** functions.) A passive grab occurs when clients grab a particular keyboard key or pointer button in a window, and the grab will activate when the key or button is actually pressed. Passive grabs are convenient for implementing reliable pop-up menus. For example, you can guarantee that the pop-up is mapped before the up pointer button event occurs by grabbing a button requesting synchronous behavior. The down event will trigger the grab and freeze further processing of pointer events until you have the chance to map the pop-up window. You can then allow further event processing. The up event will then be correctly processed relative to the pop-up window.

For many operations, there are functions that take a time argument. The X server includes a time stamp in various events. One special time, called *CurrentTime*, represents the current server time. The X server maintains the time when the input focus was last changed, when the keyboard was last grabbed, when the pointer was last grabbed, or when a selection was last changed. Your application may be slow reacting to an event. You often need some way to specify that your request should not occur if another application has in the meanwhile taken control of the keyboard, pointer, or selection. By providing the time stamp from the event in the request, you can arrange that the operation not take effect if someone else has performed an operation in the meanwhile.

A *time stamp* is a time value, expressed in milliseconds. It typically is the time since the last server reset. Time stamp values wrap around (after about 49.7 days). The server, given its current time is represented by time stamp T, always interprets time stamps from clients by treating half of the time stamp space as being later in time than T. One time stamp value, named *CurrentTime*, is never generated by the server. This value is reserved for use in requests to represent the current server time.

The following functions are used for pointer grabbing:

Function	Description
XGrabPointer	Grabs control of the pointer.
XUngrabPointer	Releases the pointer.
XChangeActivePointer Grab	Changes the specified dynamic parameters.
XGrabButton	Grabs a pointer button.
XUngrabButton	Releases a pointer button.

For many of the functions used for pointer grabbing, you pass pointer event mask bits. The following are valid pointer event mask bits:

ButtonPressMask

ButtonReleaseMask

EnterWindowMask

LeaveWindowMask

PointerMotionMask

PointerMotionHintMask

Button1MotionMask

Button2MotionMask

Button3MotionMask

Button4MotionMask

Button5MotionMask

ButtonMotionMask

KeyMapStateMask

For other functions used for pointer grabbing, you pass keymask bits. The following are valid keymask bits:

LockMask

ControlMask

Mod1Mask

Mod2Mask

Mod4Mask

Mod5Mask

Keyboard Grabbing

Xlib provides the following functions that you can use to grab or ungrab the keyboard and to allow events:

Function	Description
XGrabKeyboard	Grabs the keyboard.
XUngrabKeyboard	Releases the keyboard.
XGrabKey	Passively grabs a single key of the keyboard.
XUngrabKey	Releases the key combination of the specified window.

For many functions used in keyboard grabbing, you pass keymask bits. The following are valid keymask bits:

ShiftMask

LockMask

ControlMask

Mod1Mask

Mod2Mask

Mod3Mask

Mod4Mask

Mod5Mask

Resuming Event Processing

Use the **XAllowEvents** function to resume event processing. This function allows the processing of further events when the device has been frozen.

Moving the Pointer

Although movement of the pointer normally should be left to the control of the end user, sometimes it is necessary to move the pointer to a new position under program control. Use the **XWarpPointer** function to move the pointer to an arbitrary point in a window.

Controlling Input Focus

Xlib provides functions that you can use to set and get the input focus. The input focus is a shared resource, and cooperation among clients is required for correct interaction. See the *Inter-Client Communication Conventions Manual (ICCCM)* for input focus policy. Use the following functions to control input focus:

Function	Description
XSetInputFocus	Changes the input focus and the last-focus-change time.

Function	Description
XGetInputFocus	Obtains the current input focus.

Keyboard and Pointer Settings

Xlib provides functions that you can use to change the keyboard control, obtain a list of the auto-repeat keys, turn keyboard auto-repeat on or off, ring the bell, set or obtain the pointer button or keyboard mapping, and obtain a bit vector for the keyboard.

This section discusses the user-preference options of bell, key click, pointer behavior, and so on. The default values for many of these functions are determined by command line arguments to the X server and, on POSIX-conformant systems, are typically set in the **/etc/ttys** file. Not all implementations will actually be able to control all of these parameters.

Use the following functions to control keyboard and pointer settings:

Function	Description
XChangeKeyboardControl	Changes control of a keyboard.
XGetKeyboardControl	Obtains the current control values for the keyboard.
XAutoRepeatOn	Turns on keyboard auto-repeat.
XAutoRepeatOff	Turns off auto-repeat.
XBell	Rings a bell.
XQueryKeymap	Obtains a bit vector that describes the state of the keyboard.
XSetPointerMapping	Sets the mapping of the pointer buttons.
XGetPointerMapping	Gets the pointer mapping.
XChangePointerControl	Controls the interactive feel of the pointer.
XGetPointerControl	Gets the current pointer parameters.

The **XChangeKeyboardControl** function operates on a XKeyboardControl structure:

```
/* Mask bits for ChangeKeyboardControl */
#define         KBKeyClickPercent       (1L<<0)
#define         KBBellPercent           (1L<<1)
#define         KBBellPitch             (1L<<2)
#define         KBBellDuration          (1L<<3)
#define         KBLed                   (1L<<4)
#define         KBLedMode               (1L<<5)
#define         KBKey                   (1L<<6)
#define         KBAutoRepeatMode        (1L<<7)
/* Values */
typedef struct {
        int key_click_percent;
        int bell_percent;
        int bell_pitch;
        int bell_duration;
        int led;
        int led_mode;            /* LedModeOn, LedModeOff */
        int key;
        int auto_repeat_mode;    /* AutoRepeatModeOff,
                                    AutoRepeatModeOn,
                                    AutoRepeatModeDefault */
} XKeyboardControl;
```

The key_click_percent member sets the volume for key clicks between 0 (off) and 100 (loud) inclusive, if possible. A setting of \-1 restores the default. Other negative values generate a **BadValue** error.

The bell_percent sets the base volume for the bell between 0 (off) and 100 (loud) inclusive, if possible. A setting of \-1 restores the default. Other negative values generate a **BadValue** error. The bell_pitch member sets the pitch (specified in Hz) of the bell, if possible. A setting of \-1 restores the default. Other

negative values generate a **BadValue** error. The bell_duration member sets the duration of the bell specified in milliseconds, if possible. A setting of \-1 restores the default. Other negative values generate a **BadValue** error.

If both the led_mode and led members are specified, the state of that LED is changed, if possible. The led_mode member can be set to **LedModeOn** or **LedModeOff**. If only led_mode is specified, the state of all LEDs is changed, if possible. At most, 32 LEDs numbered from one are supported. No standard interpretation of LEDs is defined. If led is specified without led_mode, a **BadMatch** error results.

If both the auto_repeat_mode and key members are specified, the auto_repeat_mode of that key is changed (according to **AutoRepeatModeOn**, **AutoRepeatModeOff**, or AutoRepeatModeDefault), if possible. If only auto_repeat_mode is specified, the global auto_repeat_mode for the entire keyboard is changed, if possible, and does not affect the per key settings. If a key is specified without an auto_repeat_mode, a **BadMatch** error results. Each key has an individual mode of whether or not it should auto-repeat and a default setting for the mode. In addition, there is a global mode of whether auto-repeat should be enabled or not and a default setting for that mode. When global mode is **AutoRepeatModeOn**, keys should obey their individual auto-repeat modes. When global mode is **AutoRepeatModeOff**, no keys should auto-repeat. An auto-repeating key generates alternating **KeyPress** and **KeyRelease** events. When a key is used as a modifier, it is desirable for the key not to auto-repeat, regardless of its auto-repeat setting.

A bell generator connected with the console but not directly on a keyboard is treated as if it were part of the keyboard. The order in which controls are verified and altered is server-dependent. If an error is generated, a subset of the controls may have been altered.

Keyboard Encoding

A KeyCode represents a physical (or logical) key. KeyCodes lie in the inclusive range [8,255]. A KeyCode value carries no intrinsic information, although server implementors may attempt to encode geometry (for example, matrix) information in some fashion so that it can be interpreted in a server-dependent fashion. The mapping between keys and KeyCodes cannot be changed.

A KeySym is an encoding of a symbol on the cap of a key. The set of defined KeySyms includes the ISO Latin character sets (1\-4), Katakana, Arabic, Cyrillic, Greek, Technical, Special, Publishing, APL, Hebrew, and a special miscellany of keys found on keyboards (Return, Help, Tab, and so on). To the extent possible, these sets are derived from international standards. In areas where no standards exist, some of these sets are derived from Digital Equipment Corporation standards. The list of defined symbols can be found in<**X11/keysymdef.h**>. Unfortunately, some C preprocessors have limits on the number of defined symbols. If you must use KeySyms not in the Latin 1\-4, Greek, and miscellaneous classes, you may have to define a symbol for those sets. Most applications usually only include <**X11/keysym.h**>, which defines symbols for ISO Latin 1\-4, Greek, and miscellaneous.

A list of KeySyms is associated with each KeyCode. The list is intended to convey the set of symbols on the corresponding key. If the list (ignoring trailing NoSymbol entries) is a single KeySym `K`, then the list is treated as if it were the list `K NoSymbol K NoSymbol`. If the list (ignoring trailing NoSymbol entries) is a pair of KeySyms `K1 K2`, then the list is treated as if it were the list `K1 K2 K1 K2`. If the list (ignoring trailing NoSymbol entries) is a triple of KeySyms `K1 K2 K3`, then the list is treated as if it were the list `K1 K2 K3 NoSymbol`. When an explicit void element is desired in the list, the value VoidSymbol can be used.

The first four elements of the list are split into two groups of KeySyms. Group 1 contains the first and second KeySyms; Group 2 contains the third and fourth KeySyms. Within each group, if the second element of the group is NoSymbol, then the group should be treated as if the second element were the same as the first element, except when the first element is an alphabetic KeySym ″K″ for which both lowercase and uppercase forms are defined. In that case, the group should be treated as if the first element were the lowercase form of K and the second element were the uppercase form of K.

The standard rules for obtaining a KeySym from a **KeyPress** event make use of only the Group 1 and Group 2 KeySyms; no interpretation of other KeySyms in the list is given. Which group to use is determined by the modifier state. Switching between groups is controlled by the KeySym named MODE SWITCH, by attaching that KeySym to some KeyCode and attaching that KeyCode to any one of the modifiers Mod1 through Mod5. This modifier is called the *group modifier*. For any KeyCode, Group 1 is used when the group modifier is off, and Group 2 is used when the group modifier is on.

Within a group, the modifier state also determines which KeySym to use. The first KeySym is used when the Shift and Lock modifiers are off. The second KeySym is used when the Shift modifier is on, when the Lock modifier is on and the second KeySym is uppercase alphabetic, or when the Lock modifier is on and is interpreted as ShiftLock. Otherwise, when the Lock modifier is on and is interpreted as CapsLock, the state of the Shift modifier is applied first to select a KeySym; but if that KeySym is lowercase alphabetic, then the corresponding uppercase KeySym is used instead.

No spatial geometry of the symbols on the key is defined by their order in the KeySym list, although a geometry might be defined on a vendor-specific basis. The X server does not use the mapping between KeyCodes and KeySyms. Rather, it stores it merely for reading and writing by clients.

The KeyMask modifier named Lock is intended to be mapped to either a CapsLock or a ShiftLock key, but which one is left as application-specific and/or user-specific. However, it is suggested that the determination be made according to the associated KeySym(s) of the corresponding KeyCode.

The following functions are used for keyboard encoding:

Function	Description
XChangeKeyboardMapping	Defines the symbols for the keyboard mapping.
XDeleteModifiermapEntry	Deletes an entry from an **XModifierKeymap** structure.
XDisplayKeycodes	Returns the legal KeyCodes for a display.
XFreeModifiermap	Destroys an **XModifierKeymap** structure.
XGetKeyboardMapping	Returns the symbols for the specified KeyCodes.
XGetModifierMapping	Returns the KeyCodes used as modifiers.
XInsertModifiermapEntry	Adds a new entry to an **XModifierKeymap** structure.
XNewModifiermap	Creates an **XModifierKeymap** structure.
XSetModifierMapping	Sets the KeyCodes to be used as modifiers.

AIXwindows Input Extension Library

Purpose of the Extension

This document describes an extension to the X11 server. The purpose of this extension is to support the use of additional input devices beyond the pointer and keyboard devices defined by the core X protocol. This first section gives an overview of the input extension. The following sections correspond to Chapters 7 and 8, ″Window Manager Functions″ and ″Events and Event-Handling Functions″ of the *Xlib - C Language Interface* manual and describe how to use the input extension.

Design Approach

The design approach of the extension is to define functions and events analogous to the core functions and events. This allows extension input devices and events to be individually distinguishable from each other and from the core input devices and events. These functions and events make use of a device identifier and support the reporting of *n*-dimensional motion data as well as other data that is not currently reportable via the core input events.

Core Input Devices

The X server core protocol supports two input devices: a pointer and a keyboard. The pointer device has two major functions. First, it may be used to generate motion information that client programs can detect. Second, it may also be used to indicate the current location and focus of the X keyboard. To accomplish this, the server echoes a cursor at the current position of the X pointer. Unless the X keyboard has been explicitly focused, this cursor also shows the current location and focus of the X keyboard.

The X keyboard is used to generate input that client programs can detect.

The X keyboard and X pointer are referred to in this document as the *core devices*, and the input events they generate (**KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, and **MotionNotify**) are known as the *core input events*. All other input devices are referred to as *extension input devices* and the input events they generate are referred to as *extension input events*.

Note: This input extension does not change the behavior or functionality of the core input devices, core events, or core protocol requests, with the exception of the core grab requests. These requests may affect the synchronization of events from extension devices. See the explanation in the section titled Event Synchronization and Core Grabs .

Selection of the physical devices to be initially used by the server as the core devices is left implementation dependent. Functions are defined that allow client programs to change which physical devices are used as the core devices.

Extension Input Devices

The input extension controls access to input devices other than the X keyboard and X pointer. It allows client programs to select input from these devices independently from each other and independently from the core devices. Input events from these devices are of extension types (**DeviceKeyPress**, **DeviceKeyRelease**, **DeviceButtonPress**, **DeviceButtonRelease**, **DeviceMotionNotify**, and so on) and contain a device identifier so that events of the same type coming from different input devices can be distinguished.

Extension input events are not limited in size by the size of the server 32-byte wire events. Extension input events may be constructed by the server sending as many wire sized events as necessary to return the information required for that event. The library event reformatting routines are responsible for combining these into one or more client XEvents.

Any input device that generates key, button or motion data may be used as an extension input device. Extension input devices may have zero or more keys, zero or more buttons, and may report zero or more axes of motion. Motion may be reported as relative movements from a previous position or as an absolute position. All valuators reporting motion information for a given extension input device must report the same kind of motion information (absolute or relative).

This extension is designed to accommodate new types of input devices that may be added in the future. The protocol requests that refer to specific characteristics of input devices organize that information by **input device classes**. Server implementors may add new classes of input devices without changing the protocol requests.

All extension input devices are treated like the core X keyboard in determining their location and focus. The server does not track the location of these devices on an individual basis, and therefore does not echo a cursor to indicate their current location. Instead, their location is determined by the location of the core X pointer. Like the core X keyboard, some may be explicitly focused. If they are not explicitly focused, their focus is determined by the location of the core X pointer.

Input Device Classes

Some of the input extension requests divide input devices into classes based on their functionality. This is intended to allow new classes of input devices to be defined at a later time without changing the semantics of these functions. The following input device classes are currently defined:

Class	Description
KEY	The device reports key events.
BUTTON	The device reports button events.
VALUATOR	The device reports valuator data in motion events.
PROXIMITY	The device reports proximity events.
FOCUS	The device can be focused.
FEEDBACK	The device supports feedbacks.

Additional classes may be added in the future. Functions that support multiple input classes, such as the **XListInputDevices** function that lists all available input devices, organize the data they return by input class. Client programs that use these functions should not access data unless it matches a class defined at the time those clients were compiled. In this way, new classes can be added without forcing existing clients that use these functions to be recompiled.

Using Extension Input Devices

A client that wishes to access an input device does so through the library functions defined in the following sections. A typical sequence of requests that a client would make is as follows:

- **XListInputDevices** - List all of the available input devices. From the information returned by this request, determine whether the desired input device is attached to the server. For a description of the **XListInputDevices** request, see the section entitled Listing Available Devices .
- **XOpenDevice** - Request that the server open the device for access by this client. This request returns an **XDevice** structure that is used by most other input extension requests to identify the specified device. For a description of the **XOpenDevice** request, see the section entitled Enabling and Disabling Extension Devices.
- Determine the event types and event classes needed to select the desired input extension events, and identify them when they are received. This is done via macros whose name corresponds to the desired event, that is, **DeviceKeyPress**. For a description of these macros, see the section entitled Selecting Extension Device Events.
- **XSelectExtensionEvent** - Select the desired events from the server. For a description of the **XSelextExtensionEvent** request, see the section entitled Selecting Extension Device Events .
- **XNextEvent** - Receive the next available event. This is the core **XNextEvent** function provided by the standard X library.

Other requests are defined to grab and focus extension devices, to change their key, button, or modifier mappings, to control the propagation of input extension events, to get motion history from an extension device, and to send input extension events to another client. These functions are described in the following sections.

Library Extension Requests

Extension input devices are accessed by client programs through the use of new protocol requests. The following requests are provided as extensions to Xlib. Constants and structures referenced by these functions may be found in the files **XI.h** and **XInput.h**, which are attached to this document as appendix A.

The library returns **NoSuchExtension** if an extension request is made to a server that does not support the input extension. Input extension requests cannot be used to access the X keyboard and X pointer devices.

Window Manager Functions

Changing the Core Devices

These functions are provided to change which physical device is used as the X pointer or X keyboard.

Note: Using these functions may change the characteristics of the core devices. The new pointer device may have a different number of buttons than the old one did, or the new keyboard device may have a different number of keys or report a different range of keycodes. Client programs may be running that depend on those characteristics. For example, a client program could allocate an array based on the number of buttons on the pointer device, and then use the button numbers received in button events as indices into that array. Changing the core devices could cause such client programs to behave improperly or abnormally terminate, if they ignore the **ChangeDeviceNotify** event generated by these requests.

These functions change the X keyboard or X pointer device and generate an **XChangeDeviceNotify** event and a **MappingNotify** event. The specified device becomes the new X keyboard or X pointer device. The location of the core device does not change as a result of this request.

These requests fail and return **AlreadyGrabbed** if either the specified device or the core device it would replace are grabbed by some other client. They fail and return **GrabFrozen** if either device is frozen by the active grab of another client.

These requests fail with a **BadDevice** error if the specified device is invalid, has not previously been opened via an **XOpenDevice** request, or is not supported as a core device by the server implementation.

Once the device has successfully replaced one of the core devices, it is treated as a core device until it is in turn replaced by another **ChangeDevice** request, or until the server terminates. The termination of the client that changed the device does not cause it to change back. Attempts to use the **XCloseDevice** request to close the new core device fails with a **BadDevice** error.

The **XChangeKeyboardDevice** function is used to change which physical device is used as the X keyboard.

The **XChangePointerDevice** function is used to change which physical device is used as the X pointer.

Event Synchronization and Core Grabs

Implementation of the input extension requires an extension of the meaning of event synchronization for the core grab requests. This is necessary in order to allow window managers to freeze all input devices with a single request.

The core grab requests require a *PointerMode* and *KeyboardMode* argument. The meaning of these modes is changed by the input extension. For the **XGrabPointer** and **XGrabButton** requests, *PointerMode* controls synchronization of the pointer device, and *KeyboardMode* controls the synchronization of all other input devices. For the **XGrabKeyboard** and **XGrabKey** requests, *PointerMode* controls the synchronization of all input devices except the X keyboard, while *KeyboardMode* controls the synchronization of the keyboard. When using one of the core grab requests, the synchronization of extension devices is controlled by the mode specified for the device not being grabbed.

Extension Active Grabs

Active grabs of extension devices are supported through the **XGrabDevice** function in the same way that core devices are grabbed using the core **XGrabKeyboard** function, except that a device is passed as a function parameter. The **XUngrabDevice** function allows a previous active grab for an extension device to be released.

Passive Grabs of Buttons and Keys

Passive grabs of buttons and keys on extension devices are supported through the **XGrabDeviceButton** and **XGrabDeviceKey** functions. These passive grabs are released through the **XUngrabDeviceKey** and **XUngrabDeviceButton** functions.

Thawing a Device

The **XAllowDeviceEvents** function allows further events to be processed when a device has been frozen.

