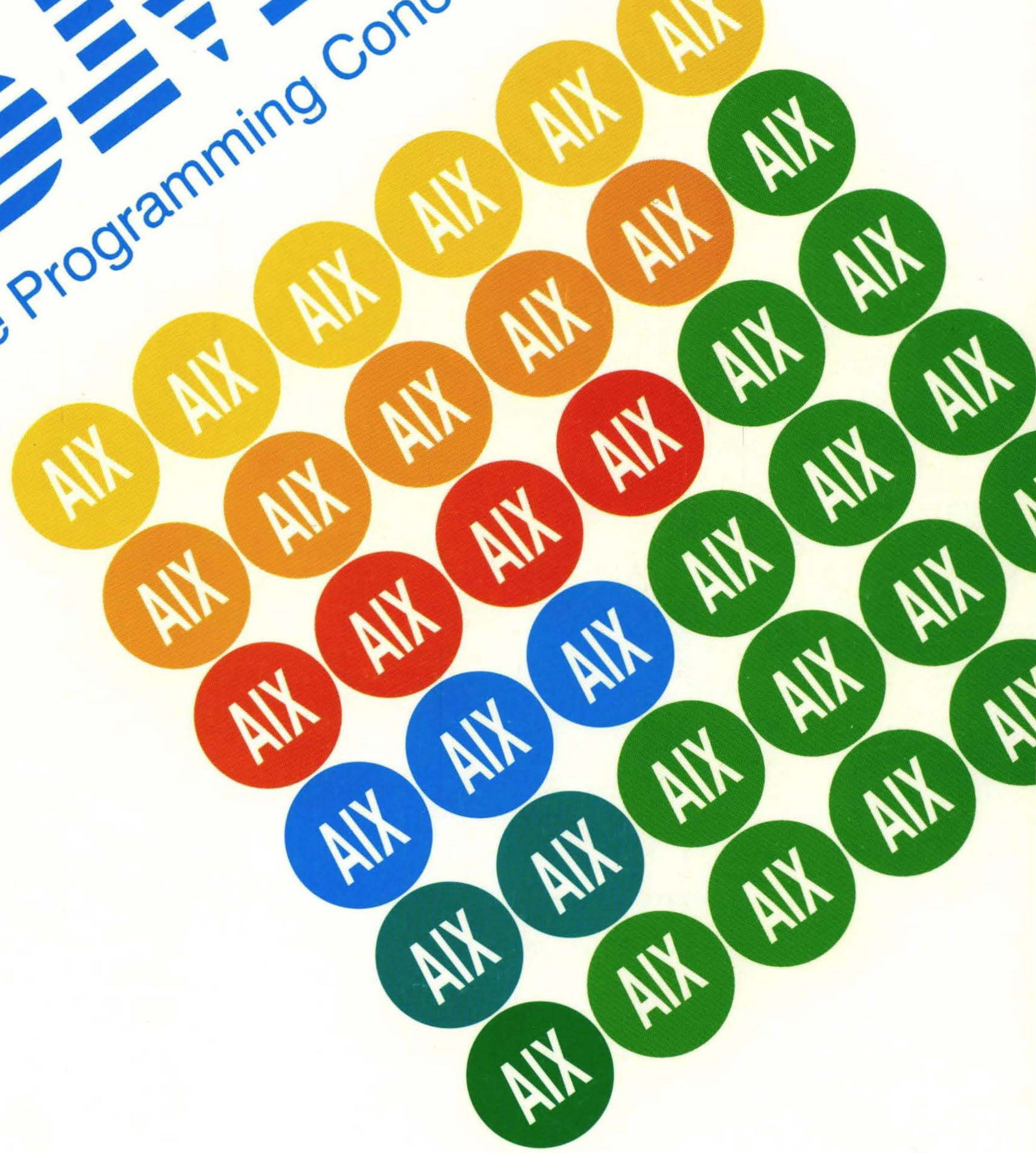
AIX Version 3 for RISC System/6000™

User Interface Programming Concepts



SC23-2209-01

Second Edition (November 1990)

Trademarks and Acknowledgements

About This Book

This book, *AIX User Interface Programming Concepts for RISC System/6000,* describes how to customize the behavior and appearance of the AIXwindows Desktop and Window Manager, how to use the AIXwindows toolkit, how to use X–Windows subroutines and libraries, how to use the Curses Subroutine Library, and how to design interactive user interfaces. It is the programmer's main source book for learning about user interfaces from a programming perspective.

Who Should Use This Book

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

How This Book is Organized

This book contains six chapters:

- Chapter 1. AIXwindows Desktop, provides information necessary to customize the behavior and appearance of the AIXwindows Desktop.
- Chapter 2. AIXwindows Window Management, provides information necessary to tailor the window management environment.
- Chapter 3. AIXwindows Toolkit, describes the contents of the AIXwindows toolkit (objects and subroutines) and explains how to use them to design interactive user interfaces, including character set support.
- Chapter 4. AIXwindows Style Guide, facilitates the design of user interfaces with a consistant behavior and appearance.
- Chapter 5. AIXwindows User Interface Language (UIL) and Resource Manager (MRM) Overview for Programming, describes the UIL, a compiled specification language used for describing the initial state of a user interface, and the MRM, which consists of library subroutines that access UIL at run time and create a user interface.
- Chapter 6. Enhanced X–Windows, provides a conceptual introduction to using Enhanced X–Windows subroutines, macros, protocols, extensions, and events.
- Chapter 7. Curses and Extended Curses, provides a conceptual overview of the Curses and Extended Curses Subroutine Libraries.

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, keywords, files, directories, and other items whose names are predefined by the system.
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.

Related Publications

The following books contain information about or related to programming the user Interface:

- *AIX Calls and Subroutines Reference for IBM RISC System/6000*, Order Number SC23–2198.
- *AIX Commands Reference for IBM RISC System/6000*, Order Number SC23–2199.
- *AIX Files Reference for IBM RISC System/6000*, Order Number SC23–2200.
- *Task Index and Glossary for IBM RISC System/6000*, Order Number SC23–2201.
- *AIX General Concepts and Procedures for IBM RISC System/6000*, Order Number SC23–2202.
- *AIX General Programming Concepts for IBM RISC System/6000,* Order Number SC23–2205.
- *AIX Communications Programming Concepts for IBM RISC System/6000*, Order Number SC23–2206.

Ordering Additional Copies of This Book

To order additional copies of this book, use Order Number SC23–2209.

Contents

Chapter 1. AIXwindows Desktop

Customizing the AIXwindows Desktop Overview

The behavior and appearance of the AIXwindows Desktop are not fixed, but are determined by rule files and resource definitions that can be edited to suit individual requirements.

The resource mechanism determines the default appearance of the main components of the Desktop, such as:

- The font used for text
- The spacing around text
- The spacing between icons
- Background patterns
- Mouse cursor shapes and mouse button triggers
- The thickness of the lines between items on menus
- Pictures for message boxes
- Text layout in message windows
- Pictures for directory controls.

In addition, the resource files specify the mapping between the mouse clicks and presses and the labels used by the system (which are called triggers). These can be altered to work with mouses that have a different number or layout of buttons.

Normally, each computer running the desktop has a resource file suited to the particular machine on which it is being run. For example, machines with large screens can use larger pictures for window controls to improve legibility.

However, users can provide their own resource files to give any desired appearance, or can switch between resource files to provide different working environments for different applications.

A separate set of files, called rule files, determine:

- Desktop layout
- Titles for files and directories
- Pictures for files and directories
- Actions when a user double–clicks on an icon
- Actions when one or more icons are dragged onto another icon
- Actions when one or more icons are dragged into a directory window
- Appearance of menus, and the action when a menu item is selected.

The default behavior of the desktop is determined by a system rule file, but this can be overridden by additional rule files that are provided by each user. Furthermore, rule files can be created that are local to a specific directory.

Rule files consist of a number of different components that determine the following aspects of the behavior of the desktop:

Icon appearance and title	The picture displayed for a file or group of files, and the title displayed below it.
Icon activation	What happens when the user double clicks the mouse on the icon.
Directory appearance	What picture is used to create the background pattern for the directory window.
Drop behavior	What happens when one or more icons are dragged into a directory window, and what happens when another icon is dropped onto an icon.
Menu appearance and behavior	What happens when a menu item is selected.
Desktop layout	Which files are on the desktop, and what their positions are.
Locked files	Which files are permanently locked on the desktop.

User Interface Guidelines

The default configuration of the desktop has been designed according to certain general principles in order to make the desktop more predictable, and therefore easier to learn and use.

Use the following principles when extending or altering the desktop for specific applications.

Icon Design

Standard desktop icons are designed within a 64x64 pixel square. Icons should include a border of at least one pixel to avoid icons bleeding onto a similarly shaded desktop pattern.

If shadows are included in icon designs, the shadow should appear to be cast by a light in the top left corner of the screen.

Icons should have the same scale. For example, a filing drawer should not be smaller than the folder it is supposed to contain.

Generic Icons

The icons should be consistent in appearance. Visual features should be used to convey information about the file that the icon represents. The three main categories of icons (directories, documents, and programs) should fall into three visibly consistent sets. For example, the standard document icons tend to be portrait in aspect ratio, and the directory icons landscape.

All file and document icons should be created in three forms to distinguish between read/write, read only, and locked.

Typical Applications

The following examples illustrate how the desktop can be configured for some practical applications.

Simulating Familiar Environments

Users who are already familiar with other desktop–based systems, or who have to share their work between the desktop and another desktop system, can configure the desktop to match the appearance and behavior of the other system. Thus they can minimize the errors associated with transfer and reduce the amount of relearning needed.

Associating data files with programs

By defining appropriate icon rules, each data file can be associated with the most relevant program, so that double–clicking on the data file invokes the appropriate program with the data file supplied to it.

Dragging a data file onto a program icon can be used as an alternative method of invoking the program. For example, a rule file could be written so that compiler source files were edited by dropping the icons which represent them on the Editor icon, and compiled by double–clicking on them.

Identifying Users' Files

Icon rules can also be used to change the foreground or background colors of the icons representing files according to who own the files. This would enable users to see at a glance which files belong to them.

Background Mail Server

The rule files are flexible enough to allow the creation of applications such as a mail server, which posts new mail as an icon on the desktop. The mail server would run as a background task, and when mail was received it would run a desktop command language command to place the mail file on the desktop. The rule file could specify that double–clicking on a mail file would open it in a text editor and delete the original file.

Trash Icon

The Trash Icon is created on the desktop using rule files, with no additional programming. The Trash Icon represents a directory, with an appropriate title and picture. It contains a rule file that causes any icon either dropped into its directory window or onto the directory icon to be moved to the directory. The **Trash** directory can be cleared by double–click on its icon with both mouse buttons.

The Trash icon would typically be locked onto the desktop by including it in the locked files list.

Automatic Encryption and Decryption

The drop rules allow directories with special characteristics to be created within the desktop, For example, a directory could be created such that all files dragged into it are encrypted. The user could decrypt them either by dragging them into another directory window, or by double–clicking on their icons.

Local Rule Files

Rule files can be local to one user, or even local to a specific directory. User–specific rule files can take care of the different computers that different users on an AIX network might be using. Each user can then double–click a program file on the network, and run it on a computer appropriate to that program.

Changing Environments

The desktop is flexible enough that a single user can switch environments at will, thus changing both the appearance and behavior of the desktop, by double–clicking on the appropriate environment file.

For example, a user might have two characteristic modes of working, programming and documenting. In the first mode, the desktop might display compilers and debuggers, and the rule files could specify that double–clicking on source files would run the appropriate compiler. In the documenting mode, the desktop might display word–processing and flow–charting programs, and double–clicking on source files would load them into the appropriate word processor.

The default rules accept any file whose name ends in the **.xde** suffix as an environment file. Double–clicking on an environment file with the left mouse button saves the current desktop state and changes the desktop to that environment. The default environment when the desktop is loaded is **xdtinitial.xde**.

How to Create and Edit AIXwindows Desktop Icons

Using a bitmap editor you can create new and unique icons for your application.

Prerequisite Conditions

- The AIXwindows desktop must be running on your terminal.
- You must have access to a bitmap editor that supports standard AIXwindows bitmaps, and have named it **bitmap**.

Procedure

1. Open the directory that contains your bitmap files. By default, this directory is named **/usr/include/X11/bitmaps**.
2. Copy an existing picture file, or the blank **blank.px** picture file.
3. Rename the new picture file to the new title, ending in the **.px** suffix.
4. Double–click with the left mouse button on the new icon. The desktop runs the bitmap editor on the new picture file.

How to Define the Appearance of AIXwindows Desktop Icons

Rule files describe how the icons and windows representing files, programs, and directories on the desktop appear and behave. The rules files are ASCII files and can be edited with any standard text editor.

Prerequisite Conditions

- The AIXwindows Desktop must be running on your terminal.
- The picture file to which you refer in defining the appearance of the icon must exist and be located in the path specified by the resources.

Procedure

1. To edit the rules in these files, drag the icon representing the rule file onto the icon representing the editor. By default, the first rule file that the desktop reads is **xdtinitial.xde** in your home directory. If the rule is to apply only to files within a certain directory, edit the file named **xdtdir.$LANG** (where **$LANG** is the value of your **LANG** environment variable, specifying the language for which you have National Language Support) within the appropriate directory. For example, if you use US English, you would edit the **xdtdir.En_US** file.
2. Create an **icon_rules** statement in the following format:

   icon_rules { [*FileSpec* { *FileRules* }] ... }

   where the **icon_rules** keyword introduces the rules, and the *FileSpec* parameter specifies the icons to which the rule apply. For example, if the rule is for the icon representing the file **compress**, the *FileSpec* parameter would be set to **compress.** The *FileSpec* parameter can also specify groups or classes of icons by including pattern matching characters or class specifications.

3. If the rule file already exists, incorporate the new icon rule into the rules the file already contains. Insert the icon rule in front of any other icon rules in the file so that it takes precedence over them.