Controlling Device Focus

The current focus window for an extension input device can be determined using the **XGetDeviceFocus** function. Extension devices are focused using the **XSetDeviceFocus** function in the same way that the keyboard is focused using the core **XSetInputFocus** function, except that a device ID is passed as a function parameter. One additional focus state, **FollowKeyboard**, is provided for extension devices.

Controlling Device Feedback

The current feedback settings of an extension input device are determined using the **XGetFeedbackControl** function. The **XFreeFeedbackList** function frees the list of feedback control information returned by the **XGetFeedbackControl** function.

The settings of a feedback on an extension device can be changed using the **XChangeFeedbackControl** function. This function modifies the current control values of the specified feedback using information passed in the appropriate **XFeedbackControl** structure for the feedback. Which values are modified depends on the valuemask passed.

Ringing a Bell on an Input Device

A bell on a specified extension input device can be set to ring using the **XDeviceBell** function.

Controlling Device Encoding

The keyboard mapping of an extension device that supports input class **Keys** can be obtained by using the **XGetDeviceKeyMapping** function. The **XFree** function frees the data returned by the **XGetDeviceKeyMapping** function.The keyboard mapping of an extension device that supports input class **Keys** can be changed using the **XChangeDeviceKeyMapping** function.

The keycodes that are used as modifiers on an extension device that supports input class **Keys** are obtained by using the **XGetDeviceModifierMapping** function. The keycodes that are to be used as modifiers for an extension device are set using the **XSetDeviceModifierMapping** function.

Controlling Button Mapping

The mapping of the buttons on an extension device is set using the **XSetDeviceButtonMapping** function. The button mapping on an extension device is obtained using the **XGetDeviceButtonMapping** function.

Obtaining the State of a Device

To obtain information that describes the state of the keys, buttons and valuators of an extension device, use the **XQueryDeviceState** function. The **XFreeDeviceState** function frees the device state data.

Events and Event-Handling Functions

The input extension creates input events analogous to the core input events. These extension input events are generated by manipulating one of the extension input devices. The following sections describe these events and explain how a client program can receive them.

Event Types

Event types are integer numbers that a client can use to determine what kind of event it has received. The client compares the type field of the event structure with known event types to make this determination. The core input event types are constants and are defined in the header file **<X11/X.h>**. Extension event types are not constants. Instead, they are dynamically allocated by the extension's request to the X server when the extension is initialized. Because of this, extension event types must be obtained by the client from the server. The client program determines the event type for an extension event by using the information returned by the **XOpenDevice** request. This type can then be used for comparison with the type field of events received by the client. Extension events propagate up the window hierarchy in the same manner as core events. If a window is not interested in an extension event, it usually propagates to the closest ancestor that is interested, unless the dont_propagate list prohibits it. Grabs of extension devices may alter the set of windows that receive a particular extension event. The following lists the event category and its associated event type or types.

Event Category	Event Type
Device key events	**DeviceKeyPress, DeviceKeyRelease**
Device motion events	**DeviceButtonPress, DeviceButtonRelease, DeviceMotionNotify**
Device input focus events	**DeviceFocusIn**, **DeviceFocusOut**
Device state notification events	**DeviceStateNotify**
Device proximity events	**ProximityIn**, **ProximityOut**
Device mapping events	**DeviceMappingNotify**
Device change events	**ChangeDeviceNotify**

Event Classes

Event classes are integer numbers that are used in the same way as the core event masks. They are used by a client program to indicate to the server which events that client program wishes to receive.

The core input event masks are constants and are defined in the header file **<X11/X.h>**. Extension event classes are not constants. Instead, they are dynamically allocated by the extension's request to the X server when the extension is initialized. Because of this, extension event classes must be obtained by the client from the server.

The event class for an extension event and device is obtained from information returned by the **XOpenDevice** function. This class can then be used in an **XSelectExtensionEvent** request to ask that events of that type from that device be sent to the client program.

For **DeviceButtonPress** events, the client may specify whether or not an implicit passive grab should be done when the button is pressed. If the client wants to guarantee that it receives a **DeviceButtonRelease** event for each **DeviceButtonPress** event it receives, it should specify the **DeviceButtonPressGrab** class in addition to the **DeviceButtonPress** class. This restricts the client in that only one client at a time may request **DeviceButtonPress** events from the same device and window if any client specifies this class.

If any client has specified the **DeviceButtonPressGrab** class, any requests by any other client that specify the same device and window and specify either **DeviceButtonPress** or **DeviceButtonPressGrab** causes an **Access** error to be generated.

If only the **DeviceButtonPress** class is specified, no implicit passive grab is done when a button is pressed on the device. Multiple clients may use this class to specify the same device and window combination.

The client may also select **DeviceMotion** events only when a button is down. It does this by specifying the event classes **DeviceButton1Motion** through **DeviceButton5Motion**. An input device only supports as many button motion classes as it has buttons.

Event Structures

Each extension event type has a corresponding structure declared in **<X11/XInput.h>**. All event structures have the following members:

Member	Description
type	Set to the event type number that uniquely identifies it. For example, when the X server reports a **DeviceKeyPress** event to a client application, it sends an **XDeviceKeyPressEvent** structure.
display	Set to a pointer to a structure that defines the display the event was read on.
send_event	Set to True if the event came from an **XSendEvent** request.
serial	Set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value.

Extension event structures report the current position of the X pointer. In addition, if the device reports motion data and is reporting absolute data, the current value of any valuators the device contains is also reported.

Device Key Events

Key events from extension devices contain all the information that is contained in a key event from the X keyboard. In addition, they contain a device ID and report the current value of any valuators on the device, if that device is reporting absolute data. If data for more than six valuators is being reported, more than one key event is sent. The `axes_count` field contains the number of axes that are being reported. The server sends as many of these events as are needed to report the device data. Each event contains the total number of axes reported in the `axes_count` field, and the first axis reported in the current event in the `first_axis` field. If the device supports input class **Valuators**, but is not reporting absolute mode data, the `axes_count` field contains 0 (zero). The location reported in the `x`, `y` and `x_root`, `y_root` fields is the location of the core X pointer. The **XDeviceKeyEvent** structure is defined as follows:

```
typedef struct
 {
 int type;              /* of event */
 unsigned long serial;  /* # of last request processed*/
 Bool send_event;       /* true if from SendEvent request */
 Display *display;      /* Display the event was read from */
 Window window;         /* "event" window reported relative to */
 XID deviceid;
 Window root;           /* root window event occurred on*/
 Window subwindow;      /* child window */
 Time time;             /* milliseconds */
 int x, y;              /* x, y coordinates in eventwindow */
 int x_root;            /* coordinates relative to root */
 int y_root;            /* coordinates relative to root */
 unsigned int state;    /* key or button mask */
 unsigned int keycode;  /* detail */
 Bool same_screen;      /* same screen flag */
 unsigned char axes_count;
 unsigned char first_axis;
 unsigned int device_state; /* device key or button mask */
 int axis_data[6];
 } XDeviceKeyEvent;
typedef XDeviceKeyEvent XDeviceKeyPressedEvent;
typedef XDeviceKeyEvent XDeviceKeyReleasedEvent;
```

Device Button Events

Button events from extension devices contain all the information that is contained in a button event from the X pointer. In addition, they contain a device ID and report the current value of any valuators on the device, if that device is reporting absolute data. If data for more than six valuators is being reported, more than one button event may be sent. The `axes_count` field contains the number of axes that are being reported. The server sends as many of these events as are needed to report the device data. Each event contains the total number of axes reported in the `axes_count` field, and the first axis reported in the current event in the `first_axis` field. If the device supports input class Valuators, but is not reporting absolute mode data, the `axes_count` field contains 0. The location reported in the `x`, `y` and `x_root`, `y_root` fields is the location of the core X pointer.

```
typedef struct {
 int type;              /* of event */
 unsigned long serial;  /* # of last request processed by
                                 server*/
 Bool send_event;       /* true if from a SendEvent request */
 Display *display;      /* Display the event was read from */
 Window window;         /* "event" window reported relative to */
 XID deviceid;
 Window root;           /* root window that the event occurred
                           on */
 Window subwindow;      /* child window */
 Time time;             /* milliseconds */
 int x, y;              /* x, y coordinates in event window */
 int x_root;            /* coordinates relative to root */
 int y_root;            /* coordinates relative to root */
 unsigned int state;    /* key or button mask */
 unsigned int button;   /* detail */
 Bool same_screen;      /* same screen flag */
 unsigned char axes_count;
 unsigned char first_axis;
 unsigned int device_state;     /* device key or button mask */
 int axis_data[6];
 } XDeviceButtonEvent;
typedef XDeviceButtonEvent XDeviceButtonPressedEvent;
typedef XDeviceButtonEvent XDeviceButtonReleasedEvent;
```

Device Motion Events

Motion events from extension devices contain all the information that is contained in a motion event from the X pointer. In addition, they contain a device ID and report the current value of any valuators on the device. The location reported in the `x`, `y` and `x_root`, `y_root` fields is the location of the core X pointer, and so is 2-dimensional. Extension motion devices may report motion data for a variable number of axes. The `axes_count` field contains the number of axes that are being reported. The server sends as many of these events as are needed to report the device data. Each event contains the total number of axes reported in the `axes_count` field, and the first axis reported in the current event in the `first_axis` field.

```
typedef struct {
 int type;       /* of event */
 unsigned long serial; /* # of last request processed by server */
 Bool send_event;   /* true if from a SendEvent request */
 Display *display;   /* Display the event was read from */
 Window window;    /* "event" window reportedrelative to */
 XID deviceid;
 Window root;     /* root window that the eventoccurred on */
 Window subwindow;   /* child window */
 Time time;      /* milliseconds */
 int x, y;       /* x, y coordinates in eventwindow */
 int x_root;      /* coordinates relative to root */
 int y_root;      /* coordinates relative to root */
 unsigned int state;  /* key or button mask */
 char is_hint;     /* detail */
 Bool same_screen;   /* same screen flag */
 unsigned int device_state; /* device key or button mask */
```

```
unsigned char axes_count;
unsigned char first_axis;
int axis_data[6];
} XDeviceMotionEvent;
```

Device Focus Events

These events are equivalent to the core focus events. They contain the same information, with the addition of a device ID to identify which device has had a focus change, and a time stamp. **DeviceFocusIn** and **DeviceFocusOut** events are generated for focus changes of extension devices in the same manner as core focus events are generated.

```
typedef struct
 {
 int type; /* of event */
 unsigned long serial;   /* # of last request processed by
                                 server */
 Bool send_event;        /* true if this came from a SendEvent
                                 request */
 Display *display;       /* Display the event was read from */
 Window window;          /* "event" window it is reported
                                 relative to */
 XID deviceid;
 int mode;       /* NotifyNormal, NotifyGrab,NotifyUngrab */
 int detail; /*
       * NotifyAncestor, NotifyVirtual, NotifyInferior,
       * NotifyNonLinear,NotifyNonLinearVirtual,              NotifyPointer,
       * NotifyPointerRoot, NotifyDetailNone
       */
 Time time;
 } XDeviceFocusChangeEvent;
typedef XDeviceFocusChangeEvent XDeviceFocusInEvent;
typedef XDeviceFocusChangeEvent XDeviceFocusOutEvent;
```

Device StateNotify Event

This event is analogous to the core keymap event, but reports the current state of the device for each input class that it supports. It is generated after every **DeviceFocusIn** event and **EnterNotify** event and is delivered to clients who have selected **XDeviceStateNotify** events. If the device supports input class **Valuators**, the `mode` field in the **XValuatorStatus** structure is a bitmask that reports the device mode, proximity state and other state information. The following bits are currently defined:

```
0x01    Relative = 0, Absolute = 1
0x02    InProximity = 0, OutOfProximity = 1
```

If the device supports more valuators than can be reported in a single **XEvent**, multiple **XDeviceStateNotify** events are generated.

```
typedef struct
 {
 unsigned char class;
 unsigned char length;
 } XInputClass;

typedef struct {
 int type;
 unsigned long serial;   /* # of last request processed by
                                server */
 Bool send_event;        /* true if this came from a SendEvent
                                request */
 Display *display;       /* Display the event was read from */
 Window window;
 XID deviceid;
 Time time;
 int num_classes;
 char data[64];
 } XDeviceStateNotifyEvent;
```

```
typedef struct {
 unsigned char class;
 unsigned char length;
 unsigned char num_valuators;
 unsigned char mode;
 int valuators[6];
} XValuatorStatus;

typedef struct {
 unsigned char class;
 unsigned char length;
 short num_keys;
 char keys[32];
} XKeyStatus;

typedef struct {
 unsigned char class;
 unsigned char length;
 short num_buttons;
 char buttons[32];
} XButtonStatus;
```

Device Mapping Event

This event is equivalent to the core **MappingNotify** event. It notifies client programs when the mapping of keys, modifiers, or buttons on an extension device has changed.

```
typedef struct {
 int type;
 unsigned long serial;
 Bool send_event;
 Display *display;
 Window window;
 XID deviceid;
 Time time;
 int request;
 int first_keycode;
 int count;
} XDeviceMappingEvent;
```

ChangeDeviceNotify Event

This event has no equivalent in the core protocol. It notifies client programs when one of the core devices has been changed.

```
typedef struct {
 int type;
 unsigned long serial;
 Bool send_event;
 Display *display;
 Window window;
 XID deviceid;
 Time time;
 int request;
 } XChangeDeviceNotifyEvent;
```

Proximity Events

These events have no equivalent in the core protocol. Some input devices such as graphics tablets or touch screens may send these events to indicate that a stylus has moved into or out of contact with a positional sensing surface. The event contains the current value of any valuators on the device, if that device is reporting absolute data. If data for more than six valuators is being reported, more than one proximity event may be sent. The `axes_count` field contains the number of axes that are being reported. The server sends as many of these events as are needed to report the device data. Each event contains the total number of axes reported in the `axes_count` field, and the first axis reported in the current event in the `first_axis` field. If the device supports input class **Valuators**, but is not reporting absolute mode data, the `axes_count` field contains 0.

```
typedef struct
 {
 int type;               /* ProximityIn or ProximityOut */
 unsigned long serial;   /* # of last request processed
                                 by server */
 Bool send_event;        /* true if this came from a
                                 SendEvent request */
 Display *display;       /* Display the event was read from */
 Window window;
 XID deviceid;
 Window root;
 Window subwindow;
 Time time;
 int x, y;
 int x_root, y_root;
 unsigned int state;
 Bool same_screen;
 unsigned char axes_count;
 unsigned char first_axis;
 unsigned int device_state;     /* device key or button mask */
 int axis_data[6];
 } XProximityNotifyEvent;
typedef XProximityNotifyEvent XProximityInEvent;
typedef XProximityNotifyEvent XProximityOutEvent;
```

Determining the Extension Version

The **XExtensionVersion** function allows a client to determine if a server supports the desired version of the input extension. The **XExtensionVersion** structure returns information about the version of the extension supported by the server.

Listing Available Devices

A client program that wishes to access a specific device must first determine whether that device is connected to the X server. This is done through the **XListInputDevices** function, which returns a list of all devices that can be opened by the X server. The client program can use one of the names defined in the **XI.h** header file in an **XInternAtom** request, to determine the device type of the desired device. This type can then be compared with the device types returned by the **XListInputDevices** request. The **XFreeDeviceList** function frees the list of input device information.

Enabling and Disabling Extension Devices

Each client program that wishes to access an extension device must request that the server open that device. This is done through the **XOpenDevice** request.

Before terminating, the client program should request that the server close the device. This is done through the **XCloseDevice** request. A client may open the same extension device more than once. Requests after the first successful one return an additional **XDevice** structure with the same information as the first, but otherwise have no effect. A single **XCloseDevice** request terminates that client's access to the device. Closing a device releases any active or passive grabs the requesting client has established. If the device is frozen only by an active grab of the requesting client, any queued events are released. If a client program terminates without closing a device, the server automatically closes that device on behalf of the client. This does not affect any other clients that may be accessing that device.

Changing the Mode of a Device

Some devices are capable of reporting either relative or absolute motion data. The mode of a device is changed from relative to absolute using the **XSetDeviceMode** function. The valid values are **Absolute** or **Relative**.

Initializing Valuators on an Input Device

Some devices that report absolute positional data can be initialized to a starting value. Devices that are capable of reporting relative motion or absolute positional data may require that their valuators be initialized to a starting value after the mode of the device is changed to **Absolute**. The valuators on such a device are initialized using the **XSetDeviceValuators** function.

Getting Input Device Controls

Some input devices support various configuration controls that can be queried or changed by clients. The set of supported controls varies from one input device to another. Requests to manipulate these controls fail if either the target X server or the target input device does not support the requested device control. Each device control has a unique identifier. Information passed with each device control varies in length and is mapped by data structures unique to that device control. Device controls are queried using the **XGetDeviceControl** function.

Changing Input Device Controls

Some input devices support various configuration controls that can be changed by clients using the **XChangeDeviceControl** function. Typically, this would be done to initialize the device to a known state or configuration. The set of supported controls vary from one input device to another. Requests to manipulate these controls fail if either the target X server or the target input device does not support the requested device control. Setting the device control also fails if the target input device is grabbed by another client, or is open by another client and has been set to a conflicting state.

Selecting Extension Device Events

Device input events are selected using the **XSelectExtensionEvent** function. The parameters passed are a pointer to a list of classes that define the desired event types and devices, a count of the number of elements in the list, and the ID of the window from which events are desired.

Determining Selected Device Events

The extension events that are currently selected from a given window can be determined using the **XGetSelectedExtensionEvents** function.

Controlling Event Propagation

Extension events propagate up the window hierarchy in the same manner as core events. If a window is not interested in an extension event, it usually propagates to the closest ancestor that is interested, unless the *dont_propagate* list prohibits it. Grabs of extension devices may alter the set of windows that receive a particular extension event. Client programs can control event propagation through the use of the **XChangeDeviceDontPropagateList** and **XGetDeviceDontPropagateList** functions.

The **XChangeDeviceDontPropagateList** function adds an event to or deletes an event from the *do_not_propagate* list of extension events for the specified window. There is one list per window, and the list remains for the life of the window. The list is not altered if a client that changed the list terminates. The **XGetDeviceDontPropagateList** function allows a client to determine the *do_not_propagate* list of extension events for the specified window. The **XSendExtensionEvent** function allows a client to send an extension event to another client.

Getting Motion History

The **XGetDeviceMotionEvents** function returns all positions in the device's motion history buffer that fall between the specified start and stop times inclusive. If the start time is in the future, or is later than the stop time, no positions are returned. The **XFreeDeviceMotionEvents** function frees the array of motion information.

```
#endif /* _XI_H_ */
```

AIXwindows Input Extension Protocol Specification

This document defines an extension to the AIXwindows protocol to support input devices other than the core X keyboard and pointer. An accompanying document defines a corresponding extension to Xlib (similar extensions for languages other than C are anticipated). This first section gives an overview of the input extension. The next section defines the new protocol requests defined by the extension. This document concludes with a description of the new input events generated by the additional input devices.

Design Approach

The design approach of the extension is to define requests and events analogous to the core requests and events. This allows extension input devices to be individually distinguished from each other and from the core input devices. These requests and events make use of a device identifier and support the reporting of *n*-dimensional motion data as well as other data that is not reportable by way of the core input events.

Core Input Devices

The X server core protocol supports two input devices: a pointer and a keyboard. The pointer device has two major functions. First, it may be used to generate motion information that client programs can detect. Second, it may also be used to indicate the current location and focus of the X keyboard. To accomplish this, the server echoes a cursor at the current position of the X pointer. Unless the X keyboard has been explicitly focused, this cursor also shows the current location and focus of the X keyboard.

The X keyboard is used to generate input that client programs can detect.

The X keyboard and X pointer are referred to in this document as the *core devices*, and the input events they generate (**KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, and **MotionNotify**) are known as the *core input events*. All other input devices are referred to as *extension input devices* and the input events they generate are referred to as *extension input events*.

Note: This input extension does not change the behavior or functionality of the core input devices, core events, or core protocol requests, with the exception of the core grab requests. These requests may affect the synchronization of events from extension devices. See the explanation in the section titled Event Synchronization and Core Grabs .

Selection of the physical devices to be initially used by the server as the core devices is left implementation dependent. Requests are defined that allow client programs to change which physical devices are used as the core devices.

Extension Input Devices

The input extension controls access to input devices other than the X keyboard and X pointer. It allows client programs to select input from these devices independently from each other and independently from the core devices.

A client that wishes to access a specific device must first determine whether that device is connected to the X server. This is done through the **ListInputDevices** request, which will return a list of all devices that can be opened by the X server. A client can then open one or more of these devices using the **OpenDevice** request, specify what events they are interested in receiving, and receive and process input events from extension devices in the same way as events from the X keyboard and X pointer. Input events from these devices are of extension types (**DeviceKeyPress**, **DeviceKeyRelease**, **DeviceButtonPress**, **DeviceButtonRelease**, **DeviceMotionNotify**, and so on) and contain a device identifier so that events of the same type coming from different input devices can be distinguished.

Any kind of input device may be used as an extension input device. Extension input devices may have zero or more keys, zero or more buttons, and may report zero or more axes of motion. Motion may be

reported as relative movements from a previous position or as an absolute position. All valuators reporting motion information for a given extension input device must report the same kind of motion information (absolute or relative).

This extension is designed to accommodate new types of input devices that may be added in the future. The protocol requests that refer to specific characteristics of input devices organize that information by **input classes**. Server implementors may add new classes of input devices without changing the protocol requests. Input classes are unique numbers registered with the X Consortium. Each extension input device may support multiple input classes.

All extension input devices are treated like the core X keyboard in determining their location and focus. The server does not track the location of these devices on an individual basis, and therefore does not echo a cursor to indicate their current location. Instead, their location is determined by the location of the core X pointer. Like the core X keyboard, some may be explicitly focused. If they are not explicitly focused, their focus is determined by the location of the core X pointer.

Input events reported by the server to a client are of fixed size (32 bytes). In order to represent the change in state of an input device the extension may need to generate a sequence of input events. A client side library (such as Xlib) will typically take these raw input events and format them into a form more convenient to the client.

Event Classes

In the core protocol a client registers interest in receiving certain input events directed to a window by modifying that window's event mask. Most of the bits in the event mask are already used to specify interest in core X events. The input extension specifies a different mechanism by which a client can express interest in events generated by this extension.

When a client opens a extension input device by way of the **OpenDevice** request, an **XDevice** structure is returned. Macros are provided that extract 32-bit numbers called **event classes** from that structure, that a client can use to register interest in extension events by way of the **SelectExtensionEvent** request. The event class combines the desired event type and device ID, and may be thought of as the equivalent of core event masks.

Input Classes

Some of the input extension requests divide input devices into classes based on their functionality. This is intended to allow new classes of input devices to be defined at a later time without changing the semantics of these requests. The following input device classes are currently defined:

KEY Reports key events.
BUTTON Reports button events.
VALUATOR Reports valuator data in motion events.
PROXIMITY Reports proximity events.
FOCUS Can be focused and reports focus events.
FEEDBACK Supports feedbacks.
OTHER The **ChangeDeviceNotify**, **DeviceMappingNotify**, and **DeviceStateNotify** macros may be invoked passing the **XDevice** structure returned for this device.

Each extension input device may support multiple input classes. Additional classes may be added in the future. Requests that support multiple input classes, such as the **ListInputDevices** function that lists all available input devices, organize the data they return by input class. Client programs that use these requests should not access data unless it matches a class defined at the time those clients were compiled. In this way, new classes can be added without forcing existing clients that use these requests to be recompiled.

Extension input devices are accessed by client programs through the use of new protocol requests. This section summarizes the new requests defined by this extension. The syntax and type definitions used below follow the notation used for the X11 core protocol.

Getting the Extension Version

The **GetExtensionVersion** request returns version information about the input extension.

Listing Available Devices

A client that wishes to access a specific device must first determine whether that device is connected to the X server. This is done through the **ListInputDevices** request, which returns a list of all devices that can be opened by the X server.

Enabling Devices

Client programs that wish to access an extension device must request that the server open that device. This is done by way of the **OpenDevice** request. Before it exits, the client program should explicitly request that the server close the device. This is done by way of the **CloseDevice** request.

A client may open the same extension device more than once. Requests after the first successful one return an additional **XDevice** structure with the same information as the first, but otherwise have no effect. A single **CloseDevice** request will terminate that client's access to the device.

Changing the Mode of a Device

Some devices are capable of reporting either relative or absolute motion data. The mode of a device is changed from relative to absolute using the **SetDeviceMode** request. The valid values are **Absolute** or **Relative**.

Initializing Valuators on an Input Device

Some devices that report absolute positional data can be initialized to a starting value. Devices that are capable of reporting relative motion or absolute positional data may require that their valuators be initialized to a starting value after the mode of the device is changed to **Absolute**. The **SetDeviceValuators** request is used to initialize the valuators on such a device.

Getting Input Device Controls

The **GetDeviceControl** request returns the current state of the specified device control. The device control must be supported by the target server and device or an error will result.

The **ChangeDeviceControl** request changes the specified device control according to the values specified in the **DeviceControl** structure. The device control must be supported by the target server and device or an error will result.

Selecting Extension Device Events

Extension input events are selected using the **SelectExtensionEvent** request.

Determining Selected Events

The **GetSelectedExtensionEvents** is used to determine which extension events are currently selected from a given window.

Controlling Event Propagation

Extension events propagate up the window hierarchy in the same manner as core events. If a window is not interested in an extension event, it usually propagates to the closest ancestor that is interested, unless the *dont_propagate* list prohibits it. Grabs of extension devices may alter the set of windows that receive a particular extension event.

Client programs may control extension event propagation through the use of the **ChangeDeviceDontPropagateList** and **GetDeviceDontPropagateList** requests.

The **XChangeDeviceDontPropagateList** function adds an event to or deletes an event from the *do_not_propagate* list of extension events for the specified window. This list is maintained for the life of the window, and is not altered if the client terminates.

Sending Extension Events

One client program may send an event to another by way of the **XSendExtensionEvent** function. The event in the **XEvent** structure must be one of the events defined by the input extension, so that the X server can correctly byte swap the contents as necessary. The contents of the event are otherwise unaltered and unchecked by the X server except to force *SendEvent* to **True** in the forwarded event and to set the sequence number in the event correctly. **XSendExtensionEvent** returns 0 (zero) if the conversion-to-wire protocol failed, otherwise it returns nonzero.

Getting Motion History

The **GetDeviceMotionEvents** request returns all positions in the device's motion history buffer that fall between the specified start and stop times inclusive. If the start time is in the future, or is later than the stop time, no positions are returned.

Changing the Core Devices

These **ChangePointerDevice** and **ChangeKeyboardDevice** requests are provided to change which physical device is used as the X pointer or X keyboard.

Note: Using these requests may change the characteristics of the core devices. The new pointer device may have a different number of buttons than the old one did, or the new keyboard device may have a different number of keys or report a different range of keycodes. Client programs that depend on those characteristics may be running. For example, a client program could allocate an array based on the number of buttons on the pointer device, and then use the button numbers received in button events as indices into that array. Changing the core devices could cause such client programs to behave improperly or abnormally terminate.

These requests change the X keyboard or X pointer device and generate an **ChangeDeviceNotify** event and a **MappingNotify** event. The **ChangeDeviceNotify** event is sent only to those clients that have expressed an interest in receiving that event by way of the **XSelectExtensionEvent** request. The specified device becomes the new X keyboard or X pointer device. The location of the core device does not change as a result of this request.

These requests fail and return **AlreadyGrabbed** if either the specified device or the core device it would replace are grabbed by some other client. They fail and return **GrabFrozen** if either device is frozen by the active grab of another client.

These requests fail with a **BadDevice** error if the specified device is invalid, or has not previously been opened by way of **OpenDevice**.

The **ChangeKeyboardDevice** request changes the X keyboard device. The specified device must support input class **Keys** (as reported in the **ListInputDevices** request) or the request fails with a **BadMatch** error. Once the device has successfully replaced one of the core devices, it is treated as a core device until it is in turn replaced by another **ChangeDevice** request, or until the server terminates. The termination of the client that changed the device will not cause it to change back. Attempts to use the **CloseDevice** request to close the new core device will fail with a **BadDevice** error.

The **ChangePointerDevice** request changes the X pointer device. The specified device must support input class **Valuators** (as reported in the **ListInputDevices** request) or the request fails with a **BadMatch** error. The valuators to be used as the x and y axes of the pointer device must be specified. Data from other valuators on the device are ignored.

Event Synchronization and Core Grabs

Implementation of the input extension requires an extension of the meaning of event synchronization for the core grab requests. This is necessary in order to allow window managers to freeze all input devices with a single request.

The core grab requests require a *PointerMode* and *KeyboardMode* argument. The meaning of these modes is changed by the input extension. For the **XGrabPointer** and **XGrabButton** requests, *PointerMode* controls synchronization of the pointer device, and *KeyboardMode* controls the synchronization of all other input devices. For the **XGrabKeyboard** and **XGrabKey** requests, *PointerMode* controls the synchronization of all input devices except the X keyboard, while *KeyboardMode* controls the synchronization of the keyboard. When using one of the core grab requests, the synchronization of extension devices is controlled by the mode specified for the device not being grabbed.

Extension Active Grabs

Active grabs of extension devices are supported by way of the **GrabDevice** request in the same way that core devices are grabbed using the core **GrabKeyboard** request, except that a *Device* is passed as a function parameter. A list of events that the client wishes to receive is also passed. The **UngrabDevice** request allows a previous active grab for an extension device to be released. To grab an extension device, use the **GrabDevice** request. The device must have previously been opened using the **OpenDevice** request.

Passively Grabbing Buttons and Keys

Passive grabs of buttons and keys on extension devices are supported by way of the **GrabDeviceButton** and **GrabDeviceKey** requests. These passive grabs are released by way of the **UngrabDeviceKey** and **UngrabDeviceButton** requests. To passively grab a single key on an extension device, use **GrabDeviceKey**. That device must have previously been opened using the **OpenDevice** request.

Thawing a Device

The **AllowDeviceEvents** request allows further events to be processed when a device has been frozen.

Controlling Device Focus

The current focus window for an extension input device can be determined using the **GetDeviceFocus** request. Extension devices are focused using the **SetDeviceFocus** request in the same way that the keyboard is focused using the **SetInputFocus** request, except that a device is specified as part of the request. One additional focus state, **FollowKeyboard**, is provided for extension devices.

Controlling Device Feedback

The **GetFeedbackControl** request gets the settings of feedbacks on an extension device. This request provides functionality equivalent to the core **GetKeyboardControl** and **GetPointerControl** functions. It also provides a way to control displays associated with an input device that are capable of displaying an integer or string. The **ChangeFeedbackControl** request changes the settings of a feedback on an extension device.

Ringing a Bell on an Input Device

The **DeviceBell** request rings a bell on an extension input device. The **ChangeFeedbackControl** request changes the base volume of the bell.

Controlling Device Encoding

The **GetDeviceKeyMapping** request gets the keyboard mapping of an extension device that has keys. The **ChangeDeviceKeyMapping** request changes the keyboard mapping of an extension device that has keys.

The **GetDeviceModifierMapping** request obtains the keycodes that are used as modifiers on an extension device that has keys. The **SetDeviceModifierMapping** request sets which keycodes are to be used as modifiers for an extension device.

Controlling Button Mapping

The **GetDeviceButtonMapping** request and the **SetDeviceButtonMapping** request are analogous to the core **GetPointerMapping** and **ChangePointerMapping** requests. They allow a client to determine the current mapping of buttons on an extension device, and to change that mapping. To get the current button mapping for an extension device, use **GetDeviceButtonMapping**. The **SetDeviceButtonMapping** request sets the button mapping for an extension device.

Obtaining the State of a Device

The **QueryDeviceState** request obtains vectors that describe the state of the keys, buttons and valuators of an extension device.

Events

The input extension creates input events analogous to the core input events. These extension input events are generated by manipulating one of the extension input devices.

Button, Key, and Motion Events

The **DeviceKeyPress**,**DeviceKeyRelease**, **DeviceButtonPress**, **DeviceButtonRelease**, and **DeviceMotionNotify** events are generated when a key, button, or valuator logically changes state. The generation of these logical changes may lag the physical changes, if device event processing is frozen. Note that **DeviceKeyPress** and **DeviceKeyRelease** are generated for all keys, even those mapped to modifier bits.

DeviceValuator Event

DeviceValuator events are generated to contain valuator information for which there is insufficient space in **DeviceKey**, **DeviceButton**, **DeviceMotion**, and **Proximity** wire events. For events of these types, a second event of type **DeviceValuator** follows immediately. The library combines these events into a single event that a client can receive by way of **XNextEvent**. **DeviceValuator** events are not selected for by clients, they only exist to contain information that will not fit into some event selected by clients.

Device Focus Events

The **DeviceFocusIn** and **DeviceFocusOut** events are generated when the input focus changes and are reported to clients selecting **DeviceFocusChange** for the specified device and window. Events generated by **SetDeviceFocus** when the device is not grabbed have *Mode* **Normal**. Events generated by **SetDeviceFocus** when the device is grabbed have *Mode* **WhileGrabbed**. Events generated when a device grab actives have *Mode* **Grab**, and events generated when a device grab deactivates have *Mode* **Ungrab**. All **DeviceFocusOut** events caused by a window unmap are generated after any **UnmapNotify** event, but the ordering of **DeviceFocusOut** with respect to generated **EnterNotify**, **LeaveNotify**, **VisibilityNotify** and **Expose** events is not constrained. **DeviceFocusIn** and **DeviceFocusOut** events are generated for focus changes of extension devices in the same manner as focus events for the core devices are generated.

Device State Notify Event

The **DeviceStateNotify** event reports the state of the device just as in the **QueryDeviceState** request. This event is reported to clients selecting **DeviceStateNotify** for the device and window and is generated immediately after every **EnterNotify** and **DeviceFocusIn**. If the device has no more than 32 buttons, no more than 32 keys, and no more than 3 valuators, this event can report the state of the device. If the device has more than 32 buttons, the event will be immediately followed by a **DeviceButtonStateNotify** event. If the device has more than 32 keys, the event will be followed by a **DeviceKeyStateNotify** event. If the device has more than 3 valuators, the event will be followed by one or more **DeviceValuator** events.

Device KeyState and ButtonState Notify Events

The **DeviceKeyStateNotify** and the **DeviceButtonStateNotify** events contain information about the state of keys and buttons on a device that will not fit into the **DeviceStateNotify** wire event. These events are not selected by clients, rather they may immediately follow a **DeviceStateNotify** wire event and be combined with it into a single **DeviceStateNotify** client event that a client may receive by way of **XNextEvent**.

DeviceMappingNotify Event

The **DeviceMappingNotify** event reports a change in the mapping of keys, modifiers, or buttons on an extension device. This event is reported to clients selecting **DeviceMappingNotify** for the device and window and is generated after every client **SetDeviceButtonMapping**, **ChangeDeviceKeyMapping**, or **ChangeDeviceModifierMapping** request.

ChangeDeviceNotify Event

The **ChangeDeviceNotify** event reports a change in the physical device being used as the core X keyboard or X pointer device. **ChangeDeviceNotify** events are reported to clients selecting **ChangeDeviceNotify** for the device and window and is generated after every client **ChangeKeyboardDevice** or **ChangePointerDevice** request.

Proximity Events

The **ProximityIn** and **ProximityOut** events are generated by some devices (such as graphics tablets or touch screens) to indicate that a stylus has moved into or out of contact with a positional sensing surface.

AIXwindows Double Buffer Extension Specification

Introduction

The Double Buffer Extension (DBE) provides a standard way to utilize double bufferiing within the framework of the X Window System. Double buffering uses two buffers, called front and back, which hold images. The front buffer is visible to the user; the back buffer is not. Successive frames of an animation are rendered into the back buffer while the previously rendered frame is displayed in the front buffer. When a new frame is ready, the back and front buffers swap roles, making the new frame visible. Ideally, this exchange appears to happen instantaneously to the user, with no visual artifacts. Thus, only completely rendered images are presented to the user, and remain visible during the entire time it takes to render a new frame. The result is a flicker-free animation.

This article contains the following sections:

- Goals
- Concepts
- Requests
- Encoding
- C Language Binding

Goals

This extension should enable clients to:

- Allocate and deallocate double-buffering for a window.
- Draw to and read from the front and back buffers associated with a window.
- Swap the front and back buffers associated with a window.
- Specify a wide range of actions to be taken when a window is swapped. This includes explicit, simple swap actions (defined below), and more complex actions (for example, clearing ancillary buffers) that can be put together within explicit ``begin'' and ``end'' requests (defined below).
- Request that the front and back buffers associated with multiple double-buffered windows be swapped simultaneously.

In addition, the extension should:

- Allow multiple clients to use double-buffering on the same window.
- Support a range of implementation methods that can capitalize on existing hardware features.
- Add no new event types.
- Be reasonably easy to integrate with a variety of direct graphics hardware access (DGHA) architectures.

Concepts

Normal windows are created using the core **CreateWindow** request, which allocates a set of window attributes and, for InputOutput windows, a front buffer, into which an image can be drawn. The contents of this buffer will be displayed when the window is visible.

This extension enables applications to use double buffering with a window. This involves creating a second buffer, called a back buffer, and associating one or more back buffer names (XIDs) with the window, for use when referring to (for example, drawing to or reading from) the window's back buffer. The back buffer name is a DRAWABLE of type BACKBUFFER.

DBE provides a relative double-buffering model. One XID, the window, always refers to the front buffer. One or more other XIDs, the back buffer names, always refer to the back buffer. After a buffer swap, the window continues to refer to the (new) front buffer, and the back buffer name continues to refer to the (new) back buffer. Thus, applications and toolkits that want to just render to the back buffer always use the back buffer name for all drawing requests to the window. Portions of an application that want to render to the front buffer always use the window XID for all drawing requests to the window.

Multiple clients and toolkits can all use double-buffering on the same window. DBE doesn't provide a request for querying whether a window has double-buffering support, and if so, what the back buffer name is. Given the asynchronous nature of the X Window System, this would cause race conditions. Instead, DBE allows multiple back buffer names to exist for the same window; they all refer to the same physical back buffer. The first time a back buffer name is allocated for a window, the window becomes double buffered and the back buffer name is associated with the window. Subsequently, the window already is a double-buffered window, and nothing about the window changes when a new back buffer name is allocated, except that the new back buffer name is associated with the window. The window remains double buffered until either the window is destroyed, or until all of the back buffer names for the window are deallocated.

In general, both the front and back buffers are treated the same. In particular, here are some important characteristics:

- Only one buffer per window can be visible at a time (the front buffer).
- Both buffers associated with a window have the same visual type, depth, width, height, and shape as the window.
- Both buffers associated with a window are ``visible'' (or ``obscured'') in the same way. When an Expose event is generated for a window, both buffers should be considered to be damaged in the exposed area. Damage that occurs to either buffer will result in an Expose event on the window. When a double-buffered window is exposed, both buffers are tiled with the window background, exactly as stated by the core protocol. Even though the back buffer is not visible, terms such as obscure apply to the back buffer as well as to the front buffer.
- It is acceptable at any time to pass a BACKBUFFER in any request, notably any core or extension drawing request, that expects a DRAWABLE. This enables an application to draw directly into BACKBUFFERs in the same fashion as it would draw into any other DRAWABLE.
- It is an error (Window) to pass a BACKBUFFER in a core request that expects a Window.
- A BACKBUFFER will never be sent by core X in a reply, event, or error where a Window is specified. If core X11 backing-store and save-under applies to a double-buffered window, it applies to both buffers equally.
- If the core **ClearArea** request is executed on a double-buffered window, the same area in both the front and back buffers is cleared.

The effect of passing a window to a request that accepts a DRAWABLE is unchanged by this extension. The window and front buffer are synonymous with each other. This includes obeying the **GetImage** semantics and the subwindow-mode semantics if a core graphics context is involved. Regardless of whether the window was explicitly passed in a **GetImage** request, or implicitly referenced (for example, one of the window's ancestors was passed in the request), the front (i.e., visible) buffer is always referenced. Thus,

DBE-naive screen dump clients will always get the front buffer. **GetImage** on a back buffer returns undefined image contents for any obscured regions of the back buffer that fall within the image.

Drawing to a back buffer always uses the clip region that would be used to draw to the front buffer with a GC subwindow-mode of **ClipByChildren**. If an ancestor of a double-buffered window is drawn to with a core GC having a subwindow-mode of **IncludeInferiors**, the effect on the double-buffered window's back buffer depends on the depth of the double-buffered window and the ancestor. If the depths are the same, the contents of the back buffer of the double-buffered window are not changed. If the depths are different, the contents of the back buffer of the double-buffered window are undefined for the pixels that the **IncludeInferiors** drawing touched.

DBE adds no new events. DBE doesn't extend the semantics of any existing events with the exception of adding a new `DRAWABLE` type called `BACKBUFFER`. If events, replies, or errors that contain a `DRAWABLE` (for example, **GraphicsExpose**) are generated in response to a request, the `DRAWABLE` returned will be the one specified in the request.