Examples

1. Create file rules describing the appearance of the icon. For example, to set the title of the icon to **Squash**, use the following statement:

   ```
   title =Squash;
   ```

2. To use the **compress.px** picture file for the icon, use the following statement:

   ```
   picture = compress.px
   ```

   If this rule is in a directory's **xdtdir.$LANG** file, save the rule file and exit the editor, then close and reopen the directory window to see the change to the rule file take effect.

How to Define the Behavior of AIXwindows Desktop File and Program Icons

By editing rule files, you can specify the actions to be performed when the user double–clicks a mouse button while the mouse pointer is on an icon.

Prerequisite Conditions

- The AIXwindows Desktop must be installed on your system before editing rule files.

Procedure

1. To edit the rules in these files, drag the icon representing the rule file onto the icon representing the editor. By default, the first rule file that the desktop reads when it is run is the **xdtinitial.xde** file in your home directory. If the rule is to apply only to files within a certain directory, edit the file named **xdtdir.$LANG** (where **$LANG** is the value of your **LANG** environment variable, specifying the language for which you have National Language Support) within the appropriate directory. For example, if you use US English, you would edit the **xdtdir.En_US** file.

2. Create a **trigger_action** statement within the **icon_rules** statement for the icon, in the format:

   trigger_action: *TriggerID* { **actions** { *ActionList* } }

   The *TriggerID* parameter has the following values:

s1	Double–click on the left mouse button.
s2	Double–click on the right mouse button.
s3	Double–click on both mouse buttons.
s*	Double–click on any mouse button for which a rule has not been explicitly defined.
d1	Drag an icon onto this icon using the left mouse button.
d2	Drag an icon onto this icon using the right mouse button.
d3	Drag an icon onto this icon using both mouse buttons.

d* Drag an icon onto this icon using any mouse button for which a rule has not been explicitly defined.

The *ActionList* parameter specifies the commands that are run when the user double–clicks with the indicated button on the icon indicated in the *FileSpec* parameter of the **icon_rules** statement. Each statement in the list of actions is preceded by one of the following prefixes:

background or **b**	In background, by way of the standard AIX shell **/bin/sh**. Standard input, output, and error are the **/dev/null** device.
terminal_emulation or **t**	Standard AIX shell within the standard terminal emulator (normally the **aixterm** program.)
internal or **d**:	Statements in the desktop command language.
logged_background or **l**	Standard AIX shell with output sent to a log file.

The commands can also include substitutions that refer to components of named files.

Examples

1. The following rule:

```
icon_rules
{
*.Z /F
        {
        trigger_action : s1
                {
                actions
                        {
                        background : uncompress <%P0 >%D0/%E0
                        desktop : check %D0/%E0
                        }
                }
        }
}
```

indicates that, for all files whose names have the **.Z** suffix, double–clicking with the left mouse button on the file icon will run the AIX **uncompress** program on the file. The desktop then checks the icon for the resulting file.

The rule contains two actions statements, using substitutions to represent the appropriate icons and files:

- `background : uncompress <%P0 >%D0/%E0` runs the AIX **uncompress** program, with input from the file represented by the icon, where `%P0` represents its absolute pathname and output goes to the file in the same directory (as shown by `%D0`), with the same name as the input file, except for the last dot and any subsequent characters (as shown by `%E0`).
- `desktop : check %D0/%E0` runs the Desktop Command Language **chk** statement, ensuring that the correct icon displays for the output file, which is shown, as in the above statement, by `%D0/%E0`.

2. The rule:

```
icon_rules
{
compress /FX
        {
        trigger_action : d1
                {
                once_per_argument ;
                actions
                        {
                        background : %P0 <%P1 >%P1.Z ;
                        desktop : check %P1.Z ;
                        }
                }
        }
}
```

   indicates that, if the user drags any icons using the left mouse button onto the icon for the executable file named **compress**, each of the files those icons represents is compressed. The icon for the file being compressed is then replaced by the icon for the compressed file.

   The **actions** clause works as follows: the path name of the dragged file is substituted for `%P1`, and the first command in the action list runs **compress** (`%P0`) on this to create a file of the same name with the **.Z** suffix. The **check** desktop command then updates the new icon.

How to Define the Behavior of AIXwindows Desktop Directory Icons

By editing rule files, you can specify the actions to be performed when the user double–clicks a mouse button while the mouse pointer is on a directory icon or within a directory window.

Prerequisite Conditions

- The AIXwindows Desktop must be installed on your system.

Procedure

1. To edit the rules in these files, drag the icon representing the rule file onto the icon representing the editor. To create rules that apply to all directories, edit the desktop's initial rule file. (By default, this is the **xdtinitial.xde** file in your home directory). If the rule is to apply only to files within a certain directory, edit the file named **xdtdir.$LANG** (where **$LANG** is the value of your **LANG** environment variable, specifying the language for which you have National Language Support) within the appropriate directory. For example, if you use US English, you would edit the **xdtdir.En_US** file.
2. Create a **directory_rules** statement for the icon, in the following format:

   directory rules { **drop_in_action...** }

   Each **drop_in_action** is in the format:

   drop_in_action: *TriggerID* { [*IconGroup* { *ActionList* }] }

   The *TriggerID* parameter has the following values:

s1	Double–click on the left mouse button.
s2	Double–click on the right mouse button.
s3	Double–click on both mouse buttons.
s*	Double–click on any mouse button for which a rule has not been explicitly defined.
d1	Drag an icon onto this icon using the left mouse button.
d2	Drag an icon onto this icon using the right mouse button.
d3	Drag an icon onto this icon using both mouse buttons.
d*	Drag an icon onto this icon using any mouse button for which a rule has not been explicitly defined.

The *IconGroup* parameter determines how the rule is applied to its group of directories.

If the *IconGroup* parameter is set to the **act_on_all** value or the **ig=a** value (or if the parameter is not set), the actions are performed once. The `%P*` value is expanded to `%P1 %P2 ...` before the statement is executed.

If the *IconGroup* parameter is set to the **once_per_argument** value or the **ig=i** value, the actions are run once per icon. For each icon, the desktop substitutes the pathname of the file for the `%P*` value, then executes the statement once.

The *ActionList* parameter specifies the commands that are run when the user double–clicks with the indicated button on the icon indicated in the *FileSpec* parameter of the **icon_rules** statement. Each statement in the list of actions is preceded by one of the following prefixes:

background or **b**	In background, by way of the standard AIX shell **/bin/sh**. Standard input, output, and error are the **/dev/null** device.
terminal_emulation or **t**	Standard AIX shell within the standard terminal emulator (normally the **aixterm** program.)
internal or **d**:	Statements in the desktop command language.
logged_background or **l**	Standard AIX shell with output sent to a log file.

The commands can also include substitutions that refer to components of named files.

Examples

1. The rule:

```
directory_rules
{
        drop_in_action
        {
        dl
                {
                once_per_argument
                actions
                        {
                                desktop : move_into %P0 %P1
                                background : compress %P0/%B1
                                desktop : check %P0/%B1
                                desktop : check –R %P0/%B1
                        }
                }
        }
}
```

moves icons into the directory, then compresses the files and redisplays the icons for the compressed files.

The rule contains the following actions statements:

- `desktop : move_into %P0 %P1` uses the desktop command language statement **move_into** to move the file represented by the dropped icon (using the substitution %P1) into the directory represented by the directory window into which it was dropped (using the substitution %P0).
- `background : compress %P0/%B1` runs the AIX **compress** program on the file represented by the icon. The statement refers to the file's absolute pathname and filename by the substitution `%P0/%P1`.
- `desktop : check %P0/%B1` redraws the icon's image, if necessary.
- `desktop : check –R %P0/%B1` indicates that if the file no longer exists, the icon is removed.

The `once_per_argument option`, included in the list of files for the **drop_in_action** clause, specifies that the rules should be run with appropriate file substitutions for each icon dropped into the directory window. Otherwise the rules would be executed only once, even when several icons were dropped in.

2. The rule could also be specified for the directory's icon:

```
icon_rules
{
        compress /D
        {
                trigger_action : dl
                {
                        once_per_argument ;
                        actions
                        {
                        desktop : mvi %P0 %P1
                        background : compress %P0/%B1
                        desktop : chk %P0/%B1
                        desktop : chk –R %P0/%B1
                        }
```

```
                    }
            }
    }
```

How to Create and Edit AIXwindows Desktop Menus

By adding menu statements to the appropriate rule file you can customize the desktop menus to fit your applications.

Prerequisite Conditions

- The AIXwindows Desktop must be installed on your system.

Procedure

1. Create a menu statement within the appropriate rule file. To create rules that apply to all directories, edit the initial rule file. (By default, this is **xdtinitial.xde** in your home directory). If the rule is to apply only to files within a certain directory, edit the file named **xdtdir.$LANG** (where **$LANG** is the value of your **LANG** environment variable, specifying the language for which you have National Language Support) within the appropriate directory. For example, if you use US English, you would edit the **xdtdir.En_US** file.

   The menu statement has the following format:

   menu : *MenuName* { [**menu_item**] ... [**dividing_line**] ... }

   where *MenuName* is the name of the new menu.

2. The **menu_item** entries can include either actions to be performed or the name of a secondary menu that appears when the user selects the menu item.

   The **menu_item** statements for actions have the following format:

   menu_item : *ItemName* { [*ActionList*] ... }

   The *ItemName* shows the name of the menu item as it appears on the menu.

   The *ActionList* parameter specifies the commands that are run when the user double–clicks with the indicated button on the icon indicated in the *FileSpec* parameter of the **icon_rules** statement. Each statement in the list of actions is preceded by one of the following prefixes:

background or **b**	In background, by way of the standard AIX shell **/bin/sh**. Standard input, output, and error are the **/dev/null** device.
terminal_emulation or **t**	Standard AIX shell within the standard terminal emulator (normally the **aixterm** program.)
internal or **d**:	Statements in the desktop command language.
logged_background or **l**	Standard AIX shell with output sent to a log file.

Examples

1. The following statement:

   ```
   menu_item : Select All        {actions{d:select -A}}
   ```

   displays the `Select All` item name on the menu. When the user selects this item, the desktop runs the desktop command language select statement to mark all icons within the current window as selected.

2. To create **menu_item** statements for secondary menus, use the following format:

   menu_item : *ItemName* { [**pull_off_menu**] ... }

   The *ItemName* shows the name of the menu item as it appears on the menu.

   The **pull_off_menu** shows the menu name of the menu that appears beside the current menu when the menu item is selected.

   For example, the following statement:

   ```
   menu_item : Miscellaneous
       {pull_off_menu=pop_mainsub_misc_menu}
   ```

   displays the `Miscellaneous` item name on the menu. When the user selects this item, the desktop displays the menu named `pop_mainsub_misc_menu`.

3. To create dividing lines, use the following statement:

   ```
   dividing_line
   ```

   for thin dividing lines, or

   ```
   thick_dividing_line
   ```

   for thick dividing lines.

4. To create **menu_item** statements for labels, use the format:

   ```
   menu_item : item_name {}
   ```

   with nothing between the braces.

   ```
   menu : MainXdtMenu
   {
           menu_item : Desktop Menu {}
           thick_dividing_line;
           menu_item : Help        {actions{t:
                    pg /usr/lpp/xdt/xdesktop.man}}
           menu_item : Open directory...
                   {actions{b:%
                           %DIRNAME='mgti -name gtiopendir'
                           % %if [ "x$DIRNAME" != x ]
                           % %then
                           % %tellxdt display_directory $DIRNAME
                           % %fi}}
           menu_item : Close Directories
                   {actions{d:close_directories }}
           menu_item : File Info... {actions{b:mxdtinfo %P*}}
           dividing_line;
           menu_item : Select All          {actions{d:select -A}}
           menu_item : Deselect All        {actions{d:select -U}}
           menu_item : Put back            {actions{d:put_back %P*}}
           dividing_line;
           menu_item : Save Desktop...     {actions{b:%
                   %FILENAME='mgti -name gtisavedt'
                   % %if [ "x$FILENAME" != x ]
                   % %then
                   % %tellxdt save_environment $%{FILENAME%}.xde
                   %       %fi }}
           menu_item : Applications
                   {pull_off_menu=pop_mainsub_app_menu}
           menu_item : Miscellaneous
                   {pull_off_menu=pop_mainsub_misc_menu}
           dividing_line;
   ```

```
        menu_item : Exit  {actions{b:%
                %if myni -center "Close down X.desktop?" -helpMsg
"Click on Ok to close down X.desktop%#10# or Cancel to continue
with this session.
                % %then
                % %     tellxdt die
                % %fi }}
}
```

The *item_name* parameter shows the label as it appears on the menu. For example, the following rule file entry contains the following items:

Label	**Action**
Desktop Menu	None
	A thick dividing line.
Help	Runs the AIX pg program within a terminal window showing the **xdesktop.man** help file.
Open Directory...	Runs a shell script which opens a dialog window on the desktop, prompting the user to enter a directory name, then opens the indicated directory.
File Info	Displays information about the files represented by the selected icons.
Select All	Marks all icons on the desktop working surface as selected.
Deselect All	Marks all icons on the desktop working surface as unselected.
Put Back	Puts back the selected icons.
	A thin dividing line.
Save desktop...	Runs a shell script which opens a dialog window prompting the user for the name of a file in which it is to save the desktop environment, then saves the environment into that file.
Applications	Displays a secondary menu of applications. You can customize the list of applications which you want to have in this menu by creating an menu named **pop_mainsub_app_menu** within your **xdtuser.$LANG** rule file (where **$LANG** is the value of your **LANG** environment variable).
Miscellaneous	Displays a secondary menu of miscellaneous items.
Close Desktop...	Runs a shell script which opens a dialog window asking the user to confirm that the desktop is to close. If the user confirms the choice, ends the desktop program.

How to Configure the AIXwindows Desktop through Resources

You can configure many aspects of the appearance and functioning of the desktop by editing the resource values in the **.Xdefaults** file in your home directory or in the **/usr/lib/X11/$LANG/app–defaults/Xdesktop** file (where $LANG is the value of your LANG environment variable, referring to a language for which you have National Language Support). These are plain text files, and can be edited with any ASCII text editor.