DBE advertises which visuals support double-buffering.

DBE does not include any timing or synchronization facilities.

Window Management Operations

The basic philosophy of DBE is that both buffers are treated the same by core X window management operations.

When the core **DestroyWindow** is executed on a double-buffered window, both buffers associated with the window are destroyed, and all back buffer names associated with the window are freed.

If the core **ConfigureWindow** request changes the size of a window, both buffers assume the new size. If the window's size increases, the effect on the buffers depends on whether the implementation honors bit gravity for buffers. If bit gravity is implemented, then the contents of both buffers are moved in accordance with the window's bit gravity (see the core **ConfigureWindow** request), and the remaining areas are tiled with the window background. If bit gravity is not implemented, then the entire unobscured region of both buffers is tiled with the window background. In either case, **Expose** events are generated for the region that is tiled with the window background.

If the core **GetGeometry** request is executed on a `BACKBUFFER`, the returned x, y, and border-width will be zero.

If the **Shape** extension **ShapeRectangles**, **ShapeMask**, **ShapeCombine**, or **ShapeOffset** request is executed on a double-buffered window, both buffers are reshaped to match the new window shape. The region difference

D = NewShape - OldShape

is tiled with the window background in both buffers, and **Expose** events are generated for *D*.

Complex Swap Actions

DBE has no explicit knowledge of ancillary buffers (for example, depth buffers or alpha buffers), and only has a limited set of defined swap actions. Some applications may need a richer set of swap actions than DBE provides. Some DBE implementations have knowledge of ancillary buffers, and/or can provide a rich set of swap actions. Instead of continually extending DBE to increase its set of swap actions, DBE provides a flexible ``idiom'' mechanism. If an application's needs are served by the defined swap actions, it should use them; otherwise, it should use the following method of expressing a complex swap action as an idiom. Following this policy will ensure the best possible performance across a wide variety of implementations.

As suggested by the term ``idiom,'' a complex swap action should be expressed as a group or series of requests. Taken together, this group of requests may be combined into an atomic operation by the implementation, in order to maximize performance. The set of idioms actually recognized for optimization is implementation dependent. To help with idiom expression and interpretation, an idiom must be surrounded by two protocol requests: **DBEBeginIdiom** and **DBEEndIdiom**. Unless this begin-end pair surrounds the idiom, it may not be recognized by a given implementation, and performance will suffer.

For example, if an application wants to swap buffers for two windows, and use core X to clear only certain planes of the back buffers, the application would issue the following protocol requests as a group, and in the following order:

- **DBEBeginIdiom** request.
- **DBESwapBuffers** request with XIDs for two windows, each of which uses a swap action of Untouched.
- Core X **PolyFillRectangle** request to the back buffer of one window.
- Core X **PolyFillRectangle** request to the back buffer of the other window.
- **DBEEndIdiom** request.

The **DBEBeginIdiom** and **DBEEndIdiom** requests don't perform any actions themselves. They are treated as markers by implementations that can combine certain groups or series of requests as idioms, and are ignored by other implementations or for non-recognized groups or series of requests. If these requests are sent out of order, or are mismatched, no errors are sent, and the requests are executed as usual, though performance may suffer.

An idiom need not include a **DBESwapBuffers** request. For example, if a swap action of Copied is desired, but only some of the planes should be copied, a core X **CopyArea** request may be used instead of **DBESwapBuffers**. If **DBESwapBuffers** is included in an idiom, it should immediately follow the **DBEBeginIdiom** request. Also, when the **DBESwapBuffers** is included in an idiom, that request's swap action will still be valid, and if the swap action might overlap with another request, then the final result of the idiom must be as if the separate requests were executed serially. For example, if the specified swap action is Untouched, and if a **PolyFillRectangle** using a client clip rectangle is done to the window's back buffer after the **DBESwapBuffers** request, then the contents of the new back buffer (after the idiom) will be the same as if the idiom weren't recognized by the implementation.

Requests

This section discusses the following requests:

- **DBEGetVersion**
- **DBEGetVisualInfo**
- **DBEAllocateBackBufferName**
- **DBEDeallocateBackBufferName**
- **DBESwapBuffers**
- **DBEBeginIdiom**
- **DBEEndIdiom**
- **DBEGetBackBufferAttributes**

DBEGetVersion

This request returns the major and minor version numbers of this extension.

DBEGetVersion:

```
client-major-version  : CARD8
client-minor-version  : CARD8
=>
server-major-version  : CARD8
server-minor-version  : CARD8
```

The *client-major-version* and *client-minor-version* numbers indicate what version of the protocol the client wants the server to implement. The *server-major-version* and the *server-minor-version* numbers returned indicate the protocol this extension actually supports. This might not equal the version sent by the client. An implementation can (but need not) support more than one version simultaneously. The *server-major-version* and *server-minor-version* allow the creation of future revisions of the Double Buffering Extension protocol which may be necessary. In general, the major version would increment for incompatible changes, and the minor version would increment for small, upward-compatible changes. Servers that support the protocol defined in this document will return a *server-major-version* of one (1), and a *server-minor-version* of zero (0).

The Double Buffering client must issue a **DBEGetVersion request** before any other Double Buffering request in order to negotiate a compatible protocol version, otherwise the client will get undefined behavior (DBE may or may not work).

DBEGetVisualInfo

This request returns information about which visuals support double buffering.

DBEGetVisualInfo:

```
screen-specifiers  : LISTofDRAWABLE
=>
visinfo            : LISTofSCREENVISINFO
```

(Errors: Drawable)

The following type definitions apply:

```
SCREENVISINFO : LISTofVISINFO
VISINFO:        [VISUALID visual;
                CARD8 depth;
                CARD8 perflevel]
```

All of the values passed in *screen-specifiers* must be valid `DRAWABLE`s (or a Drawable error results). For each drawable in *screen-specifiers*, the reply will contain a list of `VISINFO` structures for visuals that support double-buffering on the screen on which the drawable resides. The *visual* member specifies the `VISUALID`. The *depth* member specifies the depth in bits for the visual. The *perflevel* is a performance hint. The only operation defined on a *perflevel* is comparison to a *perflevel* of another visual on the same screen. The visual having the higher *perflevel* is likely to have better double-buffer graphics performance than the visual having the lower *perflevel*. Nothing can be deduced from any of the following: the magnitude of the difference of two *perflevel*s, a *perflevel* value in isolation, or comparing *perflevel*s from different servers.

If the list of *screen-specifiers* is empty, information for all screens is returned, starting with screen zero.

DBEAllocateBackBufferName

This request allocates a drawable ID used to refer to the back buffer of a window.

DBEAllocateBackBufferName:

```
window           : WINDOW
back-buffer-name : BACKBUFFER
swap-action-hint : SWAPACTION
```

(Errors: Alloc, Value, IDChoice, Match, Window)

If the *window* isn't already a double-buffered window, the window becomes double-buffered, and the *back-buffer-name* is associated with the window. The *swap-action-hint* tells the server which swap action is most likely to be used with the window in subsequent **DBESwapBuffers** requests. The *swap-action-hint* must have one of the values specified for type `SWAPACTION` (or a Value error results). See the description of the **DBESwapBuffers** request for a complete discussion of swap actions and the `SWAPACTION` type.

If the window already is a double-buffered window, nothing about the window changes, except that an additional *back-buffer-name* is associated with the window. The window remains double-buffered until either the window is destroyed, or until all of the back buffer names for the window are deallocated.

The window passed into the request must be a valid `WINDOW` (or a Window error results). The window passed into the request must be an InputOutput window (or a Match error results). The visual of the window must be in the list returned by **DBEGetVisualInfo** (or a Match error results). The *back-buffer-name* must be in the range assigned to the client, and must not already be in use (or an IDChoice error results). If the server cannot allocate all resources associated with turning on double-buffering for the window, an Alloc error results, the window's double-buffer* status (whether it is already double-buffered or not) remains unchanged, and the *back-buffer-name* is freed.

DBEDeallocateBackBufferName

This request frees a drawable ID that was obtained by **DBEAllocateBackBufferName**.

DBEDeallocateBackBufferName:

```
back-buffer-name  : BACKBUFFER
```

(Errors: Buffer)

The *back-buffer-name* passed in the request is freed and no longer associated with the window. If this is the last *back-buffer-name* associated with the window, then the back buffer is no longer accessible to clients, and all double-buffering resources associated with the window may be freed. The window's current front buffer remains the front buffer.

The back-buffer-name must be a valid `BACKBUFFER` associated with a window (or a Buffer error results).

DBESwapBuffers

This request swaps the buffers for all windows listed, applying the appropriate swap action for each window.

DBESwapBuffers:

```
windows  : LISTofSWAPINFO
```

(Errors: Match, Window, Value)

The following type definitions apply:

```
SWAPINFO : [ WINDOW window;  SWAPACTION swap-action ]
SWAPACTION : {Undefined, Background, Untouched, Copied}
```

Each window passed into the request must be a valid `WINDOW` (or a Window error results). Each window passed into the request must be a double-buffered window (or a Match error results). Each window passed into the request must only be listed once (or a Match error results). Each *swap-action* in the list must have one of the values specified for type `SWAPACTION` (or a Value error results). If an error results, none of the valid double-buffered windows will have their buffers swapped.

The *swap-action* determines what will happen to the new back buffer of the window it is paired with in the list in addition to making the old back buffer become visible. The defined actions are as follows:

Undefined
: The contents of the new back buffer become undefined. This may be the most efficient action since it allows the implementation to discard the contents of the buffer if it needs to.

Background
: The unobscured region of the new back buffer will be tiled with the window background. The background action allows devices to use a fast clear capability during a swap.

Untouched
The unobscured region of the new back buffer will be unmodified by the swap.

Copied
The unobscured region of the new back buffer will be the contents of the old back buffer.

If **DBESwapBuffers** is included in a ``swap and clear'' type of idiom, it must immediately follow the **DBEBeginIdiom** request.

DBEBeginIdiom

This request informs the server that a complex swap will immediately follow this request.

DBEBeginIdiom: As previously discussed, a complex swap action is a group/series of requests, which, taken together, may be combined into an atomic operation by the implementation. The sole function of this request is to serve as a ``marker'' that the server can use to aid in idiom processing. The server is free to implement this request as a no-op.

DBEEndIdiom

This request informs the server that a complex swap has concluded.

DBEEndIdiom: The sole function of this request is to serve as a ``marker'' that the server can use to aid in idiom processing. The server is free to implement this request as a no-op.

DBEGetBackBufferAttributes

This request returns information about a back buffer.

DBEGetBackBufferAttributes:

```
back-buffer-name  : BACKBUFFER
=>
attributes        : BUFFER_ATTRIBUTES
```

The following type definitions apply:

```
BUFFER_ATTRIBUTES: [WINDOW window;]
```

If *back-buffer-name* is a valid BACKBUFFER, the *window* field of the attributes in the reply will be the window which has the back buffer that *back-buffer-name* refers to. If *back-buffer-name* is not a valid BACKBUFFER, the *window* field of the*attributes* in the reply will be None.

Encoding

Refer to the X11 Protocol Encoding document, as this section uses syntactic conventions and data types established there.

The name of this extension is ``DOUBLE-BUFFER''.

Types

The following new types are used by the extension.

```
BACKBUFFER :  XID

SWAPACTION
 #x00 Undefined
 #x01 Background
 #x02 Untouched
 #x03 Copied

SWAPINFO
 4      WINDOW          window
 1      SWAPACTION      swap action
 3                      unused
```

```
VISINFO
 4      VISUALID         visual
 1      CARD8            depth
 1      CARD8            perflevel
 2                       unused
```

```
SCREENVISINFO
 4      CARD32           n, number in list
 8n     LISTofVISINFO    n VISINFOs
```

```
BUFFER_ATTRIBUTES
4        WINDOW         window
```

Errors

```
Buffer
 1      0                error
 1      error base + 0   code
 2      CARD16           sequence number
 4      CARD32           bad buffer
 2      CARD16           minor-opcode
 1      CARD8            major-opcode
 21                      unused
```

Requests

DBEGetVersion:

```
1   CARD8               major-opcode
1   0                   minor-opcode
2   2                   request length
1   CARD8               client-major-version
1   CARD8               client-minor-version
2                       unused
=>
1   1                   Reply
1                       unused
2   CARD16              sequence number
4   0                   reply length
1   CARD8               server-major-version
1   CARD8               server-minor-version
22                      unused
```

DBEAllocateBackBufferName:

```
1       CARD8            major-opcode
1       1                minor-opcode
4       WINDOW           window
4       BACKBUFFER       back buffer name
1       SWAPACTION       swap action hint
3                        unused
```

DBEDeallocateBackBufferName:

```
1       CARD8            major-opcode
1       2                minor-opcode
4       BACKBUFFER       back buffer name
```

DBESwapBuffers:

```
1   CARD8            major-opcode
1   4                minor-opcode
4   CARD32           n, number of window/swap action pairs in list
8n  LISTofSWAPINFO   window/swap action pairs
```

DBEBeginIdiom:

```
1       CARD8           major-opcode
1       4               minor-opcode
2       1               request length
```

DBEEndIdiom:

```
1       CARD8           major-opcode
1       5               minor-opcode
2       1               request length
```

DBEGetVisualInfo:

```
1   CARD8               major-opcode
1   6                   minor-opcode
2   2+n                 request length
4   CARD32              n, number of screen specifiers in list
4n  LISTofDRAWABLE      n screen specifiers
=>
1   1                   Reply
1                       unused
2   CARD16              sequence number
4   CARD32              reply length
4   CARD32              m, number of SCREENVISINFOs in list
20                      unused
4j  LISTofSCREENVISINFO m SCREENVISINFOs
```

DBEGetBackBufferAttributes:

```
1   CARD8               major-opcode
1   7                   minor-opcode4
2   2                   request length
4   BACKBUFFER          back-buffer-name
=>
1   1                   Reply
1                       unused
2   CARD16              sequence number
4   0                   reply length
4   BUFFER_ATTRIBUTES   attributes
20                      unused
```

C Language Binding

The header for this extension is **<X11/extensions/Xdbe.h>**. All identifier names provided by this header begin with Xdbe.

Types

The type **XdbeBackBuffer** is a `Drawable`.

The type **XdbeSwapAction** can be one of the constants **XdbeUndefined** , **XdbeBackground**, **XdbeUntouched**, or **XdbeCopied**.

Functions

The C routines provide direct access to the protocol and add no additional semantics. For complete details on the effects of these functions, refer to the appropriate protocol request, which can be derived by replacing Xdbe at the start of the function name with DBE. All functions that have return type `Status` will return non-zero for success and zero for failure.

```
Status
XdbeQueryExtension ( Display * dpy , int * major_version_return,
int * minor_version_return
```

Sets *major_version_return* and*minor_version_return* to the major and minor DBE protocol version supported by the server. If the DBE library is compatible with the version returned by the server, this function returns non-zero. If *dpy* does not support the DBE extension, or if there was an error during communication with the server, or if the server and library protocol versions are incompatible, this function returns zero. No other Xdbe functions may be called before this function. If a client violates this rule, the effects of all subsequent Xdbe calls that it makes are undefined.

```
XdbeScreenVisualInfo *
XdbeGetVisualInfo ( Display * dpy, Drawable * screen_specifiers,
int * num_screens
```

This function returns information about which visuals support double buffering. The argument *num_screens* specifies how many elements there are in the *screen_specifiers* list. Each drawable in *screen_specifiers* designates a screen for which the supported visuals are being requested. If *num_screens* is zero, information for all screens is requested. In this case, upon return from this function, *num_screens* will be set to the number of screens that were found. If an error occurs, this function returns NULL, else it returns a pointer to a list of **XdbeScreenVisualInfo** structures of length *num_screens*. The *n*th element in the returned list corresponds to the *n*th drawable in the *screen_specifiers* list, unless*num_screens* was passed in with the value zero, in which case the *n*th element in the returned list corresponds to the *n*th screen of the server, starting with screen zero.

The **XdbeScreenVisualInfo** structure has the following fields:

```
int             count           number of items in  visinfo
XdbeVisualInfo* visinfo         list of visuals and depths for
                                this screen
```

The **XdbeVisualInfo** structure has the following fields:

```
VisualID        visual          one visual ID that supports
                                double-buffering
 int            depth           depth of  visual  in bits
 int            perflevel       performance level of  visual
```

```
void
XdbeFreeVisualInfo (XdbeScreenVisualInfo * visual_info)
```

This function frees the list of **XdbeScreenVisualInfo** returned by the function **XdbeGetVisualInfo**.

```
XdbeBackBuffer
XdbeAllocateBackBufferName (Display * dpy , Window window ,
XdbeSwapAction swap_action)
```

This function returns a drawable ID used to refer to the back buffer of the specified *window*. The *swap_action* is a hint to indicate the swap action that will likely be used in subsequent calls to **XdbeSwapBuffers**. The actual swap action used in calls to **XdbeSwapBuffers** does not have to be the same as the *swap_action* passed to this function, though clients are encouraged to provide accurate information whenever possible.

```
Status
XdbeDeallocateBackBufferName (Display * dpy , XdbeBackBuffer
buffer)
```

This function frees a drawable ID,*buffer*, that was obtained using **XdbeAllocateBackBufferName**. The *buffer* must be a valid name for the back buffer of a window, or an **XdbeBadBuffer** error results.

```
Status
XdbeSwapBuffers (Display * dpy , XdbeSwapInfo * swap_info, int
num_windows)
```

This function swaps the front and back buffers for a list of windows. The argument *num_windows* specifies how many windows are to have their buffers swapped; it is the number of elements in the *swap_info* array. The argument *swap_info* specifies the information needed per window to do the swap.

The **XdbeSwapInfo** structure has the following fields:

```
Window          swap_window     window for which to swap buffers
XdbeSwapAction  swap_action     swap action to use for this
                                wap_window
```

```
Status
XdbeBeginIdiom (Display * dpy)
```

This function marks the beginning of an idiom sequence. See Complex Swap Actions for a complete discussion of idioms.

```
Status
XdbeEndIdiom (Display * dpy)
```

This function marks the end of an idiom sequence.

```
XdbeBackBufferAttributes *
XdbeGetBackBufferAttributes (Display * dpy , XdbeBackBuffer
buffer)
```

This function returns the attributes associated with the specified *buffer*.

The **XdbeBackBufferAttributes** structure has the following fields:

```
Window   window   window that buffer belongs to
```

If *buffer* is not a valid**XdbeBackBuffer**, *window* is set to None.

The returned **XdbeBackBufferAttributes** structure can be freed with the Xlib function **XFree**.

Errors

The **XdbeBufferError** structure has the following fields:

```
int             type
Display *       display          Display the event was read from
XdbeBackBuffer  buffer           resource id
unsigned long   serial           serial number of failed request
unsigned char   error_code       error base + XdbeBadBuffer
unsigned char   request_code     Major op-code of failed request
unsigned char   minor_code       Minor op-code of failed request
```

AIXwindows XTest Extension

Overview

This extension is a minimal set of client and server extensions required to completely test the X11 server without user intervention.

This extension is not intended to support general journaling and playback of user actions. This is a difficult area because it attempts to synchronize synthetic user interactions with their effects; it is at the higher level of dialogue recording and playback rather than at the strictly lexical level. This extension addresses the latter, simpler, case.

This extension provides a minimum set of facilities that solve immediate testing and validation problems.

This extension adheres to the following objectives:

- Confine the extension to an appropriate, high, level within the server to minimize portability problems. In practice this means that the extension should be at the DIX level, or use the DIX/DDX interface, or both. This has effects, in particular, on the level at which ″input synthesis″ can occur.
- Minimize the changes required in the rest of the server.
- Minimize performance penalties on normal server operation.

For additional information, refer to the following topics:

- Description
- Types
- C Language Xlib Binding

Description

The functions provided by this extension fall into two groups, client operations and server requests.

Client Operations

The **XTest** Extension includes the following client operations:

- **XTestSetGContextOfGC**
- **XTestSetGVisualIDOfVisual**
- **XTestDiscard**

These routines manipulate otherwise hidden client-side behavior. The actual implementation will depend on the details of the actual language binding and what degree of request buffering, GContext caching etc., is provided. In the C binding, defined in C Language Xlib Binding , routines are provided to access the internals of two opaque data structures—*GC*s and *visual*s—and to discard any requests pending within the output buffer of a connection. The exact details can be expected to differ for other language bindings.

These are abstract definitions of functionality. They refer to client-side objects like ″GC″ and ″VISUAL″ which are quoted to denote their abstract nature. Concrete versions of these functions are only defined for particular language bindings. In some circumstances a particular language binding may not implement the relevant abstract type or may provide it as a transparent, rather than opaque type, with the result that the corresponding function does not make sense or is not required, respectively.

Server Requests

The **XTest** Extension includes the following server requests:

- **XTestGetVersion**
- **XTestCompareCursor**
- **XTestFakeInput**

The first of these requests is similar to that provided in most extensions: it allows a client to specify a major and minor version number to the server and for the server to respond with major and minor versions of its own. The remaining two requests

1. Allow access to an otherwise ″write-only″ server resource: the cursor associated with a given window; and
2. Perhaps most importantly, allow limited synthesis of input device events, almost as if a cooperative user had moved the pointing device or pressed a key or button.

Types

The following types are used in the request and event definitions.

```
FAKE_EVENT_TYPE: {KeyPress, KeyRelease, MotionNotify, ButtonPress, ButtonRelease}


FAKE_XINPUT_EVENT_TYPE: { XI_DeviceKeyPress, XI_DeviceKeyRelease, XI_DeviceMotionNotify,
 XI_DeviceButtonPress, XI_DeviceButtonRelease, ProximityIn, ProximityOut }


FAKE_EVENT: [type:
FAKE_EVENT_TYPE; detail: BYTE; time: TIME; root: WINDOW; rootX: INT16; rootY:
INT16;]


FAKE_XINPUT_EVENT: [ type
: FAKE_XINPUT_EVENT_TYPE; detail : BYTE; time : TIME; root : WINDOW; rootX :
INT16; rootY : INT16; state : KeyButMask; same_screen : BOOL; deviceid : BYTE ]


FAKE_VALUATOR_EVENT:  [ type :
XI_DeviceValuator; deviceid : BYTE; device_state : KeyButMask; num_valuators :
```

```
BYTE; first_valuator : BYTE; valuator(0-5) : LISTofINT32 ]
```

```
CurrentCursor--1
```

The **XTest** Extension includes the following new types:

```
FAKE_EVENT_TYPE
       2        KeyPress
       3        KeyRelease
       4        ButtonPress
       5        ButtonRelease
       6        MotionNotify
FAKE_XINPUT_EVENT_TYPE
```

Note: The preceding values are defined to be the same as those for the corresponding core protocol event types. The values for FAKE_XINPUT_EVENT_TYPE are defined to be the same as those for the **XInput** Extension.

C Language Xlib Binding

The C Language routines either provide direct access to the protocol and add no additional semantics to those defined in Server Requests or they correspond directly to the abstract descriptions of client operations in Client Operations .

All **XTest** Extension functions and procedures, and all manifest constants and macros, start with the string ″XTest″. All operations are classified as server/client (Server) or client-only (Client). All routines that have return type Status will return non-zero for *success* and zero for *failure*. Even if the **XTest** Extension is supported the server may withdraw such facilities arbitrarily; in which case they will subsequently return zero.

The **XTest** Extension includes the following C Language routines:

- **XTestQueryExtension**
- **XTestCompareCursorWithWindow**
- **XTestCompareCurrentCursorWithWindow**
- **XTestFakeKeyEvent**
- **XTestFakeButtonEvent**
- **XTestFakeMotionEvent**
- **XTestFakeRelativeMotionEvent**
- **XTestGrabControl**
- **XTestSetGContextOfGC**
- **XTestSetVisualIDOfVisual**
- **XTestDiscard**

Chapter 6. AIXwindows Font Enhancements

ISO9241 Compliant Bitmap Fonts

The ISO standard 9241 ″Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs)″, which provides minimum safety and health requirements for work with display screen equipment, was adopted by the Council of European Communities. These requirements for VDTs include the following bitmap font specific items:

- Character height
- Stroke width
- Character width-to-height ratio
- Character format
- Character size uniformity
- Between-character spacing
- Between-word spacing
- Between-line spacing
- Luminance contrast of character details, within or between characters.

The ISO9241-compliant bitmap fonts that are currently available include Ergo character cell fonts: Serif, Sans Serif, and Symbol proportional fonts; and Typewriter monospaced fonts. Since different monitors have different specifications, not all of these fonts will be ISO compliant on all monitors. However, clients can be configured with any font using available customizing tools.

TrueType Rasterizer

A TrueType rasterizer is available in AIXwindows. The TrueType rasterizer supports TrueType 1.0 Font Files. Also, a TrueType font is available on AIXwindows 4.3 which supports most of the character sets currently defined in the Unicode 2.0 specification.

The font directory **/usr/lib/X11/fonts/TrueType** is in the default font path for the AIXwindows X-Server and the AIXwindows Font Server. Users preferring to off-load the rasterization from the X-Server should add an AIXwindows Font Server to the font path of their X-Server.

Font Encodings Supported by the TrueType Rasterizer

The AIXwindows TrueType rasterizer supports the font encodings required by the locales supported on version 4.3. The following tables describes the parameters used when generating a country-specific encoded bitmap font from a Unicode encoded TrueType font:

XLFD Charset	Column		Row		Default Char
	First	Last	First	Last	
iso8859-1	0x20	0xff	0x00	0x00	0x20
iso8859-2	0x20	0xff	0x00	0x00	0x20
iso8859-3	0x20	0xff	0x00	0x00	0x20
iso8859-4	0x20	0xff	0x00	0x00	0x20
iso8859-5	0x20	0xff	0x00	0x00	0x20
iso8859-6	0x20	0xff	0x00	0x00	0x20
iso8859-7	0x20	0xff	0x00	0x00	0x20
iso8859-8	0x20	0xff	0x00	0x00	0x20

© Copyright IBM Corp. 1997, 2002

XLFD Charset	Column		Row		Default Char
	First	Last	First	Last	
iso8859-9	0x20	0xff	0x00	0x00	0x20
IBM-856	0x20	0xff	0x00	0x00	0x20
IBM-1046	0x20	0xff	0x00	0x00	0x20
jisx0201.1976-0	0x20	0xdf	0x00	0x00	0x20
jisx0208.1983-0	0x21	0x7e	0x21	0x74	0x2121
IBM-udcJP	0x21	0x7e	0x65	0x7e	0x7e7e
ksc5601.1987-0	0xa1	0xfe	0xa1	0xfd	0xa1a1
cns11643.1986-1	0x21	0x7e	0x21	0x7d	0x2121
cns11643.1986-2	0x21	0x7e	0x21	0x72	0x2121
cns11643.1992-3	0x21	0x7e	0x21	0x62	0x2121
cns11643.1992-4	0x21	0x7e	0x21	0x6e	0x2121
ibm-udcTW	0x21	0x7e	0x21	0x24	0x2121
ibm-sbdTW	0x21	0x7e	0x21	0x24	0x2121
gb2312.1980-0	0x21	0x7e	0x21	0x77	0x2121
ibm-udcCN	0x21	0x22	0x21	0x22	0x2121
ibm-sbdCN	0x21	0x7e	0x21	0x7e	0x2121
ucs2.thai-0	0x00	0xff	0x00	0xff	0x0020
ucs-2	0x00	0xff	0x00	0xff	0xfffd

Note: This table is shipped in **/usr/lpp/X11/defaults/TrueType/Xlfdcset**. Users may extend the set of encodings supported by the TrueType rasterizer by making additions to this table.

Appendix A. Bidirectional Support in Xm (Motif 1.2) Library

Library Name

/usr/lib/libXm.a

Purpose

The Motif Toolkit is a library that defines a set of widgets with their corresponding functions. The Bidirectional Motif toolkit provides a convenient way of creating Arabic/Hebrew graphical user interfaces.

Description

The additions in **libXm.a** for Arabic/Hebrew fall into two categories. There are changes which deal with the reserved geometry (right-to-left screen orientation). These affect a large number of widgets and gadgets. Whereas other changes deal with inputting bilingual text and supporting bidirectional functionality (that is, character shaping, symmetric swapping, etc.). The later changes affect mostly Text widgets.

Since the additions in **libXm.a** for Arabic/Hebrew fall into two categories, a number of additional resources were incorporated in the toolkit. In the following section is a description of these resources, their names and their possible values.

Bidirectional Resources

These resources control the geometry of the screen as well as the bidi behavior of widgets. These can be set, like other resources, either from a resource file or from inside a program (the **XtSetArg** function). Depending on the way you choose to set them, the way their names and values are written will vary accordingly.

Refer to the Layout Services in the **libi18n.a**file for more information on these values.

If the resources are set from a resource file:

Resource Name	Values	Effect	Default Values
textMode	text_mode_implicit	Text display is in implicit mode	text_mode_implicit
	text_mode_visual	Text display is in visual mode	
	text_mode_explicit	Text display is in explicit mode	
layoutDirection	right_to_left	Global orientation is set Right to Left.	left_to_right
	left_to_right	Global orientation is set Left to Right.	
csdMode (character shaping)	csd_mode_auto- matic	Automatic shaping mode	csd_mode_automatic
	csd_mode_pass- thru	Passthru shaping mode	
	csd_mode_initial	Characters shaped in initial form	
	csd_mode_middle	Characters shaped in middle form	
	csd_mode_final	Characters shaped in final form	
	csd_mode_isolated	Characters shaped in Isolated form	
nssMode (numeric shaping)	nss_mode_bilingual	Numbers represented upon context	nss_mode_bilingual
	nss_mode_arabic	Numbers represented in Arabic	true
	nss_mode_hindu	Numbers represented in Hindu	false

© Copyright IBM Corp. 1997, 2002

Resource Name	Values	Effect	Default Values
	nss_mode_pass- thru	Numbers in passthru mode	false
symmetricSwap	true	Symmetric swapping on	
	false	Symmetric swapping off	
expandTail	true	Seen Family shaped on 2 cells	
	false	Seen Family shaped on 1 cell	
textCompose	true	Diacritics above consonants	
	false	Diacritics beside consonants	

If the resources are set through the widget's argument list (using the **XtSetArg** function):

Resource Name	Values	Effect
XmNtextMode	XmTEXT_MODE_IMPLICIT	Text display is in implicit mode
	XmTEXT_MODE_VISUAL	Text display is in visual upon context mode
	XmTEXT_MODE_EXPLICIT	Text display is in explicit mode
XmNcsdMode	XmCSD_MODE_AUTOMATIC	Automatic shaping mode
	XmCSD_MODE_PASSTHRU	Passthru shaping mode
	XmCSD_MODE_INITIAL	Characters shaped in Initial form
	XmCSD_MODE_MIDDLE	Characters shaped in Middle form
	XmCSD_MODE_FINAL	Characters shaped in Final form
	XmCSD_MODE_ISOLATED	Characters shaped in Isolated form
XmNnssMode	XmNSS_MODE_BILINGUAL	Numbers represented upon context
	XmNSS_MODE_ARABIC	Numbers represented in Arabic
	XmNSS_MODE_HINDU	Numbers represented in Hindu
	XmNSS_MODE_PASSTHRU	Numbers in passthru mode

Note: Take care that another resource not related to the added Arabic/Hebrew support still needs to be set, namely the **fontList** resource. The font list must include an Arabic/Hebrew font name so that Arabic/Hebrew characters can be created.

Example

The following example creates a text widget after setting the layout direction right-to-left and the text mode to implicit. This can be done either from a resource file:

```
*XmText*layoutDirection :right_to_left
*XmText*textMode :text_mode_implicit
```

Or, from inside a program:

```
ArgList args;
XtSetArg(args&1brk.0&rbrk., XmNlayoutDirection, XmRIGHT_TO_LEFT)
XtSetArg(args&1brk.1&rbrk., XmNtextMode, XmTEXT_MODE_IMPLICIT);
XmCreateTextWidget(parent_widget, text_widget_name, args, 2);
```

The Text Widget

The **XmText** widget class is affected by all the bidi resources discussed previously. It can display text in three modes: implicit, explicit and visual. It supports character and numeric shaping in addition to a couple of miscellaneous functions: symmetric swapping and the special handling of the seen family. Furthermore, the text widget supports bidi key combinations, these are the same as the ones used by the **aixterm** Command. The **XmText** widget also reacts to the **layoutDirection** resource. See the following section entitled: Effect of LayoutDirection on Motif Widgets and Gadgets.

The Text Field Widget

The **XmTextField** widget is affected by all the bidi resources. It behaves like the text widget but it does not handle the explicit text mode.

The Label and Gadget Widget

The **XmLabel** and **XmGadget** widget class is affected by the **textMode** resource in addition to the **Primitive** (or **layoutDirection**) resource. See the following section entitled: ″Effect of LayoutDirection on Motif Widgets and Gadgets.″ Setting Implicit Control Support (ICS) and automatic shaping of the label string overrides the default resource to implicit text mode. Otherwise, the label string displays in regular motif behavior.

The List Widget

The **XmList** widget class is also affected by the **textMode** resource in addition to the **layoutDirection** or **Primitive** resource like the **XmLabel** widget class. Once the text mode is set to implicit, the Transformation mechanism applies to the list widget. Otherwise, the list item string displays in regular motif behavior.

Effect of Layout Direction on Motif Widgets and Gadgets

The **layoutDirection** resource in the primitive widget affects the following Motif widgets and Gadgets to enable the bidirectional layout:

Widget Name	Effect
CascadeButton	Reverse side of cascade graphic Bring menu on left
Command	Place prompt string at right hand side, with default of ″<″
DrawButton	Show accelerator at left of label
FileSelectionBox	Lay buttons out from right-to-left Align labels on right hand side Force string direction to right-to-left
Label Widget	Locate accelerator text to left of label text Force string direction to right-to-left
List Widget	Force string direction to right-to-left
MessageBox	Lay buttons out from right-to-left Force string direction to right-to-left Propagate layout direction to subordinate
PushButton	Show accelerator at left of label
RowColumn Widget	Layout subordinates from right-to-left for all menu types

Widget Name	Effect
Scale	If XmNorientation is XmVertical, and XmNshowValue is true, show the value to the left of the scale If XmNorientation is XmHorizontal, default XmNprocessingDirection is XmMAX_ON_LEFT
ScrolledWindow	Propagate direction to subordinates Force scrollBarPlacement to XmBOTTOM_LEFT
ScrollBar	If XmNorientation is XmHORIZONTAL, default XMNprocessingDirection is XmMAX_ON_LEFT
SelectionBox	Lay buttons out from right-to-left and align labels on right-hand side Force string direction to right-to-left Propagate layout direction to subordinates
ToggleButton	Show accelerator at left of label Reverse side of toggle graphic
Text widget	Force text orientation to right-to-left

Example of Localizing a Motif Application for Bidirectional Support

This section provides an example of how to localize the **xmeditor** file in **/usr/sample** for Hebrew and Arabic. The Arabic locales are ar_AA and Ar_AA. The Hebrew locales are Iw_IL and iw_IL.

A translated version of the **xmeditor** file is also supplied. You can compare your changed version of the file to the translated version.

Enter the following command to make sure that the Arabic/Hebrew locales are installed:

```
LANG=iw_IL locale
```

If the categories are not set, the locale may not be installed. Use the SMIT Manage Language Environment menu to install the locales.

Setting the Locale and Changing Geometry

1. Add the following to the **xmeditor.c** file:
   ```
   #include <local.h>
   setlocale(LC_ALL , "");
   ```
2. Compile the **xmeditor** file.
3. Add the following line to your **.Xdefaults** file to work with Arabic in a LeftToRight geometry (LTR):
   ```
   fontList:  -dt-interface
   system-medium-r-normal-L*-*-*-*-*-*-*:
   ```
4. Enter the following command to run the **xmeditor**:
   ```
   LANG=ar_AA/./xmeditor
   ```
5. When the **xmeditor** opens, the keyboard is in the Latin layer. Go to the first text field and try the following:
   a. Press Alt+RightShift and enter data. The keyboard switches to the Arabic/Hebrew layer.
   b. Press Alt+Enter. The whole text widget is toggled and the cursor movements are reversed. The keyboard automatically switches to the appropriate layer.

c. Press Alt+LeftShift and enter data. The data is entered in English.

6. Add the following line to your **.Xdefaults** file to change to global RightToLeft (RTL) geometry, and run the **xmeditor** in a Hebrew or Arabic locale:

```
:wn'
```

and

```
layoutDirection:  right_to_left
```

The whole layout directionality of the **xmeditor** is reversed and the Latin strings are also reversed. Cascading pop-ups also work from right to left. All text fields open right to left and cursor movement is opposite from the way it is in Latin.

Numeric swapping is automatically turned on. To turn it off and always see digits in Arabic (Latin) form set, enter the following:

```
*nssMode: nss_mode_arabic
```

Translating Hard Coded Strings from Latin to Hebrew or Arabic

This example describes how to translate the **File** string in the Main Menu. However, you can use the same process to translate all **xmeditor** strings.

In this example, the strings are hard coded. However, in many applications the strings you need to translate are in separate files which often allows you to test without recompiling anything.

1. Use one of the following commands to open a Hebrew or Arabic aixterm window.
   - For Arabic enter: `LANG=ar_AA aixterm`
   - For Hebrew enter: `LANG=iw_IL aixterm`
2. Use the following steps to edit the **xmeditor.c** file:
   a. Go to the **File** string location.
   b. In an LTR screen, replace the string by entering data in Hebrew or Arabic. To enter data, press Alt+RightShift and enter the text.
   c. As you type, your text should display with all of the characters in the correct sequence.
3. Compile the **xmeditor** file and test it. Your layoutDirection can be either left_to_right or right_to_left. You should be able to see the strings as you enter them using either layout.

Translating External Files for BIDI Text

This section describes how to translate files that are external to the application. This example describes how to translate an external file with the following three names:

```
FILE
HELP
MENU
```

The output of this file as a translated flat file will be in Hebrew or Arabic, where the lower case characters represent Hebrew/Arabic characters:

```
file
help
menu
```

This is how you see the file if you open an En_US aixterm with a Hebrew or Arabic font.

If you open an Arabic or Hebrew aixterm with Arabic or Hebrew font in Implicit mode, the file will display as follows:

```
elif
pleh
unem
```

You can read the strings in the correct order.

If you use an Arabic or Hebrew aixterm to translate the original file, then what you have originally is:

```
FILE
HELP
MENU
```

Now you will start to translate the word FILE. Remove FILE and enter `elif` by pressing `f`, then`i`, then `l`, then `e`, which is the correct order of the word in Hebrew or Arabic.

The result will be in what we see in the Hebrew or Arabic aixterm as follows:

```
elif
HELP
MENU
```

If you display the string as a flat file from an En_US aixterm, the file will display in the correct logical character order as follows:

```
file
HELP
MENU
```

The shaping in Arabic is done automatically in the Arabic aixterm and in the Arabic Motif. When you translate using it, it also shapes the characters for Arabic.

Appendix B. Font Utility

The font utility, **fontutil**, is a tool used for creating and revising font images. Read the following to learn more about the **fontutil** utility:

- Font Utility Introduction
- Font Utility Limitations
- How to Use the Font Utility
- Using the Font Utility Window
- Using the Reference Font Window
- Using the Raster Editor Window
- Using the Vector Editor Window

Font Utility Introduction

The **fontutil** utility is a font editor that runs under the AIXwindows environment and allows you to create custom fonts. The font utility uses windows and is menu-driven.

The utility uses a library of fonts that can be selected and modified. This allows you to start editing a character that looks similar to what you want, rather than using the slower method of drawing a character.

The utility provides two ways to edit fonts: raster editing and vector editing. In raster editing, a display image is composed of an array of pixels arranged in rows and columns. In vector editing, display images are generated from coordinate data.

Raster editing uses a lattice-like grid. Each grid unit contains a pixel. A pixel is a picture element, or dot, on the screen. Together these pixels form a character. By turning these pixels on and off and setting the pixels to different sizes, you can create a new character or slightly alter an existing character. The raster editor is called by the **fontutil** utility to edit font files with the following file name extensions: **.snf** or **.bdf** (that is, files set to **SNF_FORMAT** or **BDF_FORMAT**).

Vector editing uses a coordinate grid. You can set points in this grid, and connect the points to form lines. To create a curve in a character, you connect a series of short, straight lines. Together, all these lines form a character. The vector editor is called by the **fontutil** utility to edit font files with the following file name extensions: **.sym** or **.xgsl** (that is, files set to **graPHIGS_FORMAT** or **XGSL_FORMAT**).

Raster editing is faster but is somewhat limited in what it can do. Scaling and other geometric transformations cannot be easily performed in raster editing.

Vector editing is more flexible and allows for more font customization; however, vector editing requires significant knowledge of XGSL and graPHIGS API fonts. For more information on fonts, see Understanding graPHIGS API Fonts and Understanding Images in graPHIGS API in the publication entitled *The graPHIGS Programming Interface: Understanding Concepts*. For more information on XGSL, see Package XGSL in the publication entitled AIX *Ada/6000 Support Package Reference*.