Prerequisite Conditions

- The AIXwindows Desktop must be installed on your system.

Procedure

1. To put the new values into effect, save the file and exit the editor, and then restart the desktop.
2. To change the string with which the desktop resources begin (if, for example, they conflict with another application that also begins its resources with the **Xdesktop*** string), edit the value of the **Xdesktop*name** resource. Since the desktop reads the value of this resource first to determine the name under which it looks up the other values, the value of the **Xdesktop*name** resource can affect the names of all desktop resources with the single exception of the **Xdesktop*name** resource.

Examples

1. The resource values for the desktop begin with the **Xdesktop*** string. For example, the following resource entry:

   ```
   Xdesktop*iconGrid.spacing:      100
   ```

   indicates that the desktop, in tidying the placement of icons, places them on the screen so that the center of each icon is100 pixels away from the center of the next icon.
2. To set the spacing so that the icons are twice as far apart, change the resource entry to:

   ```
   Xdesktop*iconGrid.spacing:      200
   ```

How to Create Messages and Dialogs through AIXwindows Desktop Utilities

Several utilities are included with the AIXwindows Desktop with which you can display messages and create dialog windows from within shell scripts and rule file statements.

Prerequisite Conditions

- The AIXwindows Desktop must be installed in your system.
- The utilities must be installed within directories in your path.

Procedure

If each utility is in your path, you can run each of them from within shell scripts or from the command by typing the command name and any arguments. You can include them in actions statements within rule files in the following format:

{ **actions** { **background** : *command* [*arguments*] ... } }

Examples

1. To display a one line message in a window on the desktop, use the **mfyi** command. For example, the following statement:

   ```
   mfyi –picture mailbox.px You have new mail.
   ```

   displays a window on the screen, displaying a mailbox icon from the bitmap file and the message "`You have new mail`". The user can remove the window by clicking on the displayed icon.

2. To display a simple question in a window on the desktop and prompt the user to cancel or confirm the answer, use the **myni** command. For example, the following statement:

   ```
   myni –picture question.px Delete file xyz?
   ```

   opens a dialog window with:

   - The icon contained in the `question.px` bit map file
   - The question: `Delete file xyz?`

   If the user clicks on the confirm button, the **myni** utility returns a code of 0. If the user clicks on the cancel button, the **myni** utility returns a code of 2.

3. To display a message in a window on the desktop and prompt the user for a response, use the **mgti** command. For example, the following statement:

   ```
   mfyi "What is the name of the file?"
   ```

   displays a window on the screen, prompting the user to enter the name of a file. The **mgti** command passes the user's response to standard output.

4. To send commands in the desktop command language to a desktop, use the **tellxdt** command. For example, the following command:

   ```
   tellxdt get_out_icon /bin/xclock
   ```

   places the clock's icon on the desktop.

   You can also use the **tellxdt** command, with various arguments, to find out information about a file or its icon. For example, the following command:

   ```
   tellxdt –t testcode.c
   ```

   returns the name of the picture file used for the icon for the **testcode.c** file

   If the desktop command which **tellxdt** is executing uses a flag which **tellxdt** also uses, place the entire desktop command in quotation marks.

How to Change the Fonts Displayed on the AIXwindows Desktop

The desktop determines the font with which it displays text for its icons and windows by reading the value of the **Xdesktop*font** and **Xdesktop*fontList** resources.

Prerequisite Tasks or Conditions

- The AIXwindows Desktop must be available on your system.
- You must have an ASCII editor, such as **vi**.

Procedure

1. To change the font for your own use, edit the **.Xdefaults** file within your home directory with any ASCII editor.
2. The **/usr/lpp/X11/defaults/Xfonts** file contains a list of valid fonts names, along with the information that AIXwindows uses in displaying them. A sample fonts file contains:

```
1    Rom14.500 0 0 9    20
2    Itl14.500 0 2 9    20
3    Bld14.500 0 1 9    20
4    Rom10.500 0 0 8    14
5    Rom6.500  0 0 4    8
6    Rom22.500 0 0 12   30
7    Rom29.500 0 0 18   40
8    Erg14.500 0 0 9    20
9    Rom7.500  0 0 6    9
10   Rom8.500  0 0 6    11
11   Rom11.500 0 0 7    15
12   Rom16.500 0 0 7    22
13   Rom17.500 0 0 11   23
14   Bld17.500 0 1 11   23
```

3. To set the font used on the desktop to one of the defined fonts, enter a value from the second column of the table. For example, to use a 14.5 point Roman typeface, set the values in the **.Xdefaults** file to:

```
Xdefaults*font:          Rom14.500
Xdefaults*fontList:      Rom14.500
```

4. For the value to take effect, save the file, then exit the desktop and restart it.

How to Set the Colors and Bitmapped Images for the Desktop

Procedure

1. To change these colors, edit the **.Xdefaults** file within your home directory with any ASCII editor.
2. The resources that affect the desktop colors and bitmapped images are in the form:

   Xdesktop* *item* *feature*

   The *item* parameter can have the following values:

newIcon	Icons for new files or directories.
desktopIcon	Icon which identifies the Desktop within the AIXwindows window manager icon box.
normal	Text windows.
message.fatal	Messages that appear when programs end unexpectedly.
message.alert	Warning messages.
message.fyi	Informative messages.
message.greeting	The message that appears when the Desktop begins.
textButton	The button which appears next to the names of text files when the Desktop displays files in name mode.

The *feature* parameter can have the following values:

foreground	Foreground color.
background	Background color.
pixmap	The bitmap image used for the display.

The **/usr/lpp/X11/rgb/rgb.txt** file contains a listing of valid color names, and the information AIXwindows uses in displaying them on the screen. For example, a section of an **rgb.txt** file might contain:

```
112 219 147        aquamarine
112 219 147        Aquamarine
50 204 153         medium aquamarine
50 204 153         MediumAquamarine
0 0 0              black
0 0 0              Black
0 0 255            blue
0 0 255            Blue
95 159 159         cadet blue
95 159 159         CadetBlue
66 66 111          cornflower blue
66 66 111          CornflowerBlue
107 35 142         dark slate blue
107 35 142         DarkSlateBlue
```

To set the background color of the icon for a newly created file to cadet blue, for example, edit its resource entry in the **.Xdefaults** file to read:

```
Xdesktop*newIcon.background: cadet blue
```

3. The **/usr/include/X11/bitmaps** directory contains bitmap files, some of which are suitable for use as desktop pixmaps. Your desktop may use another directory for its pixmaps, as indicated in the **Xdesktop*picture.directory** resource. To indicate that warning messages, for example, are to use the warning.px bitmap file, edit its resource entry to read:

```
Xdesktop*message.alert.pixmap: warning.px
```

4. For the value to take effect, save the file, then exit the desktop and restart it.

Using AIXwindows Desktop Rule Files

This chapter describes how to create and edit rule files for the AIXwindows Desktop and provides information on rule file location, structure, contents, interpretation, and substitutions.

Rule files specify the appearance and behavior of the icons representing files on the desktop. For example, rule files enable you to

- Associate an icon image with specific files or groups of files
- Associate mouse actions with specific functions to be performed
- Define events that occur when one icon is dropped onto another.

Rule files are text files, and can be edited with an ASCII text editor such as the **vi** editor.

Rule files consist of sequences of clauses. Each clause has one of two forms: either

keyword = *value*;

OR

keyword { *body* }[;]

In the second form the final semicolon is optional. The body section is normally a sequence of additional clauses, separated by semicolons.

In general, the layout of rule files is not critical, and spaces or new lines can be inserted to improve readability and make the structure of the rules clearer.

For example, the following rule:

```
icon_rules {* /D{picture=dir.px;} * /F{picture=file.px;}}
```

is equivalent to:

```
icon_rules
        {
        * /D
                {
                picture = dir.px;
                }
        * /F
                {
                picture = file.px;
                }
        }
```

Components of Rule Files

A rule file consists of a number of text components. These components can occur any number of times in any order:

Component	Description
Icon rules	Describe which bitmap file contains the image of each icon, what the icon title should be, and what occurs when a user triggers the icon by double–clicking on it, or drags another icon onto it.
Directory rules	Describe what happens when icons are dragged into directory windows.
Menu rules	Define the title and items of a menu and describe what happens when any menu item is selected.
Desktop layout	Lists all icons that appear on the desktop and describes their positions. The desktop layout is generated automatically.
Locked files	Indicate which files are locked onto the desktop and cannot be put back.

Location of Rule Files

Rule files have the following precedence:

1. **Local rule file:** Placed in any directory. They provide rules that apply only to the files in that directory.

   The **directoryRuleFile** resource determines the name of this file. If it is not specified, the file's name is **xdtdir.$LANG**, where **$LANG** is the value of the user's **LANG** environment variable showing a language code for National Language Support. The value usually used in the United States is **xdtdir.En_US.**

2. **Environment rule files:** Contain rules and a list of files that are on the desktop. By convention, the names of these files have the form ***.xde.**

   Each user has only one active environment rule file at a time. This file is known as the *current environment*. By switching environment rules as they switch environments, users can have different rules for different circumstances. For example, a user might work in a programming environment and a text editing environment.

   When a user changes environment, the positions of icons currently on the desktop are saved in the old environment rule file, and a new set of positions is read from the new environment file to create a new desktop.

3. **User rule file:** Contains rules that apply only to an individual user.

   The **userRuleFile** resource determines the name of this file. If it is not specified, the file's name is **.xdtuserinfo.**

4. **System rule file:** Applies to all desktops running on a given machine. There is only one such file. It should be changed only by the root user.

   The **systemRuleFile** resource determines the name of this file. If it is not specified, the file's name is **/usr/lpp/xdt/lpp.Sysinfo/$LANG** is the value of the user's **LANG** environment variable showing a language code for National Language Support.

   The **icon_rules** and **directory_rules** statements which use *Pattern* clauses that refer to files by their absolute pathnames override other statements which refer to the files by relative pathnames, regardless of the priority of the files that contain them. Thus, for example, a system rule file which referred to **/usr/include/X11/bitmaps/foo.px** would override, for that file, a local rule file in **/usr/include/X11/bitmaps** which would refer to ***.px.**

Keywords

Rules in rule files are introduced by the following keywords. AIXwindow Desktop Rule File Keywords describes the keywords in detail.

Keyword	**Description**
actions	Specifies the commands that the AIXwindows desktop runs when an icon or directory window is triggered, or a menu item is selected.
desktop_layout	Describes the files that are on the desktop, together with their positions.
directory_rules	Describes the actions to be performed when specified triggers occur within directory windows.
drop_in_action	Defines the effect of dropping one or more icons into a directory window.
icon_rules	Describes the behavior and appearance of files when represented by icons.
locked_on_desktop	Specifies the icons to be locked on the desktop.
menu	Defines the items on menus and the actions to be carried out when a particular item is selected.
menu_item	Specifies an item on a menu and the action to be performed when the item is selected. Used only within menu clauses.
picture	Specifies the bitmap file to be used for the icons for the associated files.

pull_off_menu	Specifies a menu to pop up beside the current menu when the associated menu item is selected.
title	Assigns a title to the specified files.
trigger_action	Specifies the actions to be carried out when the icon is selected using the specified trigger.

Special Characters

The following characters have special meanings in rule files:

Name	**Special Meaning**
% percent	Preprocessor command (Escape Sequences)
{ open brace	Group items
} close brace	End group
; semicolon	Separator
: colon	Separator
= equal sign	Separator
& ampersand	Separator.

When a semicolon is followed by a close brace it can be omitted.

Preprocessor Commands (%)

The percent symbol is considered an *escape character* in rule files. It allows special phrases and instructions to be included in rule files. Each character string preceded by a % character is replaced by another character string at a preprocessing stage (the % character was chosen to be distinct from escape characters used for this purpose in AIX).

Sequence	**Replaced by**
%%	%
%;	;
%{	{
%)	}
%:	:
%=	=
%!c	The **c** character.
%#*n*#	The *n* character code, according to C language number conventions. The *n* value cannot be 0.
%white%	Sequences of white space characters (space, tab, or new line) surrounded by the % character are removed.
%//	Used for comments; characters up to the next new line are ignored.
%/*	Used for comments; characters up to the next %*/ string are ignored.
%*/	Ends comments.
%$*name*$	The contents of the named environment variable replace the sequence. (not preprocessed).

%&*name*&	The contents of the named environment variable replace the sequence. The value is preprocessed after it is replaced.
%+*file*+	The preprocessed contents of the specified file replace the sequence.

Other characters can be included by preceding them with the `%!` escape sequence.

Comments

Comments can be included in a rule file by prefixing them with the `%//` sequence, which causes all characters up to the end of the line to be ignored, as follows:

```
icon_rules { * { title =Title;} } %// This title is for all icons.
```

Long comments can be preceded by the `%/*` sequence and followed by the `$*/` sequence. All characters between these two sequences are ignored.

Control Characters

You can include a control code or other special character in a rule file using the following sequence:

%# *decimal–code* **#**

%#O *octal–code* **#**

%#0x *hexadecimal–code* **#**

%#0X *hexadecimal–code* **#**

Note that in the above hexadecimal sequences the **0** character, not the letter **O**, is required.

For example, the double–quote character, character code 34 (that can also be represented as octal 42 and hexadecimal 22) can be written as follows:

%#34#

%#O42#

%#Ox22#

%#OX22#

Including Files

Rule files can include other named rule files by specifying the sequence, as follows:

```
%+name+
```

Path names which do not begin with a slash are relative to the following directories, as determined by the rule file being processed:

Rule File	Directory for relative includes
xdtsysinfo	/usr/lpp/xdt/lpp.sysinfo
xdtuserinfo	$HOME
environment	$HOME
.xdtdirinfo	The directory which contains the file

Including Environment Variables

The contents of an environment variable can be included in a rule file with the following sequences:

%&name&	for which the characters in the environment variables are preprocessed.