Note: The font packages are not included with the operating system and must be ordered separately.

The main purpose of the **fontutil** utility is to provide users with the flexibility to do the following:

- Convert raster font files to a vector format (**graPHIGS_FORMAT** or **XGSL_FORMAT**) and edit the characters within the raster font file with the vector editor. Users can also create vector font files that are based on raster font files.

© Copyright IBM Corp. 1997, 2002

- Convert vector font files to a raster format (**SNF_FORMAT** or **BDF_FORMAT**) and edit the characters within the vector font file with the raster editor. Users can also create raster font files that are based on vector font files.

This flexibility allows both advanced and beginning users to edit and create fonts. The advanced user can convert simpler raster font files to a vector format, thus providing the capability for significant customization, while a beginning user can convert the most complex vector font to a raster format, which then can be edited quickly and easily.

Note: Raster to vector conversion should only be performed by those users who have significant experience with creating and editing vector font files.

For complete instructions on getting started with the **fontutil** program, see How to Use the Font Utility.

Font Utility Limitations

Be sure to consider the following limitations when you use the **fontutil** program:

- For SBCS font files, input and output file formats must be the same and **bdf** and **xgsl** formats cannot be handled.
- For DBCS font files, **fontutil** is available only for version 3.1-based Kanji font files.
- When you read a font file which has font data whose encodings are -1, they are ignored; **fontuti**l can neither display them nor save the data.

  Note: **pcf** format font file support is included in version 4.

How to Use the Font Utility

Prerequisite Condition

Before starting the font utility, ensure you are running under the AIXwindows environment.

Note: Many users often change certain visual aspects of an interface in order to suit their individual needs and taste. However, when using the **fontutil** utility users must not change, edit, or delete the **Fontutil** resource file. For example, attempting to customize the font of this interface may cause serious problems with the display of menu messages. Therefore, users must not delete the **/usr/lib/X11/fonts/6x13.snf** file or change the line defining the fixed font in the **/usr/lib/X11/fonts/fonts.alias** file. If *any* changes are made to the **Fontuti**l resource file, the font utility program becomes corrupted or does not run at all.

Procedures

To use the font utility perform the following steps:

1. Enter the start-up command.
2. Create a new file or read an old file.
3. Verify the file header information.
4. Convert the current font format (optional).
5. Edit and save the image.
6. Save the file.
7. Quit the utility.

Enter the Start-Up Command

To begin the font utility, enter the following command:

```
fontutil
```

This command makes the Font Utility window appear.

Create a New File or Read an Old File

From the Font Utility screen, select the **File** option from the window menu bar. Choose either **Create** to create a font file or **Read** to read a font file.

Selecting **Create** displays the File Selection Dialog window.

Specify a font file from which to copy header information, then click on **OK** or press the Enter key. To return to the Font Utility window, select **Cancel**.

Note: On starting the font utility, you must enter a path name or directory where font files can be found. Enter this path name in the **File Filter** box in the File Selection Dialog window. Two such path names are **/usr/lib/X11/fonts** and **/usr/lpp/fonts**. The following are examples of files within these directories that you can use to create a new font file: **Bld*.snf**, **Erg*.snf**, **Itl*.snf**, and **Rom*.snf**.

If you click on **OK**, the Header Information window pops up for your file type. Verify the header information and click on **OK** or press the Enter key. To return to the File Selection Dialog window, click on **Cancel**.

If you choose to create a new file but have made changes to another file that you have not saved, you will be prompted to save the updated font file. Choose **OK** to update the file, **No** to close the file and lose your current work, or **Cancel** to return to the Font Utility window.

Selecting **OK** returns you to the File Selection Dialog window.

Enter the name of the file to be saved in the **Selection** box. (To save a font file, one of the following file name extensions must be used: **.snf**, **.bdf**, **.sym**, **.xgsl**.) Then click on **OK** or press the Enter key. To return to the Font Utility window, select **Cancel**.

If you select **OK**, the Header Information window pops up for your file type. Verify the header information and click on **OK** or press the Enter key. To return to the File Selection Dialog window, click on **Cancel**.

Selecting **Read** from the Font Utility window displays the File Selection Dialog window.

Choose a file name from the list in the **Files** box. Then click on **OK** or press the Enter key. To return to the Font Utility window, select **Cancel**. If you select **OK**, the Header Information window is displayed for your file type. Verify the header information and click on **OK** or press the Enter key. To return to the File Selection Dialog window, click on **Cancel**.

If you choose to read a font file but have made changes to another file that you have not saved, a window is displayed that prompts whether to save the updated file. The utility never allows you to open or edit more than one font file at the same time.

The following file name extensions determine which font editor is used:

.snf Determines that the raster editor is used.

.bdf Determines that the raster editor is used.

.sym Determines that the vector editor is used.

.xgsl Determines that the vector editor is used.

Verify the File Header Information

When you create or read a file, a window pops up to display the header information for the font file you selected in the File Selection Dialog window. Do not make any changes to the information listed in the Header Information window; only verify that you chose the type of file you need. If you make an incorrect choice, you can cancel the verification and choose another font file or you can convert the font format.

Convert the Current Font Format (optional)

If you want to change the header information on your new or existing file, make those changes by selecting the **Format** suboption from the **Extension** option in the Font Utility window.

Edit and Save the Image

Use the raster or vector editor to edit and save your image. See Using the Raster Editor Window and Using the Vector Editor Window for more information.

Save the File

From the Font Utility window menu bar, select the **File** option and choose the **Save** suboption.

Note: To save a file, you must use one of the following file name extensions: **.snf**, **.bdf**, **.sym**, **.xgsl**.

Quit the Utility

From the Font Utility window menu bar, select the **Exit** option. Select the **Exit** suboption to leave the font utility. If you made changes and did not save the file, the utility gives you the option to save the file.

Using the Font Utility Window

Entering the **fontutil** start-up command displays the Font Utility window. Read the following to learn more about the Font Utility window and the four main areas found within it:

- Menu Bar
- Control Panel
- Font Display Area
- Message Area

Menu Bar

The menu bar is displayed across the top of the Font Utility window. The menu options displayed are from left to right: **File**, **Refer**, **Extension**, **Information**, and **Exit**.

File

The **File** option has three suboptions, **Create**, **Read**, and **Save**. These suboptions perform the following functions:

- **Create**

  The **Create** suboption allows you to create a new font file. Chosing to create a font file displays the File Selection Dialog window.

  Select one of the font files listed in the **Files** box from which to copy header information, then click on **OK** or press the Enter key. To return to the Font Utility window, select **Cancel**. If you click on **OK**, the Header Information window is displayed for your file type.

  Verify the header information and click on **OK** or press the Enter key. To return to the File Selection Dialog window, click on **Cancel**.

  If you choose to create a new file but have made changes to another file that you have not saved, you will be prompted to save any changes to the file.

  Choose **OK** to update the file, **No** to close the file and lose your current work, or **Cancel** to return to the Font Utility window. If you select **OK**, the File Selection Dialog window is displayed.

  Enter the name of the file to be saved in the **Selection** box. (To save a font file, one of the following file name extensions must be used: **.snf**, **.bdf**, **.sym**, **.xgsl**.) Then click on **OK** or press the Enter key. To return to the Font Utility window, select **Cancel**.

  If you select **OK**, the Header Information window pops up for your file type. Verify the header information and click on **OK** or press the Enter key. To return to the File Selection Dialog window, click on **Cancel**.

- **Read**

The **Read** suboption allows you to create a new font file by editing specific characters within an existing font file. If you choose to read a font file, the File Selection Dialog window is displayed.

Choose a file name from the list in the **Files** box. The file you choose is displayed in the **Selection** box. Click on **OK** or press the Enter key. To return to the Font Utility window, select **Cancel**. If you select **OK**, the Header Information window is displayed for your file type. Verify the header information and click on **OK** or press the Enter key. To return to the File Selection Dialog window, click on **Cancel**.

If you choose to read a font file but have made changes to another file that you have not saved, you will be prompted to save any changes to the updated file. The utility never allows you to open or edit two font files at the same time.

Choose **OK** to update the file, **No** to close the file and lose your current work, or **Cancel** to return to the Font Utility window.

- **Save**

  The **Save** suboption saves a font file. When you select the **Save** suboption, the File Selection Dialog window is displayed.

  Enter the name of the font file you want to save in the **Selection** box (you can also select the file from the list in the **Files** box). To save a font file, one of the following file name extensions must be used: **.snf**, **.bdf**, **.sym**, **.xgsl**.

Refer

The **Refer** option allows you to reference a font file that is different from the one you are editing or creating. For complete information on the **Refer** option, see Using the Reference Font Window .

Extension

The **Extension** option has three suboptions, **Codeset**, **Format**, and **Size**. The **Extension** suboptions perform the following functions:

- **Codeset**

  When you select the **Codeset** suboption, a pulldown menu is displayed. The menu options are **PC** and **PC16**. Click on the code set you want. After the code set is changed, the smallest code font is displayed in the upper left corner.

- **Format**

  This suboption changes the format of the loaded font. When the **Format** suboption is selected, the Format Convert window is displayed. The selections within the Format Convert window perform the following functions:

Format Convert Function	Changes the suffix of the font file name to:
SNF_FORMAT	.snf
BDF_FORMAT	.bdf
graPHIGS_FORMAT	.sym
XGSL_FORMAT	.xgsl

- **Size**

  This suboption changes the size of the loaded font. When the **Size** suboption is selected, the Size Convert window is displayed.

  The Size Convert window lists the current height and width, and gives you two ways to change the size. You can enter a new height or width in the `NEW` entry, or you can enter a sizing percentage in the **percent** entry. However, you must enter *either* an actual size or a percent, not both.

Information

The **Information** option has the **Information** suboption, which displays the header IBM PC information for the character set you are editing. There are two different types of font files: raster fonts and vector fonts. Depending upon the type of font file you are editing, different information window will be displayed in the Header Information Window.

Exit

The **Exit** option has two suboptions: **Cancel** and **Exit**.

The **Exit** suboption quits the font utility. The **Cancel** suboption cancels the **Exit** option and takes the user back to the Font Utility window.

Control Panel

The control panel runs down the left side of the Font Utility window. The functions available are **Copy**, **Move**, **Delete**, **Edit**, **Mode**, **Undo**, and **Cancel**.

Copy

The **Copy** function copies the specified characters from the Font Utility or Reference Font window. What is copied depends on the mode that is set.

If the **Mode** option is set to **POINT**, you can select and copy only one character. After setting the mode to **POINT** and selecting the **Copy** function, click on the character you want to copy with the left mouse button. Click the left mouse button again in an empty space in the font display area, and the character you copied is pasted in that space.

To copy individual characters from the Reference Font window into the Font Utility window, use the **Copy** function provided in the Font Utility window. The Reference Font window does not have a control panel.

If the **Mode** option is set to **AREA**, select and click on two points (start point and end point) with the left mouse button. Click the left mouse button a third time at the point you want the selected area to be positioned, and the area whose range is specified by the start and end points is copied to the third (destination) point.

If the **Mode** option is set to **TOTAL AREA**, the **Copy** function acts on the entire character set of a font file. You can copy *all* the characters from the Reference Font window to the Font Utility window. After setting the **Mode** option to **TOTAL AREA**, go to the Reference Font window and click your left mouse button on the point in the grid whose coordinates are 0x0. All characters are highlighted to indicate that all characters are to be copied. Next, go back to the Font Utility window and click the left mouse button in the 0x0 point in the grid in its display area. All the characters from the Reference Font window now also appear in the Font Utility window.

Move

The **Move** function moves the specified characters from the Font Utility or Reference Font window. What is moved depends on the mode that is set.

If the **Mode** option is set to **POINT**, you can select and move only one character. After setting the mode to **POINT** and selecting the **Move** function, click on the character you want to move with the left mouse button. Click the left mouse button again in an empty space in the font display area, and the character you are moving is pasted in that space.

To move individual characters from the Reference Font window into the Font Utility window, use the **Move** function provided in the Font Utility window. The Reference Font window does not have a control panel.

If the **Mode** option is set to **AREA**, you can select and click on two points (start point and end point) with the left mouse button. Click the left mouse button a third time at the point you want the selected area to be positioned, and the area whose range is specified by the start and end points is moved to the third (destination) point.

If the **Mode** option is set to **TOTAL AREA**, the **Move** function acts on the entire character set of a font file. You can move *all* the characters from the Reference Font window to the Font Utility window. After setting the **Mode** option to **TOTAL AREA**, go to the Reference Font window and click your left mouse button on

the point in the grid whose coordinates are 0x0. All characters are highlighted, indicating that all characters are to be moved. Next, go back to the Font Utility window and click the left mouse button in the 0x0 point in the grid in its display area. All the characters from the Reference Font window now appear in the Font Utility window.

Note: You can copy and move characters within the Font Utility window and from the Reference Font window to the Font Utility window, but you *cannot* copy and move characters from the Font Utility window to the Reference Font window.

Delete

The **Delete** function removes the specified characters. What is removed depends on the mode that is set.

If the **Mode** function is set to **POINT**, select one character to be removed by clicking on it with your left mouse button. Once the character is highlighted, click your left button again and the character is deleted. If the **Mode** option is set to **AREA**, select and click on two points (start point and end point), and then click the left mouse button once again; the characters within that area are deleted. If the **Mode** option is set to **TOTAL AREA** and you click the left mouse button in the upper left corner of the display area, all characters are deleted.

Edit

The **Edit** function calls the raster or vector editor, depending on the format of the font file (which can be changed with the **Format** suboption under the **Extension** option of the Font Utility window). If the font file ends with the **.snf** or **.bdf** suffix or the font format is changed to **SNF_FORMAT** or **BDF_FORMAT**, then the raster editor is initiated. If the font file ends with the **.sym** or **.xgsl** suffix or the font format is changed to **graPHIGS_FORMAT** or **XGSL_FORMAT**, the vector editor is initiated. Click on the **Edit** function, and then click on the character you want to edit. The window for the appropriate editor is displayed, and the selected character is displayed in the editing area.

Mode

The **Mode** option sets the mode to **POINT**, **AREA**, or **TOTAL AREA**. The selected mode determines how the other functions operate. You can switch among these three modes by clicking on the **Mode** button in the control panel. Each time you click, the next mode is displayed. After the **TOTAL AREA** mode, the selection goes back to **POINT** mode.

Undo

The **Undo** function undoes the last function selected (for example, **Copy** or **Move**). This option is available *only* when the last command entered has completed.

Cancel

The **Cancel** function cancels the currently running command.

Font Display Area

This area is a 10x10 grid square in the center of the Font Utility Window. The image for each character code is displayed in this area; as many as 256 character images can be displayed at one time. Each character image in the grid square is displayed after it is translated to the fixed size.

When displaying the character code, bits 0 through 3 represent the vertical line and bits 4 through 7 represent the horizontal line. For example, if CODE is set to `60`, the vertical line number is set to `8`, and the horizontal line number is set to `A`, the code of the font character is `608A`.

The font display area has a scroll bar that is located directly below the displayed character set. You can use the scroll bar to move within the font file to access the character images you want to edit.

Message Area

The message area appears across the bottom of the screen and has three lines. The top line lists the font file that is opened. The second line displays the current mode, code, and set. The bottom line displays any current messages.

Using the Reference Font Window

Within the menu bar of the Font Utility window there is an option labeled **Refer**. The **Refer** option allows you to reference a font file different from the one you are editing or creating. The **Refer** option provides the Reference Font Window, which displays the character set of a font file. You can copy and move individual characters from this window to the Font Utility window to combine and customize characters from different font files within one new file.

When you select **Refer**, the File Selection Dialog window appears. Specify the font file to which you want to refer, and click on **OK** or press the Enter key. This displays the Reference Font window, which is a character grid of the current font. You can now edit the file displayed in the Font Utility window while referring to the file in the Reference Font window. You can also copy and move characters from the Reference Font window into the Font Utility window. This allows you to combine characters from different fonts within one character set.

The Reference Font window consists of a menu bar across the top, a font display area, and a message area. The font display area displays each character within the font file you specified. The message area displays the file name and set of the file you specified. The menu bar has two options: **Information** and **Exit**.

The **Information** option displays the Header Information window for the character set you are referencing. There are two different types of font files: raster fonts and vector fonts. Depending upon the type of font file you are referencing, different information window will be displayed in the Header Information Window.

The **Exit** option has two suboptions: **Exit** or **Cancel**. The **Exit** suboption exits the Reference Font window and returns to the Font Utility window. The **Cancel** suboption cancels the **Exit** command and returns to the Reference Font window.

Using the Raster Editor Window

The Raster Editor window is the window for editing character images with the raster editor. The raster editor is called by the font utility to edit characters within font files that have a suffix of **.snf** or **.bdf** (are set to **SNF_FORMAT** or **BDF_FORMAT**).

When you select the **Edit** function and a character from the display area in the Font Utility window (and are working with a raster font file), the Raster Editor window is displayed. Read the following to learn more about the Raster Editor window and the four main areas found within it:

- Menu Bar
- Control Panel
- Editing Area
- Message Area

Menu Bar

The menu bar is displayed across the top of the window. The menu options displayed are, from left to right, **Extension**, **Information**, **Entry**, and **Exit**.

Extension

The **Extension** option has two suboptions, **Frame Edit** and **Refer Point**. Selecting the **Frame Edit** displays the Frame Edit window. **Frame Edit** changes the size of the frame that encloses the character image in the editing area.

The `TOP`, `BOTTOM`, `RIGHT`, `LEFT`, and `WIDTH` of the frame can be changed. Click on the item that you wish to change, and enter the new value. Then click on **OK** or press the Enter key. The editing area is changed accordingly. To return to the Raster Editor window, click on **Cancel**.

The **Refer Point** suboption defines a reference point in the editing buffer. The editing buffer provides extra space in which to create and edit character images. Use the scroll bars surrounding the editing area to move the character box visible in the editing buffer. Position the character box so that it encloses a blank area. The area that is enclosed in the character box is the area that appears in the editing area. So, you can turn pixels on and off, and use all the other functions found in the control panel to create a new character image. You can also use the **Pickup** function, the **<** (previous code) function, and the **>** (forward code) function to reference an established character that you want to edit.

The **Refer Point** suboption is useful when creating or editing new characters within the editing buffer. For example, select **Refer Point** (on condition) and a large dot appears over the left corner of the editing area. This point is also visible in the editing buffer. If you select Refer Point again, pointing is stopped (off condition). As you create several new characters and move around within the buffer, you can use reference points to help you guide the character box so that it encloses each character image as it did when you originally created or edited it.

Information

If you click on the **Information** option, you can look at the character information of the character image displayed in the editing area. The **STARTCHAR**, **ENCODING**, **SWIDTH**, **DWIDTH**, and **BBX** fields are shown.

Entry

The **Entry** option has one suboption, **Image Entry**.

The **Image Entry** suboption saves the image of the character in the editing area to the Font Utility window.

The code of the character image displayed in the editing area automatically appears in the **Selection** box when the window is displayed. Enter the code that appears in the **Selection** box by clicking **OK** or pressing the Enter key. To cancel the function and return to the Raster Editor window, click on **Cancel**. If you enter the code, a window is displayed that prompts whether the character data for the character you are saving has changed.

Select **Yes** to save the edited version of the character image, or select **No** to save the image in its original form.

Exit

The **Exit** option has two suboptions: **Exit** and **Cancel**.

The **Exit** suboption exits the raster editor. If you have changed the character image displayed in the editing area and have not saved it, you will be promted to save your work.

Select **Yes** to save your changes, **No** to close the Raster Editor window and lose your current work, or **Cancel** to return to the Raster Editor window. If you select **Yes**, you will be promted to specify the code of the altered character image.

The code of the character image displayed in the editing area automatically appears in the **Selection** box when the window is displayed. Enter the code that appears in the **Selection** box by clicking **OK** or pressing the Enter key. To cancel the function and return to the Raster Editor window, click on **Cancel**.

If you enter the code, you will be promted asking whether the character data for the character you are saving has changed. Select **Yes** to save the edited version of the character image. If you select **No**, the window in which you enter the character code is displayed once again. You can cancel the **Exit** option from this window.

The **Cancel** suboption cancels the **Exit** option and returns to the Raster Editor window.

Control Panel

The control panel runs down the left side of the Raster Editor window and has several functions, which appear in the following order:

Clear

The **Clear** function clears an area in the editing area or the editing buffer. What is cleared depends on the mode that is set. If the **Mode** option is set to **AREA**, select and click on two points (starting bit and ending bit) with the left mouse button. Click the left mouse button a third time and the square whose diagonal points are the starting bit and ending bit is cleared. If the **Mode** option is set to **POINT**, click on any bit in the editing area with the left mouse button and that single bit is cleared. If the **Mode** option is set to **TOTAL AREA**, click on any bit in the editing buffer with the left mouse button and the entire buffer (as well as the editing area) is cleared.