%$name$	for which none of the characters in the environment variables are preprocessed.

Referring to File Names

Rule files can refer to files in four ways:

Absolute path name	The full name of the file. It begins with a slash and contains all the directories leading to the rule file.
Base name	The name of the file within its directory. It is the part of the absolute path name following the last slash.
Directory name	The name of the directory holding the file. It is the part of the absolute path name preceding the last slash.
Relative path	If the file is in the **$HOME** directory, its relative path is the same as its base name. If the file is in a subdirectory of **$HOME**, the relative path name is the path from the working directory to the file. Otherwise it is the same as the absolute path name.

For example, the various names of the **/user/fred/work/letter** file are:

Absolute path name	**/user/fred/work/letter**
Base name	**letter**
Dir name	**/user/fred/work**

The relative path name depends on the value of **$HOME**:

$HOME	**Relative path name**
/user	**fred/work/letter**
/user/fred	**work/letter**
/user/fred/work	**letter**
any other	**/user/fred/work/letter**

There is one exception: The directory name reference for the root directory, **/**, is **/.** (slash dot) and its base name is **/** (slash).

Substitutions

Rule files use *substitutions* to refer to file names that correspond to icons that are triggered or dropped.

The name of a file can be substituted for the title of its icon, and the names of any of the files involved can be substituted into commands in **actions_of** statements within rule files.

Substitutions are made by placing a substitution sequence in the rule file at the appropriate point. A sequence consists of the following:

- percent sign (%),
- letter indicating the information to be substituted, and
- In some cases, a number or an asterisk (*).

The following letters must be followed by a number or an asterisk:

Letter	**Meaning**
B	Base name of the file

C	Class of the file (as used by the **icon_rules** keyword), in capital letters.
D	The name of the directory holding the file.
E	Base name (as for **B**), with the last dot and any subsequent characters removed.
R	If the file is in a directory under **$HOME**, the path name of the file relative to **$HOME**. If the file is not in a directory under **$HOME**, the absolute path name of the file. In the titles of icons within directory windows, the base name of the file.
P	Absolute path name of the file

The **number** or **asterisk** determines the file or files for which information is to be substituted. The permitted cases are as follows:

- In **picture** or **title** statements, the number **0** indicates the file represented by the icon.
- In **actions_of** statements within **trigger_action** statements for static triggers, the number **0** represents the file represented by the icon on which the user had clicked.
- In **actions_of** statements within dynamic trigger **trigger_action** clauses, or within **drop_in_action** statements:
- The number **0** indicates the file represented by the icon onto which the user had dropped other icons.
- The numbers **1** through **9** represent the appropriate file in the list of files dropped onto the icon. If the number is larger than the number of files dropped onto the icon, a null value is substituted.
- An asterisk (*) represents the files of each of the icons dropped in turn. The values are separated by spaces. For example, if three icons are dropped, then the **%P*** sequence is the same as the **%P1 %P2 %P3** sequence.

For example, the **/fred/jim/data.tmp** file is substituted as follows:

Sequence	**Substitution**
%P0	**/fred/jim/data.tmp**
%B0	**data.tmp**
%E0	**data**
%D0	**/fred/jim**
%R0	**/fred/jim/data.tmp**
%C0	**FXWM (for example)**

The following letters are not followed by numbers or asterisks:

Letter	**Meaning**
N	In **picture** or **title** statements, and in **actions_of** statements within **trigger_action** statements for static triggers, the number **0**. In **actions_of** statements within dynamic trigger **trigger_action** clauses, or within **drop_in_action** statements, the number of files represented by the icons which were dropped.
T	In **picture** or **title** statements, null. In **actions_of** statements within **trigger_action** statements for static triggers, the static trigger identifier. In

	actions_of statements within dynamic trigger **trigger_action** clauses, or within **drop_in_action** statements, the dynamic trigger identifier.
X	The name of the version of the Desktop.

AIXwindows Desktop Rule File Keywords

The AIXwindows keywords are used to control actions in AIXwindows Desktop rule files. The AIXwindows keywords are:

- **actions** or **ac**
- **desktop_layout** or **dt**
- **directory_rules** or **dd**
- **drop_in_action** or **da**
- **icon_rules or ic**
- **locked_on_desktop** or **lf**
- **menu** or **me**
- **menu_item** or **mi**
- **picture** or **pi**
- **pull_off_menu** or **pm**
- **title** or **ti**
- **trigger_action** or **ta**

actions or ac Keyword

Purpose

Specifies the commands that the AIXwindows desktop runs when an icon or directory window is triggered, or a menu item is selected.

Syntax

actions { [*Control* : *Command* ;] ... }

Description

actions statements can be empty or contain one or more commands separated by semicolons. Each command is preceded by a command identifier or letter specifying how the command is to be run.

Parameters

Control Specifies how the command is to be run:

background or **b**	In background, by way of the standard AIX shell **/bin/sh**. Standard input, output, and error are the **/dev/null** device.
terminal_emulation or **t**	Standard AIX shell within the standard terminal emulator (normally the **aixterm** program.)
internal or **d**:	Statements in the desktop command language.
logged_background or **l**	Standard AIX shell with output sent to a log file.
Command	Specifies the command to be run. If the *Control* parameter is set to the **d** value, the command is a statement in the desktop command language; if the *Control* parameter is set to another value, the command is an AIX command.

Example

The following actions statement contains three commands.

```
actions
        {
        terminal_emulation : vi myfile.1;
        background : troff <myfile.1 >myfile.1.trf;
        internal : chk myfile.1.trf
        }
```

These commands are run in order, with each being carried out after the previous one has finished. However, while the desktop is waiting for one command to finish running, it may be doing other things, and several action lists can be running at the same time.

desktop_layout or dt Keyword

Purpose

Describes the files that are on the desktop, together with their positions.

Syntax

desktop_layout { [*FileName* [@*Position*] ;] ... }

Description

The desktop layout is normally generated automatically by the desktop, but could be modified by a program to alter the initial appearance and layout of a desktop. Desktop layout lists that do not appear in environment files are ignored.

Parameters

FileName	specifies the name of a file to be included on the desktop.
Position	Specifies, if present, the position of the icon on the desktop, as indicated by the following values:
G*x,y*	The **G** value followed by two numbers separated by a comma, shows the x and y coordinates of the icon in grid units (as determined by the **xdt*iconGrid.spacing** resource).
P*x,y*	The **P** value followed by two numbers separated by a comma, shows the x and y coordinates of the icon in pixels.
F	Indicates that the icon is to be placed at the first free position of the grid.

If the *Position* parameter is omitted, the icon is placed at the first free position of the grid.

Example

The following example shows the positions of the icons representing several files and directories on a desktop:

```
desktop_layout
        {
        /               @       G       0       0 ;
        /usr/bin        @       G       1,      0 ;
        /fred           @       F                 ;
        /fred/main      @       G       4,      7 ;
        /bin            @       P       211,    874 ;
        /fred/data
        }
```

The positions specified in the example are described below:

File or Directory **Position**

/	Upper left corner of the grid
/usr/bin	To the right of the icon for the root directory,
/fred	At the first free position of the grid (to the right of the icon for **/usr/bin**)
/fred/main	On the grid, four positions to the right and seven rows down from the upper left corner of the directory window.
/bin	At a position 211 pixels to the right and 874 pixels down from the upper left corner of the directory window.
/fred/data	At the first free position on the desktop.

directory_rules or dd Keyword

Purpose

Specifies the actions to be performed when specified triggers occur within directory windows.

Syntax

directory_rules { [*drop_in_action*] ... }

Description

A **directory_rules** clause within the rule file for a directory refers to that directory, as if it had been specified in an appropriate file clause in the parent directory's rule file. The entries in a **directory_rules** clause in any other rule file apply to all directories, as if they had been specified in an **icon_rules** clause with the * pattern and the **D** class.

For example, the **directory_rules** clause:

```
directory_rules {
        drop_in_action : d1 {
                actions {
                        desktop : move_into %P0 %P*
                        }
                }
        drop_in_action : d2 {
                actions {
                        desktop : copy_into %P0 %P*
                        }
                }
        drop_in_action : d3 {
                actions {
                        desktop : link_into %P0 %P*
                        }
                }
        }
```

has these effects:

- If the user drags icons into the directory window using the left mouse button, the files which the icons represent are moved into the directory. (The substitution `%P0` represents the absolute pathname of the directory; the substitution `%P*` represents the absolute pathnames of each of the icons which are dragged into the directory window.)
- If the user drags icons into the directory window using the right mouse button, the files which the icons represent are copied into the directory.
- If the user drags icons into the directory window using both mouse buttons, the files which the icons represent are copied into the directory.

drop_in_action or da Keyword

Purpose

Defines the effect of dropping one or more icons into a directory window.

Syntax

drop_in_action : *Trigger* { [*actions*] ... [*IconGroup*] ... }

Description

Parameters

Trigger	Specifies the trigger action for which the actions are performed.
Actions	Specifies the actions to be performed.
IconGroup	Specifies how many times the action is to be performed.If the *IconGroup* parameter is set to the **act_on_all** value or the **ig=a** value (or if the parameter is not set), the actions are performed once.If the *IconGroup* parameter is set to the **once_per_argument** value or the **ig=i** value, the actions are run once per icon.

Example

The following directory rules cause dragging an icon into a directory window to copy or move the file, depending on whether the left or the right mouse button is used. In each case the %P1 sequence is replaced by the path name of the file dropped, and the %P0 sequence by the path name of the directory window into which it was dropped.

```
directory_rules
        {
        drop_in_action : d1
                {
                once_per_argument;
                actions {
                        desktop : copy_into %P0 %P1
                        }
                }
        drop_in_action : d2
                {
                once_per_argument;
                actions {
                        desktop : move_into %P0 %P1
                        }
                }
        }
```

An important difference to note between the icon group options is the action taken when several icons are triggered at the same time. With the **once_per_argument (ig=i)** value, the action list is duplicated several times, one copy for each icon dropped, and then each copy is modified by the substitution system to include details about that icon.

Suppose the action list is as follows:

```
actions
        {
        d : mvi /waste %P1      ;
        b : echo %P1 >>/waste/.filelist
        }
```

and the **/fred/fred**, **/fred/jim**, and **/fred/sheila** icons are dropped. Because the `%P1` sequence is replaced by the path name of the icon dropped, the action list actually run is:

```
actions
        {
        desktop : move_into /waste /fred/fred ;
        background : echo /fred/fred >>/waste/.filelist
        desktop : move_into /waste /fred/jim ;
        background : echo /fred/jim >>/waste/.filelist
        desktop : move_into /waste /fred/sheila ;
        background : echo /fred/sheila >>/waste/.filelist
        }
```

icon_rules or ic Keyword

Purpose

Describes the behavior and appearance of files when represented by icons.

Syntax

icon_rules { *Pattern* [/ *Class*] { [*Picture*] [*Title*] [*trigger_action*] ... [*drop_in_action*] ... } }

Description

The *title*, *trigger_action*, and *drop_in_action* parameters may appear in any order.

Patterns are file names, optionally containing certain reserved or pattern matching characters according to standard AIX pattern matching syntax:

- The ? character specifies any single character. For example, the **a?c** pattern includes the **abc** and **aac** files, but not the **abbc** file.
- The * character specifies any sequence of characters within file names, including none. For example, the **a*c** pattern includes the **ac**, **abc**, **acbc**, and **as4hx..06f,s:c** files.
- Placing characters within brackets, as in the **[abc]** pattern, indicates any one of the specified set of characters.

  The set of characters can be abbreviated using the – (minus sign) character to represent a range; for example, the **A–D** pattern is equivalent to the **ABCD** pattern. Ranges should only be between two letters of the same case, or two digits.

  Prefixing the set with a ! (exclamation point) specifies any characters not included within the set. For example, the **[!A–Za–x]*** pattern means any file name beginning with a character other than a letter.
- The / (slash) special file name can appear as a file specification by writing it as `/ /d` (files called slash that are directories).

Classes represent the properties of files in a concise form. These properties fall into four sets as follows:

- File type
- Execute permission
- Read-write permissions
- Ownership

A file has exactly one property of each set. Each property is represented by a letter (of either case).

The meanings of the class letters are as follows:

File type

B	Block special file.
C	Character special file.
D	Directory.
F	Regular file.
G	Ghost (non–existent) file. (Implies the H, N, and O classes, described below).
I	Inaccessible file. (Implies the H, N, and O classes, described below)
P	Pipeline.
S	Symbolic link to a non–existent file. (Implies the H, N, and O classes, described below).

Execute Permissions for Files

X	User can run the file.
A	File has run permission for someone, but not for the user.
N	No one can run the file

Execute Permissions for Directories

X	User can list the contents of the directory.
A	Someone, but not the user, can list the contents of the directory.
N	No one can can list the contents of the directory.

Read/Write Permissions for Files

W	User can read and write the file.
V	User can read but not write the file, and the file has write permission for someone.
K	User can read the file, and no one can write to the file.
H	User can not read the file.

Read/Write Permissions for Directories

W	User can read from and write to the directory.
V	User can read from but not write to the directory, and the directory has write permission for someone.
K	User can read from the directory, and no one can write to it.
H	User can not read from the directory.

Ownership

M	User owns the file.
O	User does not own the file.

The class strings are interpreted according to the following rules:

- The order of letters is not significant.
- Redundant letters are ignored.

- If multiple letters from a set appear, the class includes files whose properties match any of the classes. For example, the BCNWVO class is interpreted as the '(B or C) and N and (W or V) and O' classes.
- If no letters of a set appear, the class is interpreted as if they all appeared. For example, class `D` is the same as `DXANWVKHMO` (both mean 'all directories').

A number of other codes can also be used in classes. Each stands for a common combination of the standard codes:

Q	G, I, or S (not a real file).
E	X or A (can be run by somebody).
U	A or N (can not be run by the user).
L	V or K (can be read but not written by the user)..
R	W or L (can be read by the user).

For example, the DEO and DAXO classes have the same meaning.

The following classes are the most useful in rule files:

D	Directories.
F	Files.
FE	Executable files.
FX	Files executable by the user.
FN	Data files.
FNW	Data files that the user can alter.
FNT	Data files that the user can read..

Directory rules in local rules files apply to the directory window of the directory holding the rule files. Directory rules in other rule files apply to all directories.

Parameters

Pattern	Specifies the file names to which the subsequent rules apply.
Class	Specifies a file or group of files to which the subsequent rules apply. It is usually clearer to group all the rules that apply to one group of files together. The pattern and class definitions must be separated by a space.
Picture	Specifies the name of the bitmap file to be used for the icon. There should be at most one *Picture* parameter.
Title	Specifies the title text displayed for the icon.
Trigger_action	Specifies actions clauses to be performed when the user triggers the icon.
Drop_in_action	Specifies *actions* clauses to be performed when the user drops other icons into a directory with the same name as this icon.

Examples

1. The following example defines icons and titles for the calculator, clock, and editor programs, and causes the editor program to be run for any file icons dragged onto its icon, with those files loaded in ready for editing:

```
icon_rules
        {
        xcalc { title =Calculator; picture = xcalc.px }
        xclock { title =Clock; picture = xclock.px }
        xedit
                {
                title =XEdit;
                picture = quill.px;
                trigger_action : s1 {}
                trigger_action : s2 {}
                trigger_action : d2 {}
                trigger_action : d1 {
                        actions { background : %P0 %P1 };
                                once_per_argument }
                }
        }
```

2. The next example defines a rule that moves any file or files dropped onto a directory icon with the left mouse button into that directory:

```
icon_rules
        { */D
                { trigger_action : d1
                        {
                        actions {
                                desktop : move_icon %P0 %P*} ;
                        act_on_all
                        }
                }
        }
```

3. The final example shows a section of the standard definition of the icon rules for the Trash icon. The Trash icon is implemented as a directory with a suitable picture and title.

   For convenience, dragging with the left mouse button moves a file to the **Trash** directory, rather than copying it as the usual default.

   The **Trash** directory can be emptied by double clicking with both mouse buttons. The **rm –rf %P0/*** command deletes all files in the directory.

```
icon_rules
        {
        waste /D
                {
                title =Trash;
                picture =waste.px;
                trigger_action : s3 { actions { background :
                                          rm –r %P0 } }
                }
        }
```

locked_on_desktop or lf Keyword

Purpose

Specifies the icons to be locked on the desktop.

Syntax

locked_on_desktop { [*FileName* ;] ... }

Description

Locked files appear on the desktop. The user cannot put them back into the directories from which they came.

Locked file lists in the system and user rule files apply whenever the AIXwindows desktop is run. A locked file list in an environment file applies only while that environment file is current. Locked file lists are ignored in local rule files.

Example

```
locked_on_desktop
        {
        /           ;
        /bin        ;
        /usr/bin    ;
        }
```

menu or me Keyword

Purpose

Defines the items on menus and the actions to be carried out when a particular item is selected.

Syntax

menu : *Menu_Name* { [*Menu_Item*] ... [*Dividing_ Line*] ... }

Parameters

Menu_name	Specifies the menu name.

Dividing_line	Displays a thick or thin dividing line in place of a menu item on the menu when included in a menu rule. The dividing line cannot be selected. Two options are available as follows:
	`dividing_line (dv=n)`
	`thick_dividing_line (dv=t)`

Example

The following example defines the clean_up menu which puts away selected icons from the desktop and closes selected windows:

```
menu : clean_up
        {
        menu_item : put away icons
                {
                actions { d : put_back_icon %P* }
                }
        dividing_line
        menu_item : close windows
                {
                actions { d : cdw %P* }
                }
        }
```

menu_item or mi Keyword

Purpose

Specifies an item on a menu and the action to be performed when the item is selected. Used only within menu clauses.

Syntax

menu_item : *ItemName* { [*Actions*] }

menu_item : *ItemName* { [*Pull_off_menu*] }

Parameters

ItemName	Specifies the name of the menu item as it appears on the menu.
Actions	Specifies actions to be performed when the item is selected.
Pull_off_menu	Specifies a subsidiary menu to appear beside the current menu.

picture or pi Keyword

Purpose

Specifies the bit mapped picture file to be used for the icons for the associated files.

Syntax

picture = *picture_file*

Description

The picture file usually has a **.px** file extension. For example the following means that all directories are to use the picture in the picture file **dir.px**.:

```
*/D { picture = dir.px }
```

If no picture file is specified, the icon is invisible.

pull_off_menu or pm Keyword

Purpose

Specifies a menu to appear beside the current menu when the associated menu item is selected.

Syntax

pull_off_menu = *Menu_Name*

Parameter

Menu_Name	Specifies the name of the new menu. It must correspond to a defined menu.

title or ti Keyword

Purpose

Assigns a title to the specified icons.

Syntax

title =*Name*

Description

All spaces are significant between the equals sign and the semicolon or closing bracket. Substitutions can be used to include the actual file name in the title.

Examples

1. The following clause:

   ```
   xclock { title =Clock}
   ```

   sets the title of the **xclock** program.

2. If necessary for formatting purposes, new lines, spaces, and tabs can be included before and after the icon title. Surrounding unwanted white space with a pair of percent signs ensures that it is ignored. The following rule:

   ```
   icon_rules { * { title =Title for all icons} }
   ```

   is equivalent to:

```
icon_rules { * { title =Title %
                 %for all icons} }
```

trigger_action or ta Keyword

Purpose

Specifies the actions to be carried out when a specified trigger occurs with the mouse pointing to the icon.

Syntax

trigger_action : *Trigger_ID* { [*actions*] [*Icon_Group*] }

Parameters

Trigger_ID	Specifies the trigger for which this action is performed.
actions	Specifies the actions to be performed. Each trigger_action clause can contain at most one *actions* parameter.
icon group	Specifies how many times the action is to be performed. Each trigger_action clause can contain at most one *icon_group* parameter. If the *icon_Group* parameter is set to the **act_on_all** or **ig=a** values (or if the parameter is not set), the actions are performed once.If the icon group parameter is set to the **once_per–argument** or **ig=**i values, the actions are run once per icon.

Examples

1. The following example:

   ```
   trigger_action : s1 { actions { b : xclock } }
   ```

   shows that the s1 trigger–id (a double click with the left mouse button) runs the **xclock** command.

   If the body is blank, the trigger has no effect.

2. The following example:

   ```
   * / D { trigger_action : s3 {} }
   ```

   shows that the s3 trigger–id (a double click with both mouse buttons) on a directory does nothing.

3. The following example:

   ```
   icon_rules
           {
           *.c     { picture = csrc.px; }
           * / D
                   {
                   picture = dir.px;
                   trigger_action : s1 { ac { d :  ddw %P0 t  } }
                   }
           [ab]* { title =AB file %B0 }
           }
   ```

   has these effects:

   - all files whose names end in the **.c** extension use the picture found in the **csrc.px** bitmap file.
   - all directories use the picture found in the **dir.px** bitmap file.
   - when any directory is triggered with the s1 trigger, a directory window is opened to show that directory in time order.

- any file whose name begins with the lowercase letters `a` or `b` has the `AB file` icon title followed by the base name of the file.

Using Picture Files with the AIXwindows Desktop

Rule files can refer to pictures either through absolute path names or relative path names. For relative path names, the desktop first examines the picture directory (as indicated by in the **.Xdefaults** mechanism), and then it examines the **/usr/include/X11/bitmaps** directory.

For example, if the picture directory is set to the **/user/fred/pictures** directory and you are trying to find the **core.pic** picture file. The desktop looks for the following files:

/user/fred/pictures/core.pic

/usr/include/X11/bitmaps/core.pic

A picture file specification may have a slash in its name, so that, for example, the desktop would look for the **patterns/checked.pic** picture file as follows:

/user/fred/pictures/patterns/checked.pic

/usr/include/X11/bitmaps/patterns/checked.pic

Picture files are standard AIXwindows bitmap files, containing configuration items and picture data. These items can appear in any order, except that all configuration items must appear before all data items.

Configuration Items

Each configuration item must be on a separate line, in the following format:

#define *name value*

The fields must all appear on the same line, with the # sign in the first column.

The first part of the name field is equivalent to the characters of the picture file's base name that precede the first period. It can only contain letters, numbers, and underscores. For example, for the **pic.px** picture file, the first part of the name would be **pic.**

picture.**width**
picture.**height**
Must be included. Their values are numbers and indicate the width and height of the picture in pixels. If the picture is used for an icon, button, or cursor, this is the size of the object. If it is used as a background, the picture is tiled across the area; these items are still required to enable the desktop to interpret the data items.

*picture*_**x_hot**
*picture*_**y_hot**
Must either both be included or both be omitted. They are only used if the picture is the data portion of a cursor, and indicate the coordinates within the picture where the cursor is actually located. For example, if both values are 0, the actual point of the cursor would be the top left corner of the picture.

*picture*_**bg**
*picture*_**fg**
Must be the names of colors, surrounded by double quotes, giving the foreground and background colors of the picture. If the color string begins with a # character, the remainder of the color name encodes the actual color. The string gives the red, green, and blue components of the color, in that order, as one, two, three, or four hexadecimal digits each (Components written 5, 50, 500, and 5000 are all the same, and differ from 05, 050, and 0005.). For example:

black is `"#000000000000"` or `"#000"`
white is `"#ffffffffffff"`
red is `"#ffff00000000"`

If the color does not begin with a # character, the string's meaning depends on the AIXwindows server. The **/usr/lpp/X11/rgb/rgb.txt** file contains a listing of valid color names and the information AIXwindows uses in displaying them on the screen.

If these items are omitted, the desktop reads the default value for the foreground from the **Xdesktop.fileIcon.pixmap.foreground** resource and reads the default value for the background from the **Xdesktop.fileIcon.pixmap.background** resource.

Picture Data

Picture data shows the actual bitmap of the picture, in the format

static unsigned char *picture_bits* **[ ] = {** *data* **}**

The word **unsigned** can be omitted.

The *data* parameter represents the actual data, consisting of a sequence of two digit hexadecimal values, each prefixed with 0x and separated by commas.

There may be up to 20 such values per line, though it is usual to have 12 values.

The total number of values is equal to:

((*picture_width***+7)/8)****picture_height*

with

((*picture_width***+7)/8)**

values for each row of the picture. The division is rounded down to the next lower integer.

Each value represents eight consecutive pixels, except for the last value in the row, which may represent less.

The following example represents the picture file for a menu pointer:

```
#define menu_d_width     16
#define menu_d_height    16
#define menu_d_x_hot     14
#define menu_d_y_hot     5
#define menu_d_bg        "black"
#define menu_d_fg        "white"
static char menu_m_bits[] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0e, 0x00,
        0xfa, 0x1f, 0x02, 0x20, 0xc2, 0x1f, 0x02, 0x02,
        0xc2, 0x03, 0x02, 0x02, 0xfa, 0f01, 0xfe, 0x00
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
```

Defining Triggers for the AIXwindows Desktop

This chapter describes how to define the trigger maps, trigger IDs, and resources that the AIXwindows Desktop uses to interpret mouse actions.

To give the desktop the flexibility to work with any type of mouse with from one to five buttons, the rules describing mouse actions are usually expressed in terms of triggers rather than physical buttons.

Mapping strings within desktop rule files describe all triggers to be interpreted by the desktop. This string consists of a sequence of mappings, separated by semicolons. Spaces are ignored within the mapping string.

Each trigger mapping maps a trigger to a trigger-id string, in the following form:

TriggerSet = *TriggerID*

where:

TriggerSet	Indicates a list of steps separated by commas, and
TriggerID	Indicates the letter s or d followed by a number.

The desktop converts clicks on the mouse buttons first into triggers, and then into trigger ids. This mechanism is controlled by three items in the **.Xdefaults** mechanism:

- the mapping (a string)
- the maximum motion (a number of pixels)
- maximum up time (measured in milliseconds).

The conversion is accomplished in the following stages:

1. The motions and button presses are converted into triggers, controlled by the maximum motion and the maximum up time.
2. The triggers are converted to trigger IDs through the mapping string.

   Most users will not change the trigger to trigger ID mapping, since it has been optimized for the particular mouse supplied with the computer system. On PS/2 systems, the mapping has been optimized for a two–button mouse, with the following mapping:

Button	**Location**
1	The left mouse button
2	The right mouse button
3	Both mouse buttons

Trigger Sets

A *trigger set* is a set of steps made up of closely spaced button presses and releases. The trigger includes presses and releases of the indicated mouse buttons, beginning with the first button pressed when all buttons are raised, and including all presses and releases until the next point when all the buttons are raised.

A step is labelled by giving the numbers of all the mouse buttons that are depressed at any time during the step, no matter what order they are in or how long they are down.

For example, all three of the following examples would be labelled as 1 and 2, or 12:

- Press button 1, press button 2, release button 1, and release button 2.
- Press button 2, press button 1, release button 1, and release button 2.
- Press button 1, press button 2, release button 2, press button 2, release button 2, and release button 1.

If the mouse pointer moves by a distance greater than the amount indicated in the **xdesktop.triggers.maxMotion** resource, the step is considered a drag; otherwise, it is a click.

Clicks and drags are described by giving the numbers of the mouse buttons, as in `12`.

The steps in a trigger definition are separated by commas. For example, a double click on button 2 is described as 2,2.

If the last step in the sequence is a drag, the trigger is defined as a *dynamic trigger*, and the desktop signifies detection of the drag by changing the cursor to the drag cursor or to the multi–drag cursor. Other triggers are defined as *static* triggers.

A trigger ends when either no button is pressed for the maximum up time after a step, or at the end of a drag, whichever comes first.

The TriggerID String

The *TriggerID* string identifies the string by which the **trigger_action** clauses in the rule file refer to the trigger.

The letter **s** is used for static triggers, such as clicks or double clicks, where the mouse is not moved. The letter d is used for dynamic triggers or drags, such as where one icon is dropped onto another.