Set

The **Set** function fills an area in the editing area or the editing buffer. What is filled depends on the mode that is set. If the **Mode** option is set to **AREA**, select and click on two points (starting bit and ending bit) with the left mouse button. Click the left mouse button a third time and the area whose diagonal points are the starting bit and ending bit is filled. If the **Mode** option is set to **POINT**, click on any bit in the editing area with the left mouse button and that single bit is filled. If the **Mode** option is set to **TOTAL AREA**, click on any bit in the editing buffer with the left mouse button and the entire buffer (as well as the editing area) is filled.

Reverse

The **Reverse** function changes an area displayed in the editing area or the editing buffer to reverse video. What is changed to reverse video depends on the mode that is set. If the **Mode** option is set to **AREA**, select and click on two points (starting bit and ending bit) with the left mouse button. Click the left mouse button a third time and the square whose diagonal points are the starting bit and ending bit is changed to reverse video. If the **Mode** option is set to **POINT**, click on any bit in the editing area with the left mouse button and that single bit is displayed in reverse video. If the **Mode** option is set to **TOTAL AREA**, click on any bit in the editing buffer with the left mouse button and the buffer (as well as the editing area) is displayed in reverse video.

Copy

The **Copy** function copies the specified range of bits in the editing area. Set the **Mode** option to **AREA**, select and click on two points (starting bit and ending bit) with the left mouse button. Click the left mouse button a third time at the point you want the selected area to be positioned. The area whose range is specified by the start and end points is copied to the third (destination) point.

Overwrite

The **Overwrite** function copies and overwrites the specified range of bits in the editing area. Set the **Mode** option to **AREA**, select and click on two points (starting bit and ending bit) with the left mouse button. Click the left mouse button a third time at the point you want the selected area to be positioned. The area specified by the starting bit and ending bit is copied to the third (destination) point and overwrites what may already exist at the destination.

Move

The **Move** function moves the specified range of bits in the editing area. Set the **Mode** option to **AREA**, select and click on two points (starting bit and ending bit) with the left mouse button. Click the left mouse

button a third time at the point you want the selected area to be positioned; the area specified by the starting bit and ending bit is moved to the third (destination) point.

< (Previous Code)

The **<** (previous code) function retrieves the previous code of the font you are editing. That is, each time you click on the **<** (less than sign) with the left mouse button, the character that precedes the one you are editing is displayed in the editing area. This function allows you to continue editing character images within the same font file without having to return to the Font Utility window to select each character individually.

If you have changed the displayed character image and do not perform an Image Entry before you select the **<** (previous code) function, you will be promoted to save your changes.

If you select **No**, the displayed character is not saved and the image changes to its original design. If you select **Yes**, another window is displayed that prompts you to specify the code for the character image. Once the code is specified and you verify that the character data has changed, the edited version of the character image is saved.

> (Forward Code)

The **>** (forward code) function retrieves the forward code of the font you are editing. That is, each time you click on the **>** (greater than sign) with the left mouse button, the character that follows the one you are editing is displayed in the editing area. This function allows you to continue editing character images within the same font file without having to return to the Font Utility window to select each character individually.

If you have changed the displayed character image and do not perform an Image Entry before you select the **>** (forward code) function, you will be promted to save your changes.

If you select **No**, the displayed character is not saved and the image changes to its original design. If you select **Yes**, another window is displayed that prompts you to specify the code for the character image. Once the code is specified and you verify that the character data has changed, the edited version of the character image is saved.

Pickup

The **Pickup** function retrieves the specified character from the Font Utility window. This allows you to use a character with font characteristics similar to those you want and to edit that character.

To pick up a character from the Font Utility window, click on the **Pickup** function in the Raster Editor window and then click on the character you want in the Font Utility window. The character is displayed in the editing area.

If you have made any changes to the character image originally displayed and do not perform an Image Entry before you select the **Pickup** function, you will be promted to save your changes.

If you select **No**, the character image is not saved and the image will change to its original design. If you select **Yes**, another window is displayed that prompts you to specify the code for the character image. Once the code is specified and you verify that the character data has changed, the edited version of the character image is saved.

Mode

The **Mode** option changes the mode. The mode that is set determines how the other functions operate. Three values can be set: **POINT**, **AREA**, or **TOTAL AREA**. You can switch among these three modes by clicking through the selections with the left mouse button. Each time you click, the next mode is displayed. After the **TOTAL AREA** mode, the selection goes back to **POINT** mode.

Undo

The **Undo** function undoes the last function selected (for example, **Copy** or **Move**). This option is available *only* when the last command entered has completed.

Cancel

The **Cancel** function cancels the currently running command.

Editing Area

This area is a lattice-like grid in the center of the Rasterizer Editor window. Each grid can contain a pixel, which can vary in size. The editing area has two scroll bars, moving buttons (located at either end of the scroll bars), a base line, left line, and right line.

To the right of the editing area is the editing buffer area. The editing buffer area provides extra space in which to create and edit character images and displays the actual size of character images. Use the scroll bars surrounding the editing area to move the character box visible in the editing buffer. The area that is enclosed in the character box is the area that appears in the editing area.

The moving buttons move the focus of the editing area to the right or left, up or down, pixel by pixel. This becomes important when editing characters that may be too large to be displayed (in their entirety) within the screen limitations of the editing area.

The base line emphasizes the ascent and descent SNF parameters. The left line emphasizes the left SNF parameter, and the right line emphasizes the right SNF parameter.

In the editing area, the mouse buttons function as follows: the left button turns the pixel on (fills the pixel), the middle button acts as a toggle switch, and the right button turns the pixel off (clears the pixel).

Message Area

The message area appears across the bottom of the screen and has two lines. The top line lists the current mode, code, and set. The bottom line displays any current messages.

Using the Vector Editor Window

The Vector Editor window is the window for editing character images with the vector editor. The vector editor is called by the font utility to edit characters within font files that have a suffix of **.sym** or **.xgsl** (are set to **graPHIGS_FORMAT** or **XGSL_FORMAT**).

When you select the **Edit** function and a character from the display area in the Font Utility window (and are working with a vector font file), the Vector Editor window is displayed. Read the following to learn more about the Vector Editor window and the four main areas found within.

- Menu Bar
- Control Panel
- Editing Area
- Message Area

Menu Bar

The menu bar appears across the top of the window. The menu options are (left to right) **Extension**, **Information**, **Entry**, and **Exit**.

Extension

The **Extension** option has one suboption, **Frame Edit**. The **Frame Edit** suboption changes the frame information for the character in the editing area.

The **TOP**, **BOTTOM**, **RIGHT**, **LEFT**, and **WIDTH** of the frame can be changed. Click on the item that you wish to change, and enter the new value. Then click on **OK** or press the Enter key. The editing area will be changed accordingly. To return to the Vector Editor window without making changes, click on **Cancel**.

Information

Click on the **Information** option to view the character information of the character image displayed in the editing area. The **FLAGS**, **TOP**, **BOTTOM**, **RIGHT**, **LEFT**, **ASCII Code**, and **EBCDIC Code** fields are shown.

Entry

The **Entry** option has one suboption, **Image Entry**. The **Image Entry** suboption saves the image of the character in the editing area to the Font Utility window. The code of the character image displayed in the editing area automatically appears in the **Selection** box when the window is displayed. Enter the code that appears in the **Selection** box by clicking **OK** or pressing the Enter key. To cancel the function and return to the Vector Editor window, click on **Cancel**. If you enter the code, you are prompted to save any character data that may have changed.

Select **Yes** to save the edited version of the character image, or select **No** to save the image in its original form.

Exit

The **Exit** option has two suboptions: **Exit** and **Cancel**.

The **Exit** suboption exits the vector editor. If you have changed the character image displayed in the editing area and have not saved it, you will be prompted to save any changes.

Select **Yes** to save your changes, **No** to close the Vector Editor window and lose your current work, or **Cancel** to return to the Vector Editor window. If you select **Yes**, you will be prompted to specify the code of the altered character image.

The code of the character image displayed in the editing area automatically appears in the **Selection** box when the window is displayed. Enter the code that appears in the **Selection** box by clicking **OK** or pressing the Enter key. To cancel the function and return to the Vector Editor window, click on **Cancel**.

Control Panel

The control panel runs down the left side of the window and has several functions, displayed in the following order:

Line

The **Line** function forms or edits a straight line in the editing area. The mouse functions as follows:

- If you want to draw a line, use the left button to form an end point. A line follows this end point. You can create another end point at the end of the line with the left mouse button.
- If at this point you do not want the line to continue, click the center button. The center button ends the line.
- If you decide to change the direction of the line you have drawn, click the right mouse button over the center of the line. For example, if a line is angled to the left, it is changed to angle to the right.

Move

The **Move** function moves the specified area within the editing area. What is moved depends on the mode that is set. If the **Mode** option is set to **AREA**, select the area you want to move by holding down the left mouse button and moving the mouse so the area is enclosed in the box that is displayed. Once the area you want to move is enclosed, release the left mouse button, and then click it one more time. The lines in the area selected now move as you move the mouse. Click the left mouse button when the lines are in the position you want. If the **Mode** option is set to **POINT**, click on any single end point and move the mouse to reposition the end point. Click the left mouse button when the end point is positioned where you want it.

Copy

The **Copy** function copies the specified area within the editing area. What is copied depends on the mode that is set. If the **Mode** option is set to **AREA**, select the area you want to copy by holding down the left

mouse button and moving the mouse so the area is enclosed in the box that is displayed. Once the area you want to copy is enclosed, release the left mouse button and click it one more time. The lines in the area selected are now copied and move as you move the mouse. Click the left mouse button when the lines are in the position you want. You can copy individual lines by setting the **Mode** option to **POINT** and clicking on the middle of any line with the left mouse button. The line is copied and you can position it by clicking the left mouse button.

Delete

The **Delete** function removes specified lines from the editing area. What is removed depends on the mode that is set. If the **Mode** option is set to **AREA**, select the area you want to delete by holding down the left mouse button and moving the mouse so the area is enclosed in the box that is displayed. Once the area you want to delete is enclosed, release the left mouse button and click it one more time. The lines in the area selected are deleted. If the **Mode** option is set to **POINT**, click on any end point in the screen of the editing area with the left mouse button and that endpoint is deleted. If the **Mode** option is set to **TOTAL AREA**, click in the editing buffer area with the left mouse button and the lines there (as well as in the editing area) are deleted.

Resize

The **Resize** function allows you to reduce or enlarge specified lines in the editing area. Set the **Mode** option to **AREA,** and select the area you want to resize by holding down the left mouse button and moving the mouse so the area is enclosed in the box that is displayed. Once the area you want to resize is enclosed, release the left mouse button and click it one more time. The lines in the area selected now become smaller or larger (depending on the starting point you select) as you move the mouse. When the lines are the size you want, click the left mouse button to set them at that size.

< (Previous Code)

The **<** (previous code) function retrieves the previous code of the font you are editing. That is, each time you click on the **<** (less than sign) with the left mouse button, the character that precedes the one you are editing is displayed in the editing area. This function allows you to continue editing character images within the same font file without having to return to the Font Utility window to select each character individually.

If you have changed the displayed character image and do not perform an Image Entry before you select the **<** (previous code) function, you will be prompted to save any changes you made.

If you select **No**, the displayed character is not saved and the image changes to its original design. If you select **Yes**, another window is displayed that prompts you to specify the code for the character image. Once the code is specified and you verify that the character data has changed, the edited version of the character image is saved.

> (Forward Code)

The > (forward code) function retrieves the forward code of the font you are editing. That is, each time you click on the **>** (greater than sign) with the left mouse button, the character that follows the one you are editing is displayed in the editing area. This function allows you to continue editing character images within the same font file without having to return to the Font Utility window to select each character individually.

If you have changed the displayed character image and do not perform an Image Entry before you select the **>** (forward code) function, you will be prompted to save any changes you have made.

If you select **No**, the displayed character is not saved and the image changes to its original design. If you select **Yes**, another window is displayed that prompts you to specify the code for the character image. Once the code is specified and you verify that the character data has changed, the edited version of the character image is saved.

Mode

The **Mode** option changes the mode. The mode that is set determines how the other functions operate. Three values can be set: **POINT**, **AREA**, or **TOTAL AREA**. You can switch among these three modes by

clicking through the selections with the left mouse button. Each time you click, the next mode is displayed. After the **TOTAL AREA** mode, the selection goes back to **POINT** mode.

Undo

The **Undo** function undoes the last function selected (for example, **Copy** or **Move**). This option is available *only* when the last command entered has completed.

Cancel

The **Cancel** function cancels the currently running command.

Editing Area

This area is a coordinate grid in the center of the Vector Editor window. End points can appear on or off each coordinate. The editing area has two scroll bars, moving buttons (located at either end of the scroll bars), a cap line, a base line, and a center line.

To the right of the editing area is the editing buffer area. The editing buffer area provides extra space in which to create and edit character images and displays the actual size of character images. The editing buffer area contains a character box. The area that is enclosed in the character box is the same area that appears in the editing area.

Use the scroll bars surrounding the editing area to move the character box in the editing buffer. You can also move the character box, and thus change the focus of the editing area, by clicking on the top left corner of the box with the left mouse button. Now you can position the character box anywhere within the editing buffer area by clicking the left mouse button. After positioning the character box with either method, you can return to the editing area and create or edit a new character.

The moving buttons move the focus of the editing area in single increments to the right or left, or up or down. This becomes important when editing characters that may be too large to be displayed (in their entirety) within the screen limitations of the editing area. The base line, cap line, and center line are parameters for graPHIGS API character images.

Message Area

The message area appears across the bottom of the screen and has two lines. The top line lists the current mode, code, and set. The bottom line displays any current messages.

Appendix C. Display Power Management

Display Power Management (DPM) is available for display devices that conform to the VESA Display Power Management Signaling standard, when attached to graphics display adapters that also support that standard.

The VESA standard defines four levels of power consumption:
- Full on
- Standby
- Suspend
- Off

In full-on mode, the display is in its full power (normal) state with no power savings. The standby, suspend, and off states are the low-power states in order of increasing power savings.

Note: Refer to your display and graphics display adapter product information to determine if your products support the DPM function.

You can define when a supporting display enters each of the power management states through SMIT. Select **Devices** from the main menu, then **Graphic Display**, and then **Display Power Management**. In this dialog there are three time-out values corresponding to each of the low-power states. These time-outs define the interval from the last operator input until the corresponding power management state is entered.

The default values for these time-outs are:

Standby	20 minutes
Suspend	30 minutes
Off	40 minutes

The DPM function is supported in both the LFT and graphics modes. In AIXwindows mode, changes to the time-out values in ODM take effect after the next time the screen saver is deactivated, unless the user has specified power management options on the X Server command line. For LFT mode, changes to the time-out values take effect after the next reboot.

For displays not supporting VESA power management, the AIXwindows screen saver function continues to provide the screen-saving function. A screen-blanking function has been added to LFT mode to reduce burn-in and thus extend the life of the display.

AIXwindows users can also override the DPM time-outs with the **-pm** flag when AIXwindows is started. The **-pm** flag allows three parameters, corresponding to the three time-out values, for example:

```
-pm t1 t2 t3
```

The time-out values are specified in minutes. The command:

```
xinit -s 10 -pm 20 30 40
```

indicates that AIXwindows is to be run with the screen saver activated after 10 minutes and the display to be put in standby state at 20 minutes, suspend state at 30 minutes, and the off state at 40 minutes from the last detected input event.

The DPM time-out function is disabled if either the screen saver is disabled (`-s 0`) or the first DPM (standby) time-out is set to `0` (zero).

© Copyright IBM Corp. 1997, 2002

Appendix D. Setting Up X11 Graphic Input Devices (LPFKeys, Dials, Tablet, Spaceball)

The X11 graphic input devices are supported by X11, GL, graPHIGS, and OpenGL. To set up X11 graphic input devices, refer to the procedure for the appropriate device.

Tablet

- Attach the graphics tablet to either the tablet port or the serial ports of your machine. The tablet port is not available on all machines. When a tablet port is present, it is located next to the mouse and keyboard ports. On some systems the tablet port is labeled with a T. If the graphics tablet is attached to a serial port, you may need to add a tty to the system. To add a tty:
  1. Enter `smit devices`
  2. Select **TTY**.
  3. Select **Add a TTY**.
  4. Select the serial port to which the graphics tablet is connected.
- Install the following device-dependent software package for the graphics tablet:

  AIXwindows Graphics Input Adapter Software

Dials and LPFKeys

- Attach the buttons (lighted programmable function keys or LPFKeys) and dials to either the GIO (Graphics I/O) adapter or the serial ports of your machine. If the dials and LPFKeys are attached to a serial port, you may need to add a tty to the system. To add a tty:
  1. Enter `smit devices`
  2. Select **TTY**.
  3. Select **Add a TTY**.
  4. Select the serial port to which the dials and LPFKeys are connected.
- Install the following device-dependent software packages for dials and LPFKeys:
  - AIXwindows Graphics Input Adapter Software
  - AIXwindows Serial Graphics Input Adapter Software
- When you reboot your system, the dials and LPFKeys that are attached to the GIO adapter should be configured automatically. You can also configure them using SMIT.

  Note: Serially-attached dials and LPFKeys must be configured using SMIT before they are used for the first time. They are configured automatically on subsequent reboots.

  To configure dials and LPFKeys:
  1. Enter `smit devices`
  2. Select **Graphic Input Devices**.
  3. Select **Dials/LPFKeys**.
  4. Select **Add a Dials/LPFKeys**.
  5. Select the device type to configure.
  6. Select the device's attached location (for example, gio0 or tty0).

© Copyright IBM Corp. 1997, 2002

Spaceball

Attach the Spaceball to the serial port of your machine. The Spaceball requires a special serial cable and connector that provides AC needed by the Spaceball.

Note: Some systems may be configured differently. See the information that came with your Spaceball.

- After connecting the Spaceball with the special serial cable, you may need to add a tty to the system. To add a tty:
  1. Enter `smit devices`
  2. Select **TTY**.
  3. Select **Add a TTY**.
  4. Select the serial port to which the Spaceball is connected.
- Install the following device-dependent software packages for the Spaceball:
  - AIXwindows Graphics Input Adapter Software
  - AIXwindows 6094-030 Spaceball 3-D Input Device Software
- To add the actual Spaceball device:
  1. Enter `smit devices`
  2. Select **Graphic Input Devices**.
  3. Select **Spaceball**.
  4. Select **Add Spaceball**.
- You may need to reset the Spaceball by pressing the following buttons on the Spaceball:

  `1, 1, 2, 2, 3, 5, 6, 7, 8`

Appendix E. The X Virtual Frame Buffer

This chapter discusses the X Virtual Frame Buffer, an AIX feature that allows an application to render into the main memory of the computer instead of the hardware graphics adapter. This feature is also related to the hardware, because it can fully or partially replace the hardware.

Overview

The AIX X server has been enhanced to support the X Virtual Frame Buffer technology. The X Virtual Frame Buffer (XVFB) allows the X server to initialize and run without the presence of any physical graphics adapter. In the past, the X server required one or more graphics adapters in order to run and would exit with an error if none were present.

Furthermore, in a standard X Window System environment, each 3D application running on a system must share the same hardware frame buffer. While this is acceptable for viewing clients locally, it is not acceptable for viewing clients remotely when the graphics windows overlap in the screen space. This overlap causes rendering to be serialized and slows overall performance.

Using the XVFB, each application has a private 3D rendering area. Because there is no window overlap, rendering can take place in parallel. The downside for any locally attached display is that you cannot see the image on the screen; it must be displayed over the network to be viewed.

In the XVFB environment, an application renders images on a server machine. These images are then distributed to viewing stations on a network, saved to a database, or used in some other way. The XVFB environment is convenient for viewing 3D graphics on a low-end system for casual users.

The XVFB software is supported by AIX 4.3 or later. The XVFB is also supported on the RS/6000 SP. Install the **X11.vfb** package, which requires approximately 1 MB of disk space. Make sure to reboot the system after installing this package.

Installing the XVFB

This section describes how to install the software for the XFVB on various versions of AIX.

AIX 4.3

The XVFB for AIX 4.3 is installed from the following fileset:

- U454163 - OpenGL.OpenGL_X.dev.vfb.04.03.0000.0000
- U454162 - X11.vfb.04.03.0000.0000

Two PTFs are needed to fix a window resizing problem, but because they are not available, you must migrate to AIX 4.3.1 or later to use the XVFB with AIX 4.3 .