For example, the usual assignment for a three button mouse is for the three trigger–ids **s1, s2,** and **s3** to be double clicks on the left, center, and right mouse buttons, respectively. The usual assignment for a two button mouse is that the same trigger–ids refer to double clicks on the left button, the right button, and both mouse buttons, respectively.

A static trigger can also be used to control the selection of icons. This is done by using one of the following codes instead of a trigger id:

+s	If the current icon (the icon under the cursor) is not selected, select it.
–s	If the current icon is selected, deselect it.
!s	Deselect all selected icons; then select the current icon.
~s	If the current icon is selected, deselect it. Otherwise, select it.

For example, the following string indicates that a double click on button 1 selects the current icon in addition to any icons already selected. A double click on button 2 deselects the icon.

```
1=+s ; 2=–s
```

The desktop ignores trigger id strings containing more than five steps.

Macro definitions

Macro definitions allow one or more buttons to be abbreviated to a single letter. This allows mappings to be made more abstract, and easier to convert for a different number of buttons.

For example, suppose that you have designed a set of mappings for a three button mouse, and that you want to convert it to work on a two button mouse. One way might be to say that the center button is represented by using both left and right buttons together. By specifying all the mappings in terms of the letters L, C, and R, rather than the numbers 1, 2, and 3, the mappings are easier to change (especially since the right button changes from being number 3 to number 2.)

A macro definition consists of a set of button numbers, an equals sign, and a single letter. That letter can then be used in any future trigger description or macro definition.

For example, a trigger mapping with three static trigger IDs, three dynamic trigger IDs, and three selection control triggers, might be written as follows for a three button mouse:

```
1=L          ; 2=C         ; 3=R         ; \
L=!s         ; C=+s        ; R=–s        ; \
L,L=s1       ; C,C=s2      ; R,R=s3      ; \
L=d1         ; C=d2        ; R=d3
```

The backslashes indicate that the mapping is continued on the next line.

To convert to the two button mouse, change the first line as follows:

```
1=L              ; 12=C            ; 2=R             ; \
```

The mapping then becomes equivalent to the following:

```
1=ls             ; 12=+s           ; 2=-s            ; \
1,1=s1           ; 12,12=s2 ; 2,2=s3         ; \
l=d1             ; 12=d2           ; 2=d3
```

The AIXwindows desktop utilities can be used from shell prompts, within shell scripts, and within rule files to manipulate and communicate with the AIXwindows Desktop.

Desktop Utility Descriptions

The desktop includes the following AIX utilities which can be included in **actions** statements within rule files or within shell scripts:

mfyi	(for your information) Displays a specified message in a window. You can also specify the size and color of the window.
myni	(yes or no input) Displays a simple dialog box, and returns a code indicating which button the user pressed.
mgti	(get text input) Displays a prompt and reads in text.
tellxdt	Passes desktop command language commands to the desktop and requests information from the desktop about icons.

mfyi Utility

Purpose

Displays an information message in a window.

Syntax

mfyi [*–ToolkitOption...*] [*–Option...*] [*TextForDisplay*]

Description

The **mfyi** utility displays an icon, the information type, and left–justified text within a window. The desktop removes the window when the user clicks on the icon to acknowledge the information.

Flags

The *ToolkitOption* parameter values are the standard Xtoolkit command line arguments specified in the following list:

–bg *Color*	Specifies the border color of the window. The **/usr/lpp/X11/rgb/rgb.txt** file contains a listing of valid color names and the information AIXwindows uses in displaying them.
–fg *Color*	Specifies the color to use for displaying text.
–fn *Font*	Specifies the font to be used for displaying normal text. The **/usr/lpp/X11/defaults/fonts** file contains a list of valid font names and the information AIXwindows uses in displaying them.
–geometry *Geometry*	Specifies the preferred size and position of the information window
–display *Host:Display*	Specifies the X–server to contact.

–name *Name*	Specifies the application name to use when retrieving resources.
–xrm *ResourceString*	Specifies a resource string to be used. This is especially useful for setting resources that do not have separate command line options.

Additional options can be specified with the *Option* parameter.The valid options are:

–center (or –centre)	Indicates that the message window is centered on the screen. This option overrides the x and y coordinates given in a **–geometry** command line option. This option is not valid if the **–override** command line option has not been specified.
–help	Indicates that a brief summary of the allowed options is output to standard error.
–picture *Name*	Indicates the bit map file name for the icon image. Overrides the icon type specified by the **–type** argument.
–transfor *%W*	Causes the deskware utility windows to be transient.
–type *Text*	Specifies the type of information and the type of icon to be displayed. The *Text* parameter can have the following values: **error** **information** **message** **warning**

Return Codes

0	Ending through the acknowledge symbol
1	Ending without displaying information

Example

```
mfyi –picture warning.xbm –type warning Warning: Accounting stopped
```

In this example, **mfyi** opens an information window displaying the icon contained in the **warning.xbm** bit map file and the text: `"Warning: Accounting stopped"`.

myni Utility

Purpose

Displays a simple dialog box.

Syntax

myni [*–ToolkitOption...*] [*–Option...*] [*TextForDisplay*]

Description

The **myni** utility displays a question and two control buttons labeled **cancel** and **confirm**. The **myni** utility also returns a code indicated the button on which the user clicked.

Flags

The *ToolkitOption* parameter values commonly used are the standard Xtoolkit command line arguments specified in the following list:

–bg *Color*	Specifies the border color of the window. The **/usr/lpp/X11/rgb/rgb.txt** file contains a listing of valid color

	names and the information AIXwindows uses in displaying them.
–fg *Color*	Specifies the color to use for displaying text.
–fn *Font*	Specifies the font to be used for displaying normal text. The **/usr/lpp/X11/defaults/fonts** file contains a list of valid font names and the information AIXwindows uses in displaying them.
–geometry *Geometry*	Specifies the preferred size and position of the information window
–display *Host:Display*	Specifies the X–server to contact.
–name *Name*	Specifies the application name to use when retrieving resources.
–xrm *ResourceString*	Specifies a resource string to be used. This is especially useful for setting resources that do not have separate command line options.

Additional options can be specified with the *Option* parameter . The valid options are:

–cancel *FileName*	Indicates the bit map file to be used for the cancel button.
–center (or –centre)	Indicates that the message window is centered on the screen. This option overrides the x and y coordinates given in a **–geometry** command line option. This option is not valid if the **–override** command line option has not been specified.
–confirm *FileName*	Indicates the bit map file to be used for the confirm button.
–help	Indicates that a brief summary of the allowed options is output to standard error.
–no *Text*	If no bit map file has been assigned to the cancel control button, indicates the text label for the button.
–picture *Name*	Indicates the bit map file name for the icon image. Overrides the icon type specified by the **–type** argument.
–transfor *%W*	Causes the deskware utility windows to be transient.
–yes *Text*	If no bit map file has been assigned to the confirm control button, indicates the text label for the button.

Return Codes

0	Ending through the confirm control button
1	Ending due to internal error
2	Ending through the cancel control button

Example

The following example opens a dialog window with:

- The icon contained in the bit map file `question.xbm`
- The confirm button using the file indicated in the bit map file `tick.xbm`
- The cancel button using the file indicated in the bit map file `cross.xbm`
- The question: `Delete file xyz?`

```
myni -picture question.xbm -confirm tick.xbm -cancel cross.xbm
Delete file xyz?
```

mgti Utility

Purpose

Prompts the user for input and returns the text entered to standard output.

Syntax

mgti [*Option...*] [*DefaultString*]

Parameters

The *ToolkitOption* parameter values commonly used are the standard Xtoolkit command line arguments specified in the following list:

–bg *Color*	Specifies the border color of the window. The **/usr/lpp/X11/rgb/rgb.txt** file contains a listing of valid color names and the information AIXwindows uses in displaying them.
–fg *Color*	Specifies the color to use for displaying text.
–fn *Font*	Specifies the font to be used for displaying normal text. The **/usr/lpp/X11/defaults/fonts** file contains a list of valid font names and the information AIXwindows uses in displaying them.
–geometry *Geometry*	Specifies the preferred size and position of the information window
–display *Host:Display*	Specifies the X–server to contact.
–name *Name*	Specifies the application name to use when retrieving resources.
–xrm *ResourceString*	Specifies a resource string to be used. This is especially useful for setting resources that do not have separate command line options.

Additional options can be specified with the *Option* parameter . The valid options are:

–center (or –centre)	Indicates that the message window is centered on the screen. This option overrides the x and y coordinates given in a **–geometry** command line option. This option is not valid if the **–override** command line option has not been specified.
–help	Indicates that a brief summary of the allowed options is output to standard error.
–transfor *%W*	Causes the deskware utility windows to be transient.
–prompt *Prompt*	Prompts the user for information.

tellxdt Utility

Purpose

Passes desktop command language commands to the desktop and requests information from the desktop about icons.

Syntax

tellxdt [*Name*][*Query*][*Command*]

Flags

If no parameters are indicated, the **tellxdt** command displays a usage summary.

The following *Name* parameters determine the desktop to which commands or queries are sent. If none are indicated, the commands are sent to the desktop opened most recently.

–l	Lists the names of all available desktops.
–a	Sends the command to all running desktops.
–n *name*	Send the command to the named desktop.

The following *Query* parameters request information from the desktop about icons.

–t *FileName*	Returns the title of the icon for the named file.
–i *FileName*	Returns the name of the picture file for the file's icon image.
–c *FileName*	Returns the class code for the file.
–s *Number FileName*	Returns the static trigger action for the numbered trigger.
–S *Number FileName*	Returns the static trigger action for the numbered trigger, performing substitutions.
–d *Number FileName*	Returns the dynamic trigger action for the numbered trigger.
–D *Number FileName*	Returns the dynamic trigger action for the numbered trigger, performing substitutions.
–L	Lists all active triggers for the desktop.
–	Accepts commands from standard input until it receives an End of File character.

The *Command* parameter can be any command in the desktop command language, with the appropriate arguments.

The desktop command language allows certain actions to be carried out within the desktop. These actions are mainly concerned with icons, directory windows, and files.

Desktop command language statements can be included within **actions** clauses in rule files, or invoked from within a shell using the **tellxdt** command. For example, the following command:,

```
tellxdt get_out_icon desktop
```

places the clock's icon on the desktop.

Desktop command language statements have two advantages:

- They automatically update the desktop.
- They do not rely on the availability of particular AIX binary files.

Statements in the desktop command language consist of words separated by spaces. The end of a statement is determined by the mechanism that initially generates the statement; for example, within a rule file the end of a statement is indicated by a semicolon. A backslash causes the subsequent character to be part of the current word, even if it is a space character. For example, the following statement contains only two words:

```
This\ is\ a\ statement with\ only\ two\ words
```

All valid statements begin with either a descriptive statement name or a three letter equivalent, followed in some cases by a number of parameters. Some statements require an exact number of parameters, and the effect of having the wrong number is undefined. Other statements accept any number of parameters.

Chapter 2. AIXwindows Window Management

Changing the Appearance and Behavior of AIXwindows Overview

You can change the way that windows and menus appear and behave by editing information in the window management resource database. This database is built from the following sources, listed in order of precedence, low to high:

- the **/usr/lpp/X11/$LANG/app–defaults/Mwm** file (where **$LANG** represents the value of the user's **LANG** environment variable).
- **resource_MANAGER** root window property or **$HOME/.Xdefaults** file
- **XENVIRONMENT** environment variable or **$HOME/.Xdefaults–host** file
- **mwm** command line options.

Entries in the resource database can refer to other resource files for specific types of resources. These include files that contain bit maps, fonts, and window management specific resources such as menus and behavior specifications, and button and key bindings.

Information about window management is included in the standard **.Xdefaults** file format, with the **mwm** resource name and the **Mwm** class name. In the following discussion of resource specifications **Mwm** and **mwm** can be used interchangeably.

The AIXwindows window manager uses the following resource sets:

Component appearance resources:

These resources specify appearance resources of window management user interface components. They can be applied to the appearance of window management menus, feedback windows (for example, the window reconfiguration feedback window), client window frames, and icons.

Specific appearance and Behavior resources:

These resources specify window management appearance and behavior (for example, window management policies). They are not set separately for different window manager user interface components.

Client- Specific resources:

These window management resources can be set for a particular client window or class of client windows. They specify client–specific icon and window frame appearance and behavior.

Resource identifiers can be either a resource name (in which case the first letter is in lowercase) or a resource class (in which case the first letter of the name is in uppercase). If the value of a resource is a file name and if the file name is prefixed by "~/", then it is relative to the path contained in the **$HOME** environment variable (generally the user's home directory). This is the only environment variable that window management uses directly (the **$XENVIRONMENT** environment variable is used by the resource manager).

Understanding the AIXwindows Window Management Resource Description File

The AIXwindows window management resource description file contains descriptions of resources and functions that are used for window management. It contains descriptions of resources that cannot be easily encoded in the defaults files.

The **configFile** resource indicates the pathname of the user's AIXwindows window management resource description file. By default, if the file exists, the filename is **$HOME/.mwmrc**.

The following types of resources are described in the file:

Buttons	Window management functions can be bound (associated) with button events.
Keys	Window management functions can be bound (associated) with key press events.
Menus	Menu panes can be used for the window menu and other menus posted with key bindings and button bindings.

The AIXwindows window management resource description file is a standard text file. Items of information in the file are separated by white space (blanks, tabs, and new line characters). Blank lines are ignored.

All strings and characters are interpreted within the file, except for the following information:

- Strings surrounded by double quotes (").
- Single characters preceded by the backslash character (\).
- Lines in which the first character is an exclamation point (!), which are regarded as comments.
- Text following a pound sign (#), unless the pound sign is preceded by a backslash, which are regarded as comments.

Resource Description File Functions

Window management functions can be accessed with button and key bindings and with window management menus. Functions are indicated as part of the specifications for button and key binding sets and menu panes. The function specification has the following syntax:

Function = *FunctionName* [*FunctionArgument*...]...

The value of the *FunctionName* parameter is the name of a window management function and the *FunctionArgument* parameter is a list of quoted or unquoted items.

If a function is specified that is not one of the supported functions, the window management interprets it as the **f.nop** function and does nothing.

You can indicate which resource types can specify a function and the context in which the function can be used.. Function contexts include the following:

root	No client window or icon is selected as an object for the function.
window	A client window is selected as an object for the function. This includes the window's title bar and frame. Some functions are applied only when the

window is in its normalized state (such as the **f.maximize** function) or its maximized state (such as the **f.normalize** function).

icon An icon is selected as an object for the function.

If a function is specified in a type of resource that does not support that function or is called in a context that does not apply, the function is treated as **f.nop**. The following table indicates the resource types and function contexts in which **mwm** functions apply:

Function	Contexts	Resources
f.beep	root, icon, window	button, key ,menu
f.circle_down	root, icon, window	button, key, menu
f.circle_up	root, icon, window	button, key, menu
f.exec	root, icon, window	button, key, menu
f.focus_color	root, icon, window	button, key, menu
f.focus_key	root, icon, window	button, key, menu
f.kill	icon, window	button, key, menu
f.lower	root, icon, window	button, key, menu
f.maximize	icon, window (normal)	button, key, menu
f.menu	root, icon, window	button, key, menu
f.minimize	window	button, key, menu
f.move	icon, window	button, key, menu
f.next_cmap	root, icon, window	button, key, menu
f.next_key	root, icon, window	button, key, menu
f.nop	root, icon, window	button, key, menu
f.normalize	icon, window (maximized)	button, key, menu
f.pack_icons	root, icon, window	button, key, menu
f.pass_keys	root, icon, window	button, key, menu
f.post_wmenu	root, icon, window	button, key
f.prev_cmap	root, icon, window	button, key, menu
f.prev_key	root, icon, window	button, key, menu
f.quit_mwm	root	button, key, menu
f.raise	root, icon, window	button, key, menu
f.raise_lower	icon, window	button, key, menu
f.refresh	root, icon, window	button, key, menu
f.refresh_win	window	button, key, menu
f.resize	window	button, key, menu
f.restart	root	button, key, menu
f.send_msg	icon, window	button, key, menu
f.separator	root, icon, window	menu
f.set_behavior	root, icon, window	button, key, menu
f.title	root, icon, window	menu

Specifying Component Appearance Resources for AIXwindows Window Management

The AIXwindows component appearance resources specify the appearance of the components of the AIXwindows window management user interface. These resources can be applied to the appearance of window management menus, feedback windows, client window frames, and icons.

Resources with the **active** prefix refer only to frames and icons.

The syntax for specifying component appearance resources that apply to window management icons, menus, and client window frames is the following:

Mwm*ResourceID*: *Value*

For example, the **Mwm*foreground** resource is used to specify the foreground color for window management menus, icons, and client window frames.

The syntax for specifying component appearance resources that apply to a particular window management component is the following:

Mwm*Type***ResourceID*: *Value*

The *Type* parameter has a value of **menu**, **icon**, **client**, or **feedback**.

If **menu** is specified, the resource is applied only to window management menus; if **icon** is specified, the resource is applied to icons; and if **client** is specified, the resource is applied to client window frames. For example,:

- The **Mwm*icon*foreground** resource is used to specify the foreground color for window management icons.
- The **Mwm*menu*foreground** resource specifies the foreground color for window management menus.
- The **Mwm*client*foreground** resource is used to specify the foreground color for window management client window frames.

You can configure separately the appearance of the title area of a client window frame (including window management buttons). The syntax for configuring the title area of a client window frame is the following:

Mwm*client*title*ResourceID*: *Value*

For example, the **Mwm*client*title*foreground** resource specifies the foreground color for the title area. Defaults for title area resources are based on the values of the corresponding client window frame resources.

You can configure the appearance of menus based on the name of the menu. The syntax for specifying menu appearance by name is the following:

Mwm*menu*MenuName***ResourceID*: *Value*

For example, the **Mwm*menu*my_menu*foreground** resource specifies the foreground color for the menu named **my_menu.**

Specifying Specific Appearance And Behavior Resources for AIXwindows Window Management

The AIXwindows specific appearance and behavior resources control AIXwindows window management policies for all user interface components.

The syntax for specifying window policies is the following:

Mwm*ResourceID*: *Value*

For example, the **Mwm*keyboardFocusPolicy** resource specifies the window management policy for setting the keyboard focus to a particular client window.

Specifying Client Specific Resources for AIXwindows Window Management

The AIXwindows client specific resources specify the appearance and behavior for the icons and window frames of a particular client window or class of client windows.

The syntax for specifying client–specific resources is the following:

Mwm*C*lientName** *ResourceID*

or,

Mwm* *Class** *ResourceID*

For example, the **Mwm*aixterm*windowMenu** resource specifies the window menu to be used with aixterm clients.

The syntax for specifying client–specific resources for all classes of clients is

Mwm* *ResourceID*: *Value*

Individual client specifications take precedence over the general specifications for all clients. For example, the **Mwm*windowMenu** resource specifies the window menu to be used for all classes of clients that do not have a window menu specified.

The syntax for specifying resource values for windows that have an unknown name and class (that is, windows without an associated **WM_CLASS** property) is the following:

Mwm* *Defaults** *ResourceID*: *Value*

For example, the **Mwm*defaults*iconImage** resource specifies the file name of the icon image to be used for windows that have an unknown name and class.

Chapter 3. AIXwindows Toolkit

AIXwindows Toolkit Overview for Programming

The AIXwindows Toolkit is a collection of C language data structures and subroutines stored in an AIXwindows library named the **libXm.a** library. The tools in this library simplify the creation of interfaces for X11 client applications running in an X-Windows environment. These tools are based upon (and often call upon) the X-Windows Toolkit, a graphic toolkit that provides direct access to the lower levels of the AIX operating system. One obvious consequence of this interaction is that the naming conventions established for the AIXwindows Toolkit parallel those established for the X-Windows Toolkit. Together, these two toolkits provide a consistent and effective means of generating graphical objects and combining them to create AIXwindows interfaces.

Object Oriented Data Structures

AIXwindows data structures are object oriented. This means that they describe related classes of simple graphical objects. These object classes are called *widget classes* or *gadget classes* depending upon the interactive options they support. Each graphical object generated from these data structures contains *appearance* information, *behavioral* information (information concerning the general behavior of each widget or gadget), and *state* information (information that determines the initial state of each widget or gadget). The AIXwindows Toolkit helps you combine individual *instances* of various widget and gadget classes into *compound* interactive objects containing several widgets and gadgets. These objects are the foundation of each AIXwindows interface. They provide the sophisticated visual cues necessary for ongoing user interaction.

Widget classes and gadget classes are arranged in a formal hierarchy. Each class shares a common set of resources to which it adds resources inherited from the classes that precede it in the hierarchy. This organizational approach conserves data storage space and reduces related overhead while maintaining a standard look and feel for AIXwindows interfaces.

Object Oriented Subroutines

In addition to widget class data structures and gadget class data structures, the AIXwindows Toolkit also provides object oriented subroutines with which to create, alter, and manipulate AIXwindows objects. These subroutines are divided into the following categories:

- Creation Subroutines
- Convenience Creation Subroutines
- Support Subroutines

Understanding AIXwindows Toolkit Naming Conventions

The naming conventions associated with the AIXwindows system and the AIXwindows Toolkit directly parallel X-Windows naming conventions. The following conventions are used to name AIXwindows tools and related software:

Prefixes

The **Xm** prefix is the primary AIXwindows name-space identifier. It is used for the following names:

- AIXwindows subroutines
- AIXwindows widget-gadget class names and instance pointers (Class pointers use the **xm** prefix as a name-space identifier.)
- AIXwindows versions of AIXwindows string macros (an N is appended to all resource name prefixes; a C is appended to all resource class name prefixes; an R is appended to all resource type prefixes.)
- AIXwindows include directories (directories that contain include files and global definitions)
- AIXwindows libraries containing include directories
- AIXwindows defines and callback reasons.

The **XmN** prefix is reserved for resource names, classes, and types. The **Xt** prefix identifies X-Windows subroutines and data structures.

Suffixes

- The **.h** suffix identifies include files.
- The **P.h** suffix identifies private include files.

Other Naming Conventions

- Spaces are removed from the names of subroutines, object class names, class pointers, instance pointers, and structures, as well as resource names, classes, and types. The character following each removed space is capitalized.
- Spaces are replaced by underscores in defines names and callback reasons. All other letters (except the m in the prefix) are capitalized.

Examples of Naming Conventions

Specific examples of these AIXwindows naming conventions include:

Item	**Example of Name**
subroutine	**XmCreateArrowButton**
widget/gadget class name	**XmArrowButtonGadget**
widget/gadget class pointer	**xmArrowButtonGadgetClass**
string macro (resource name)	**XmNtopShadowPixmap**
string macro (resource class)	**XmCTopShadowPixmap**
string macro (resource type)	**XmRPixmap**
include filenames	**Xm.h**
include directory	**/usr/include/Xm**
library containing include directory	**/usr/lib/libXm.a**
define name	**XmVERTICAL**
callback reason	**XmCR_VALUE_CHANGED**
X-Windows subroutine	**XtGrabKeyboard**

See AIXwindows Widgets and Gadgets Overview for Programming for a detailed description of widgets and gadgets.

AIXwindows Environment/6000 Character Set Support Overview for Programming

AIXwindows Environment/6000 consists of Graphical User Interfaces (GUIs) based on OSF/Motif Release 1.0 running in an X11 environment under AIX for RISC System/6000.

This overview describes the two character sets added or extended to provide Double Byte Character Set (DBSC) support for the AIXwindows environment for the RISC System/6000.

This overview uses the following terminology to distinguish between the original AIXwindows Environment/6000 and the current AIXwindows Environment/6000:

- The original AIXwindows Environment/6000 is referred to as SBCS (Single-Byte Character Set) AIXwindows Environment/6000 because it supports the SBCS (the character set usually used for European languages such as English).
- The current AIXwindows Environment/6000 is referred to as DBCS/SBCS AIXwindows Environment/6000 because in addition to supporting the DBCS usually used for Asian languages such as Japanese, it also supports the SBCS.

This overview also distinguishes between the original version and current version of AIX for RISC System/6000 that supports, as SBCS languages, any European language of code page pc850 such as English and French, and the original version and current version of AIX for RISC System/6000 with Japanese Kanji support that supports English—ASCII characters up to hex value 7F (127)—as an SBCS language and Japanese shift JIS code of code page pc932 as a DBCS language.

Note: The AIX for RISC System/6000 and the AIX for RISC System/6000 with Japanese Kanji support are different in some important aspects, including the supported code pages previously described.

The SBCS AIXwindows System/6000 Environment

The SBCS AIXwindows System/6000 Environment runs under the original version of AIX for RISC System/6000 or AIX for RISC System/6000 with Japanese Kanji Support. It supports the SBCS national languages supported by the original AIX for RISC System/6000 (including all European languages of code page pc850 such as English or French) but does not support DBCS national languages.

Note: English (ASCII characters up to hex value 127) is supported as an SBCS language in the original version of AIX for RISC System/6000 with Japanese Kanji Support.

The DBCS/SBCS AIXwindows System/6000 Environment

The DBCS/SBCS AIXwindows System/6000 Environment runs under the current version of AIX for RISC System/6000 or AIX for RISC System/6000 with Japanese Kanji Support. It supports all DBCS/SBCS national languages supported by the current version of AIX for RISC S/6000 or the current version of AIX for RISC System/6000 with Japanese Kanji Support. For example, the current version of AIX for RISC S/6000 supports all European languages of code page pc850 such as English and French, but does not support any DBCS language such as Japanese. Conversely, AIX for RISC System/6000 with Japanese Kanji Support supports Japanese as its DBCS national language and English (ASCII characters up to hex value 127) as its SBCS national language. However, AIX for RISC System/6000 with Japanese Kanji Support does not support any other SBCS languages such as French or German.

For more information, see How to Support a National Language in a **libXm** Application on page 3–21.

Functions and Features of the DBCS/SBCS AIXwindows Environment/6000

The following functions and features were added to the SBCS AIXwindows System/6000 Environment to create the DBCS/SBCS AIXwindows System/6000 Environment:

DBCS/SBCS Character Input

The **XmInputMethod** widget has been added to the SBCS AIXwindows System/6000 Environment toolkit widget set. It provides an interface between the AIX Input Method and the DBCS/SBCS AIXwindows Environment. The **XmInputMethod** widget also provides a common input method architecture to interface with various AIX Input Methods.

Applications that use the **XmInputMethod** widget interface can process the input/output of any DBCS/SBCS language supported by the AIX Input Method. In applications without the **XmInputMethod** widget interface, a key event is processed as SBCS key in either the AIX for RISC System/6000 or the AIX for RISC System/6000 with Japanese Kanji Support, which provides the following language support:

- All European languages of code page pc850 (such as English and French) are supported as SBCS languages in the current version of AIXwindows for RISC System/6000
- The English language (ASCII characters up to hex value 127) is supported as SBCS language in AIX for RISC System/6000 with Japanese Kanji Support.

Note: To input DBCS characters, set the LC_CTYPE locale category appropriately in your application.

Applications with or without an **XmInputMethod** widget can accommodate SBCS-only input. However, applications that do not create an **XmInputMethod** widget cannot process DBCS input.

Regardless of whether it is providing SBCS input or DBCS input, the **XmInputMethod** widget can be used with the same application program interface (API). This feature helps programmers develop applications that can be ported across SBCS and DBCS environments.

DBCS/SBCS Character Output

Note: To input DBCS characters, set the LC_CTYPE locale category appropriately in your application.

The **XmInputMethod** widget that has been added to the original SBCS AIXwindows widget set provides an AIXwindows toolkit for application programming. This toolkit includes a hierarchy of related graphical objects called widgets and gadgets, and the subroutines that manipulate them. All of the objects can display DBCS/SBCS characters.