If the PTF U452591 is installed, CATweb Navigator Version 1 runs successfully.

- U452591 - OpenGL.OpenGL_X.rte.soft.4.3.0.1

AIX 4.3.1

The XVFB for AIX 4.3.1 is installed from the AIX 4.3.1 product CDs. Install the following filesets:

- OpenGL.OpenGL_X.dev.vfb.04.03.0001.0000
- X11.vfb.04.03.0001.0000

Two PTFs are needed to fix a window resizing problem and are available on fixdist:

- U456096 - OpenGL.OpenGL_X.dev.vfb.4.3.1.1

© Copyright IBM Corp. 1997, 2002

- U456079 - OpenGL.OpenGL_X.rte.soft.4.3.1.1

AIX 4.3.2

The XVFB for AIX 4.3.2 is installed from the AIX 4.3.2 product CDs. Install the following filesets:

- OpenGL.OpenGL_X.dev.vfb.04.03.0002.0000
- X11.vfb.04.03.0001.0000

AIX 4.3.3

The XVFB for AIX 4.3.3 is installed from the AIX 4.3.3 product CDs. Install the following filesets:

- OpenGL.OpenGL_X.dev.vfb.04.03.0003.0000
- X11.vfb.04.03.0003.0000

To get 8-bit pseudocolor support, install the following PTF, which is available on fixdist:

- U474006 - 4330-06 Recommended Maintenance for AIX 4.3.3

AIX 5.1

The XVFB for AIX 5.1 is installed from the AIX 5.1 product CDs. Install the following filesets:

- OpenGL.OpenGL_X.dev.vfb.05.01.0000.0000
- X11.vfb.05.01.0000.0000

Starting the XVFB

The XVFB is loaded into the X server with the **-vfb** flag:

```
/usr/bin/X11/X -force -vfb -x abx -x dbe -x GLX :n &
```

where *n* is the display number you wish to have associated with this instance of the XVFB. This starts the X server without using any installed graphics adapter and loads the OpenGL extensions to the X server.

The **xinit** command can also be used, which starts both the X server and the window manager:

```
/usr/bin/X11/xinit -- -force -vfb -x abx -x dbe -x GLX :n &
```

The **-vfb** flag can also be added to the *EXTENSIONS* line in your **.xserverrc** file.

To have the XVFB effective at system boot, have the system administrator add the following entry in the **/etc/inittab** file:

```
xvfb:2:respawn:/usr/bin/X11/X -force -vfb -x abx -x dbe -x GLX :n > /dev/null
```

This entry causes the X server to start at system boot time and restart automatically if the server ever exits or dies.

You can run more than one X server at a time, with the following restrictions:

- You cannot use the COSE Desktop.
- You may use multiple instances of the XVFB X server.
- You can use only a single instance of the X server running to a graphics adapter.

If you have a system with a graphics adapter, and you want to run one or more XVFB X servers as well as an X server to your graphics adapter, do the following:

1. Start your X server to the graphics adapter:
   ```
   /usr/bin/X11/xinit
   ```
2. From an xterm/aixterm, start your XVFB server:
   ```
   /usr/bin/X11/X -vfb -x GLX -x abx -x dbe -force :n &
   ```

The display number *n* needs to be other than the one used with the graphics adapter.

Starting with Maintenance Level 4330-06, the XVFB can be started with depth 8 or depth 24 (24 is the default). To specify depth 8, include **-d** 8 on the command line. If the depth is 8, the colorclass will be 3 (pseudocolor). If the depth is 24, the colorclass will be 4 (truecolor). These are the only colorclasses supported for XVFB. Depth 8 is not supported for DirectSoft OpenGL.

Testing the XVFB

Because you cannot see the frame buffer when using XVFB, it is difficult to confirm everything is working correctly. Fortunately, several X clients ship with AIX that can be used to query and view window contents, and can be used to help verify that XVFB is rendering the correct images. These clients include **xwininfo**, **xwd**, and **xwud**.

Verifying that XVFB is Being Used

To verify that an X server is running with the XVFB, use the following command:

```
/usr/lpp/X11/Xamples/bin/xprop -display sysname:n -root | grep VFB
XVFB_SCREEN(STRING) = "TRUE"     <== indicates XVFB is being used
```

where *sysname* is the system name and *n* is the display number for which you are inquiring.

Verifying that XVFB is Working

If you are unsure whether XVFB is installed and started correctly, use the following method to test XVFB. Your system must be on a network and you need access to another system (with a screen) to view the contents of the XVFB.

1. On the XVFB system, start the X server using the **-vfb** flag:
   ```
   /usr/lpp/X11/bin/X -force -vfb -x GLX -x abx -x dbe :n &
   ```
2. On the XVFB system, run the **xclock** client program:
   ```
   xclock -display :n &
   ```
3. On the other system, make sure X is running and that clients can connect:
   ```
   xhost +
   ```
4. On the XVFB system, find the window ID for the **xclock** client:
   ```
   xwininfo -root -tree | grep xclock
   ```

   The first number (0x12345678) is the window ID.
5. On the XVFB system, use **xwd/xwud** to display the client window of the XVFB system on the other system.
   ```
   xwd -id 0x12345678 | xwud -display othersystem:0.0
   ```

   An image of the **xclock** you started on the XVFB system displays on the other system.

Implementing XVFB in Application Code

XVFB allows developers to write Web-based 3D graphics applications for an RS/6000 server without the need of a 3D graphics adapter. In most environments, XVFB-enhanced applications can achieve near-linear scalability by adding processors to multiple-processor systems, because each client can render into its own frame buffer without interaction with the X server. Users viewing data at client systems receive the benefit of XVFB with no changes to software or invocation methods to applications. For an application to operate in a XVFB environment, the application must be enhanced in the following way:

1. To extract a rendered image from a rendering server:

   Extract or retrieve the image to be displayed from the rendering server. Typically, an application would do an **XGetImage** for X applications or a **glReadPixels** for OpenGL applications. The application

decides the frequency and type of actions that will result in the extraction of the image from the rendering server. One criteria might be each time the buffer is swapped using the **XdbeSwapBuffers** or **glXSwapBuffer** subroutine.

2. To send commands to a remote application on a server platform:

   To give the application input to perform, such as opening files or transforming objects, there should be a method to send the application input from a remote source. This can be a command language, a socket connection, interaction with an HTTP server, or some other type of communication services. The **XRecord** and **XTest** extensions can be used to send events to the X server and communicate with the application through traditional X events.

3. To display an image on a display station:

   The XVFB environment should contain a method to display the rendered image on the display station. The method of display of the extracted image is at the discretion of the programmer. If the display station is an AIX workstation executing an AIXwindows X server, the extracted image could be displayed using the **xwud** command. The **xwud** command takes an image in the xwd format, creates a window, and does an **XPutImage** to display the extracted image.

   More sophisticated methods could include image compression and Java display applets. The advantage of a Java display applet is that the image could be displayed on a variety of platform types.

   Applications can also check for the value of the XVFB_SCREEN property to determine if they are running with the XVFB. The following code shows how this can be done:

```
int isXVFB(Display *display,Screen screen)
{
      Atom atom,actual_type;
      int actual_format,status;
      unsigned long nitems,bytes_after;
      unsigned char *prop;
      atom = XInternAtom(display,"XVFB_SCREEN",True);
      if (atom == None)
         return False;
      status = XGetWindowProperty(display,RootWindow(display,screen),
      atom,0,100,False,atom,&actual_type,&actual_format,&nitems,
      &bytes_after,&prop);
      if (strcmp((char*)prop,"TRUE") == 0)
         return True;
      return False;
}
```

Working with the XVFB

The XVFB works by providing an X Device-Dependent X (DDX) layer that drives a software graphics adapter. The frame buffer is stored in system memory and all graphics processing (lines, polygons, text, and so on) is done in the software using the system CPU.

The XVFB is intended to be used in a ″rendering server″ environment. With XVFB, X applications can run and render images, and the images can be queried back into the application for saving to a file, distributing across the network, saving to a database, and so on. In this mode, the X application is not being used directly by a user in an interactive way. Instead, the X application is being driven remotely as a rendering server.

With no physical graphics device, no RAMDAC (random access memory digital-to-analog converter) exists to generate RGB signals. Therefore, it is impossible to connect a monitor to your system and view the contents of the X server Virtual Frame Buffer. This situation is a good solution for rendering server environments, where the added expense of a physical graphics adapter and display are not required. When you cannot see the X frame buffer directly, it does make debugging your application more difficult. It is suggested that applications be developed with a physical graphics adapter and then ported to the XVFB. X client tools like **xwininfo**, **xwd**, and **xwud** can be used to help verify your application is running correctly with the XVFB.

Input devices in the XVFB environment are not required. Because you cannot see the frame buffer, moving the mouse around and typing on the keyboard are not very useful. Rendering server applications are driven remotely with direct socket communication, interaction with an HTTP server, through CORBA connections, message passing interface (MPI), or other methods. Most applications need modifications in order to be controlled remotely for a rendering server environment.

OpenGL is currently supported with the XVFB; however, PEX, GL 3.2, and graPHIGS are not.

DirectSoft OpenGL and the XVFB

DirectSoft OpenGL (DSO) has been implemented to work with the XVFB and was designed specifically to enhance CATweb server performance by:

- Eliminating extraneous interprocess communication
- Eliminating process context-switching overhead
- Making rendering and image reading more direct and efficient.

DSO is a pure software implementation of OpenGL that runs as a direct OpenGL Context. Therefore, all of the CPU-intensive OpenGL work (3D rendering) is part of the application process, not part of the X server process. By running direct, all of the interprocess communications with the X server are eliminated, making 3D rendering much more efficient. As a result, the AIX operating system does not have to perform context switching between the X server and the 3D rendering applications, making system utilization more efficient.

DSO and XVFB enable SMP machines as viable and scalable CATweb servers. DSO creates a private rendering area for each 3D rendering application. When it is time to render a new image, the application draws to its private rendering area. If two CATweb clients request new images at the same time, each of the 3D rendering applications can draw new images concurrently, each to their own private rendering area. Because multiple 3D rendering applications can draw concurrently, multiple CPUs can be exploited concurrently. Without DSO and XVFB, the 3D rendering applications draw to a physical, shared frame buffer, which means they must take turns and operate serially. With each 3D rendering application taking turns to draw, only a single CPU can be effectively exploited.

If you use an OpenGL application with DirectSoft OpenGL and XVFB, the testing scenario with **xwd** and **xwud** discussed earlier in this chapter will not work. This is because to increase performance, the default OpenGL DirectSoft behavior does not blit the contents of the private frame buffer to the X Virtual Frame Buffer. This behavior is fine for most OpenGL rendering server applications because the image will be queried from the private frame buffer using **glReadPixels**. However, using **xwd** and **xwud** just grabs the contents of a blank window.

To verify that DirectSoft OpenGL is working, set the **_OGL_MIXED_MODE_RENDERING** environment variable to `1`, and run your OpenGL application. This will force OpenGL to actually blit the rendered image from the private software frame buffer to the X Virtual Frame Buffer. After this is done, you can use **xwd** and **xwud** to grab and dump the contents of your OpenGL window.

Note: It is recommended that you not keep the **_OGL_MIXED_MODE_RENDERING** environment variable set all the time, because this will slow the overall operation of DirectSoft OpenGL.

Applications can determine if they are using DirectSoft OpenGL by checking the OpenGL rendering string using the **glGetString** subroutine, and checking the OpenGL context using the **glXIsDirect** subroutine. The following code can be used:

```
int isDirectSoftOpenGL(Display *display,GLXContext context)
{
if (glXIsDirect(display,context) == FALSE)
      return FALSE;
```

```
if (strcmp(glGetString(GL_RENDERER),"SoftRaster") == 0)
      return TRUE;
    return FALSE;
}
```

CATweb and the XVFB

CATweb Navigator allows users with Java-enabled Web browsers to view and navigate product information created with CATIA Solutions. By using the intuitive set of Java applets that make up the CATweb Navigator client, users can connect to a CATweb Navigator server machine, select models for viewing, and then view and navigate the models with a 3D viewer, 2D schematic viewer, or a report-style viewer. One CATweb Navigator server machine can support multiple concurrently active CATweb Navigator clients. For each CATweb client that attaches to the CATweb server, one or more CATweb processes is started on the CATweb server to handle the requests of that particular CATweb client. One of these CATweb processes is a 3D rendering application, which runs on the CATweb server and is responsible for:

- Loading the requested CATIA model
- Rendering the 3D model when the CATweb client requests new views
- Compressing the final rendered image
- Transferring the image to the Java CATweb client

This application uses both the X Window System and OpenGL libraries to quickly and accurately render 3D images. XVFB and DirectSoft OpenGL allow CATweb Navigator server machines to operate without expensive graphics adapters and to effectively exploit Symmetric Multi Processing (SMP) machines. In addition, XVFB and DirectSoft OpenGL allow a CATweb Navigator server to be usable at boot time without requiring a user to log in interactively, making it easier to set up and administer a CATweb Navigator server.

The CATweb Navigator does not use the XVFB by default. The image server rendering is done through the X real frame buffer and all the concurrent CATweb renderings are synchronized to share this single resource (CATweb rendering lock). In the XVFB mode, each CATweb process uses its own X Virtual Frame Buffer, and the rendering synchronization is no longer necessary, improving the overall performance in a multi-user environment.

To deactivate the default CATweb rendering lock, change the following value in the **runServerCATIA** file in the **.../CATwebNavigator/bin** directory :

```
VirtualFrameBufferOn=1 (default value = 0)
```

The CATweb rendering unlock is effective at the next CATweb connection.

Appendix F. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Dept. LRAS/Bldg. 003
11400 Burnet Road
Austin, TX 78758-3498
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

© Copyright IBM Corp. 1997, 2002

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:
(c) (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. (c) Copyright IBM Corp. _enter the year or years_. All rights reserved.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

- AIX
- Common User Access
- CUA
- IBM
- Presentation Manager
- RS/6000

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be the trademarks or service marks of others.

Index

A

AIXwindows
- overview 1

app-custom files
- choosing resources 17
- creating 17
- format 14
- location 14
- organizing resource categories 17
- understanding 14
- writing guidelines 17

arguments
- send protocol 32
- variable length 33

B

bidirectional resources 85
bidirectional support 85
browser
- choices 13
- cursors 13
- filenames 13

browsers
- fonts 11
- picture 12
- related information 13
- using 10

buffering
- double 69

C

C language binding 77
CATweb
- X Virtual Frame Buffer 116

changing the mode of a device 65
choices browser 13
choosing resources 17
colors browser 10
core input devices 63
creating app-custom files 17
cursors browser 13
customizing tool
- browsers 10
  - related information 13
- choices browser 13
- colors browser 10
- cursor browser 13
- filenames browser 13
- fonts browser 11
- introduction 5
- main window 6
  - application name 7
  - file pulldown menu 7
  - help pulldown menu 9
  - options pulldown menu 9

customizing tool *(continued)*
- main window *(continued)*
  - related information 10
  - resource category button 7
  - scrolled window area 7
  - view pulldown menu 8
- overview 5
- pictures browser 12
- related information 5
- starting 5
  - basic procedure 5
  - detailed 6
  - prerequisite 5
  - related information 6

D

data structures
- locking 30

DBE
- see double buffer extension 69

DBEAllocateBackBufferName 73, 76
DBEBeginIdiom 75, 76
DBEDeallocateBackBufferName 74, 76
DBEEndIdiom 75, 77
DBEGetBackBufferAttributes 75, 77
DBEGetVersion 72, 76
DBEGetVisualInfo 73, 77
DBESwapBuffers 74, 76
deriving the correct extension opcode 34
design approach 63
DirectSoft OpenGL
- X Virtual Frame Buffer 115

display functions
- opening 21
- using 21

display power management 107
double buffer extension specification 69
- C language binding 77
- complex swap actions 71
- concepts 70
- goals 69
- requests 72, 75
  - errors 76
  - requests 76
  - types 75
- window management 71

DPM
- see display power management 107

dynamically loadable x server extensions 35

E

easter editor window
- control panel
  - clear function 100

© Copyright IBM Corp. 1997, 2002

examples
  localizing a motif application for bidirectional support 88
  setting the locale and changing geometry 88
  translating external files for BIDI text 89
  translating hard coded strings from Latin to Hebrew or Arabic 89
extension devices
  changing input device controls 62
  changing the mode 61
  controlling event propagation 62
  determining selected events 62
  enabling and disabling 61
  getting input device controls 62
  initializing valuators 65
  input 63
  selecting events 62
extensions
  double buffer 69
  functions 27
  input library 51
  protocol requests 27
  screen saver 38
  using 27
  XTest 79

F

file pulldown menu 7
filenames browser 13
files
  app-custom 14
font enhancements 83
  ISO9241 compliant bitmap fonts 83
  TrueType rasterizer 83
font utility 91
  how to use 92
  introduction 91
  limitations 92
  start-up command 92
font utility window
  control panel 96
    Cancel function 97
    Copy function 96
    Delete function 97
    Edit function 97
    Mode option 97
    Move function 96
    Undo function 97
  font display area 97
  Information option 95
  menu bar 94
  message area 98
  using 94
fonts browser 11
format of app-custom files 14

G

gadget 87
gadgets
  layout direction effect 87
GC
  see graphics context 29
getting input device controls 65
getting the extension version 65
graphic input devices
  setting up 109
    dials 109
    LPFKeys 109
    spaceball 110
    tablet 109
graphical overlay planes 23
  GXT_OVERLAYS 24
graphics
  batching 29
graphics context
  caching 29

H

help pulldown menu 9
hooking into the Xlib 28

I

input device functions
  controlling input focus 48
  keyboard and pointer settings 49
  keyboard encoding 50
  keyboard grabbing 48
  moving the pointer 48
  pointer grabbing 46
  resuming event processing 48
Input Device Functions in Enhanced X Windows 46
input extension library 51
  core input devices 52
  design approach 51
  extension input devices 52
  input device classes 53
  library extension requests 53
  listing available devices 61
  purpose 51
  using extension input devices 53
  window manager functions 54
input extension protocol specification 63

L

library
  bidirectional support in Xm 85
Library
  Xlib 19
listing available devices 65
location of app-custom files 14
lock data structures 30

M

main window for customizing tool 6

memory
  allocating 34
  deallocating 34

N

national language support 25
  exceptions 25
nls
  see national language support 25
nonrectangular window shape extension 35

O

options pulldown menu 9
organizing resource categories 17

P

pictures browser 12
protocol
  sending requests and arguments 32

R

raster editor window
  control panel 100
    > (forward code) function 101
    <<<< (previous code) function 101
    cancel function 102
    copy function 100
    mode option 101
    move function 100
    overwrite function 100
    pickup function 101
    reverse function 100
    set function 100
    undo function 101
  editing area 102
  entry option 99
  exit option 99
  extension option 99
  information option 99
  menu bar 98
  message area 102
  using 98
reference font window
  using 98
replies
  defining
    formats 32
    using Enhanced-Windows 30
requests
  defining
    formats 31
    using Enhanced X-Windows 30
  send protocol 32
resource category button 7

S

screen saver extension 38
  assumptions 39
  errors 39
  events 41
  requests 39
  types 39
scrolled window area 7
setting up graphic input devices 109
shared memory transport 22
  environment variables 23
  goals 22
  other mechanisms 23
SMT
  see shared memory transport 22
stub
  deriving correct extension opcode 34
stubs
  writing 30
synchronous calling 34

U

understanding the app-custom files 14
using extensions in AIXwindows 27

V

vector editor window
  control panel 103
    > (forward code) function 104
    <<<< (previous code) function 104
    Cancel function 105
    Copy function 103
    Delete function 104
    Line function 103
    Mode option 104
    Move function 103
    Resize function 104
    Undo function 105
  editing area 105
  Entry option 103
  Exit option 103
  Extension option 102
  Information option 103
  menu bar 102
  message area 105
  using 102
view pulldown menu 8

W

widgets
  gadget 87
  label 87
  layout direction effect 87
  list 87
  text 87
  text field 87

window manager
  overview 3
  restoring defaults 3
  starting and exiting 3
window shape
  nonrectangular 35
writing app-custom files
  guidelines 17

X

X Virtual Frame Buffer 111
  CATweb 116
  DirectSoft OpenGL 115
  implementing in code 113
  installing 111
  starting 112
  testing 113
  working with 114
Xlib Library 19
xserver extensions
  dynamically loadable 35
XTest Extension 79
XVFB
  see X Virtual Frame Buffer 111

Printed in U. S. A.

SC23-4870-02