SBCS and DBCS can be handled with the same application program interface. This feature helps programmers develop applications that can be ported across SBCS and DBCS environments.

Locale-sensitivity

A locale-sensitive application is one that can easily adapt to local National Language Support (NLS) features (for example, "Japanese character set") at run time.

The C-language **setlocale** subroutine helps maintain local-sensitivity by setting locale categories such as LC_CTYPE in one of the following two ways:

- From your application
- From the **LANG** environment variable at run time.

An application that invokes the **setlocale** subroutine can, without recompilation, select one language corresponding to its run-time environment, provided that language is also supported by the current version of AIX for RISC System/6000 or AIX for RISC System/6000 with Japanese Kanji Support.

The **XmInputMethod** widget, like the widget/gadget set, can handle DBCS/SBCS with the same API. The **setlocale** subroutine can dynamically select the appropriate AIX Input Method from all the Input Methods supplied by AIX for RISC System/6000 or AIX for RISC System/6000 with Japanese Kanji Support. This dynamic selection capability allows input and output of that language without recompilation, which helps application programmers develop locale-sensitive applications.

Upward Compatibility with SBCS AIXwindows Environment/6000

Applications built with the SBCS AIXwindows Environment/6000 toolkit are upward compatible at the source code level with the following exception:

- Changes in the data structure for the **XmFontList** widget class required corresponding changes in the subroutines associated with this widget class. The following subroutines were updated:
  - **XmFontListAdd**
  - **XmFontListCreate**
  - **XmFontListFree.**
- In addition, the following subroutine was added to the original set of SBCS AIXwindows System/6000 Environment subroutines to complete the DBCS/SBCS AIXwindows Environment/6000 subroutine set:
  - **XmStringLoadQueryFont**

An application that uses these subroutines to create or alter an **XmFontList** widget will maintain upward compatibility at the source code level. Applications that do not use these four **XmFontList** subroutines (or that extract members from **XmFontList**) might not be upward compatible.

Applications built with the SBCS AIXwindows Environment/6000 toolkit are also object compatible with the exception of applications that use widget private header files (such as **CommandP.h**). Applications that use widget private header files must be recompiled with the DBCS/SBCS AIXwindows Environment library.

Given the same locale, compatible applications can use the same resource files with DBCS/SBCS AIXwindows Environment/6000 toolkit that they use with SBCS AIXwindows Environment/6000 toolkit.

Conformance to IBM SAA and CUA

The DBCS/SBCS AIXwindows Environment/6000 maintains the same look and feel as the OS/2 Presentation Manager (PM) which conforms to the IBM Systems Application Architecture (SAA) and Common User Access (CUA). The PM currently supports the Japanese input environment.

Note: For Japanese support, refer to manual N:SN18–0101 from the base manual N:SC18–0812 which is titled "IBM AIX RISC System/6000 AIX Japanese Processing Functions" (This documentation is available to Japanese customers only.)

Widget Classes and Subroutines Added

The following widget classes are not supported by OSF/Motif Release1.0 but are supported as IBM-unique extensions.

- **XmInputMethod Widget Class** – This widget class provides an interface between an AIX Input Method and the DBCS/SBCS AIXwindows Environment/6000.
- **XmCreateInputMethod Subroutine** – This creation subroutine is added to the **XmInputMethod** widget class subroutine set.
- **XmStringLoadQueryFont Subroutine** – This subroutine parses a **fontList** string of fonts into a **FontList** structure.

XmInputMethod Widget Class Resources Added

The following resources have been added to the **XmInputMethod** widget class resource set:

- **XmNfocusWindow**
- **XmNfontList**
- **XmNinputMethod**
- **XmNpreEditFontList**
- **XmNpreEditType**
- **XmNpreEditWindow**
- **XmNsuppressInputMethod**
- **XmNworkWindow.**

Resource Added to Six Convenience Creation Subroutines

Note: Each creation subroutine has been changed slightly to support the **XmInputMethod** widget interface.

DBCS/SBCS AIXwindows Environment/6000 introduces the following resource to the resource sets for the **XmCreateBulletinBoardDialog**, **XmCreateFileSelectionDialog**, **XmCreateFormDialog**, **XmCreateMainwindow**, **XmCreateSelectionDialog**, and **XmCreatePromptDialog** Convenience Creation Subroutines:

- **XmNinputMethodWidget.**

Changes in Compound Strings Subroutines that Support DBCS/SBCS

The following subroutines that handle Compound Strings have been changed slightly to support DBCS. They now support both DBCS and SBCS:

Note: The only character set parameter currently available for DBCS is the **XmSTRING_DEFAULT_CHARSET** character set parameter.

- **XmFontListAdd**
- **XmFontListCreate**
- **XmFontListFree**
- **XmStringBaseline**
- **XmStringByteCompare**
- **XmStringCompare**
- **XmStringConcat**
- **XmStringCopy**
- **XmStringCreate**

- **XmStringDirectionCreate**
- **XmStringDraw**
- **XmStringDrawImage**
- **XmStringDrawUnderline**
- **XmStringEmpty**
- **XmStringExtent**
- **XmStringFree**
- **XmStringFreeContext**
- **XmStringGetLtoR**
- **XmStringGetNextComponent**
- **XmStringGetNextSegment**
- **XmStringInitContext**
- **XmStringLength**
- **XmStringLineCount**
- **XmStringLtoRCreate**
- **XmStringNConcat**
- **XmStringNCopy**
- **XmStringPeekNextComponent**
- **XmStringSegmentCreate**
- **XmStringSeparatorCreate**
- **XmStringWidth.**

Changes to the XmNmnemonic Resources of XmLabel and XmLabelGadget

A mnemonic is valid in the following object classes even if it is not matched with any character in the label string:

- **XmLabel** Widget Class
- **XmLabel** Gadget Class
- **XmRowColumn** Widget Class.

In the DBCS/SBCS AIXwindows Environment/6000, SBCS non-alphanumeric mnemonics can be displayed correctly but not activated. No warning is given about inoperable mnemonics.This is consistent with OSF/Motif Release 1.0 which allows display of non-alphanumeric mnemonics that cannot be activated.

Changes to Resources of type XmString

The ability to handle DBCS has been added to the following resources so that they support both DBCS and SBCS:

- **XmNacceleratorText**
- **XmNapplyLabelString**
- **XmNcancelLabelString**

- **XmNcommand**
- **XmNdialogTitle**
- **XmNdirMask**
- **XmNdirSpec**
- **XmNfilterLabelString**
- **XmNhelpLabelString**
- **XmNlablString**
- **XmNlistLabelString**
- **XmNmessageString**
- **XmNokLabelString**
- **XmNpromptString**
- **XmNselectionLabelString**
- **XmNtextString**
- **XmNtitleString**.

Note: To display DBCS characters, the LC_CTYPE locale category must be set appropriately in your application.

Changes to Resources of the type XmStringTable

The following resources of the type **XmStringTable** now support DBCS as well as SBCS:

- **XmNhistoryItems**
- **XmNitems**
- **XmNlistItems**
- **XmNselectedItems**.

Note: To display DBCS characters, the LC_CTYPE locale category must be set appropriately in your application.

Changes to Other Resources that Now Support DBCS

The following AIXwindows resources now support DBCS as well as SBCS:

- **XmNvalue**
- **XmNfontList**.

Note: To display DBCS characters, the LC_CTYPE locale category must be set appropriately in your application.

DBCS Strings

The capability to display a DBCS string as a title or menu is added to the AIXwindows window manager. The AIXwindows window manager now supports both DBCS and SBCS.

Localization of .mwmrc and Mwm

The window manager of AIXwindows uses **.mwmrc** as a configuration file, and **Mwm** as an application defaults resource specification file. Users can customize their own AIXwindows environment by editing these two files.

A new search path has been added for **.mwmrc** and **Mwm.** This new search path depends on the language environment as specified by the environment variable **LANG**. This path is an additional path. None of the original search paths associated with the original version of AIXwindows Environment/6000 (SBCS AIXwindows Environment/6000) has been modified.

For more information, see How to Set a Running Environment for Specific Languages on page 3–22.

Mnemonic Characters Not Included in the Label

The definition of mnemonics within **MenuPane** widgets is enhanced in DBCS/SBCS AIXwindows. Any SBCS character can be set as a valid mnemonic, even it is not matched as a character in the string of the label.

In the DBCS/SBCS AIXwindows environment, SBCS non-alphanumeric mnemonics can be displayed correctly but not activated. No warning is given about inoperable mnemonics.This is consistent with OSF/Motif Release 1.0 which allows display of non-alphanumeric mnemonics that could not be activated.

Suggested Reading

Related Information

Understanding the XmInputMethod Widget Class contains additional information about the **XmInputMethod** Widget Class.

Changing the Appearance and Behavior of AIXwindows for a Specific National Language contains detailed information about the AIXwindows Window Manager (**mwm**) and its associated resource description files.

Understanding the XmInputMethod Widget Class

The following topics are associated with the **XmInputMethod** Widget Class:

- Components of the **XmInputMethod** Widget
  - **workWindow** area

    A **workWindow** area usually contains at least one **XmText** widget that requires DBCS/SBCS input from the user.
  - **focusWindow** area

    A **focusWindow** area is the window under the **XmInputMethod** widget subtree that has the focus.
- An **XmInputMethod** widget is a parent widget that provides geometry and focus management for its children. The AIX Input Method creates the following three children of the **XmInputMethod** widget:

Note: Each child widget is resized whenever the application window is resized. If the application program contains a geometry resize policy, all the text strings in the children are rearranged as well.

  - **preEditWindow** area
  - **statusWindow** area
  - **auxiliaryWindow** area
- The **XmInputMethod** Widget Class Resource Set
  - preEditWindow

- focusWindow (the window that has the focus)
- workWindow

Note: You can access these resources using the Enhanced X-Windows **XtGetValues** subroutine.

The workWindow Area

A **workWindow** area provides a working area for an **XmInputMethod** widget. It can be an **XmRowColumn** widget or any other container widget.

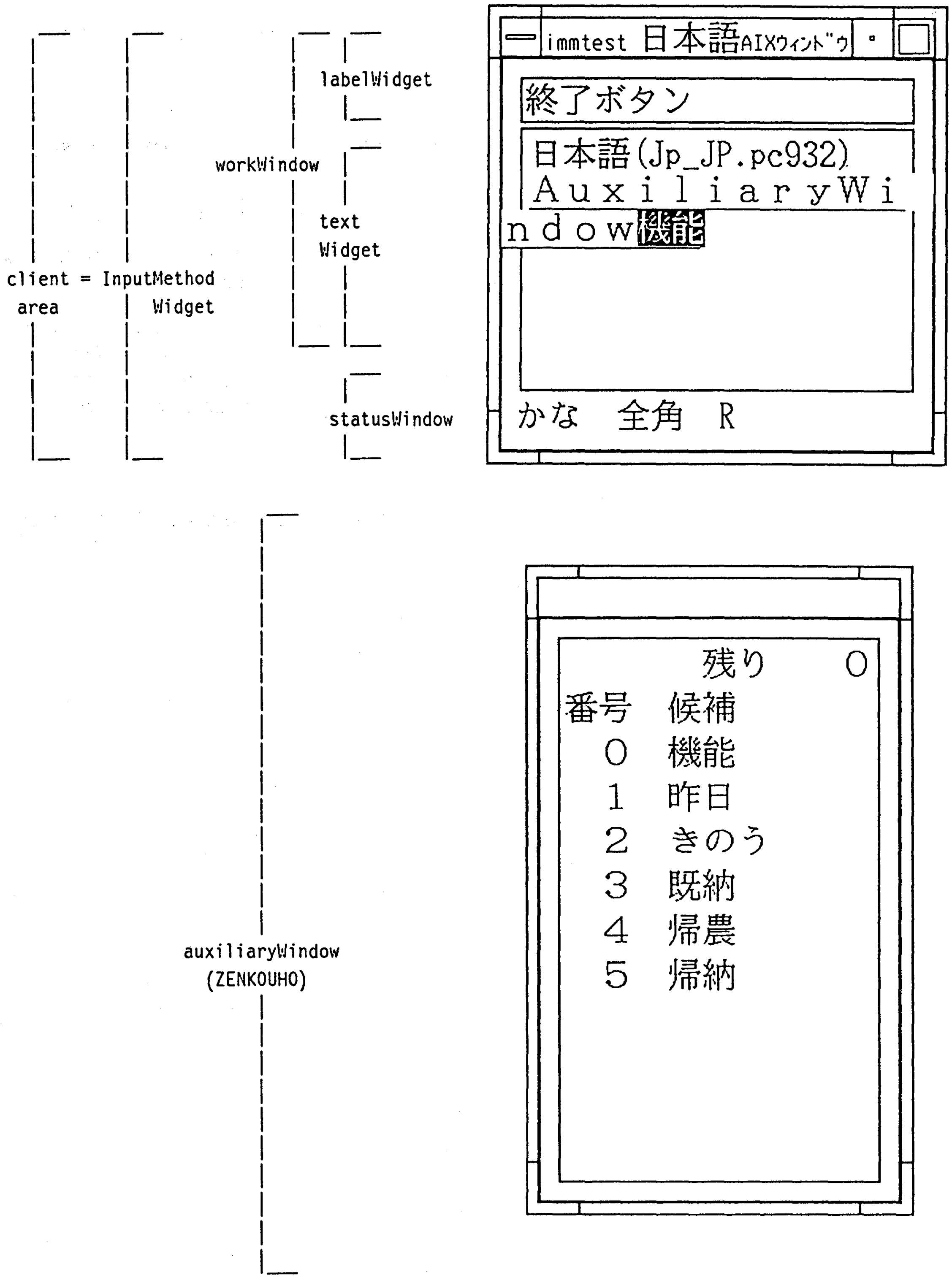
It is created and associated with an **XmInputMethod** widget in one of the following ways:

- Directly (You create the container widget that serves as the **workWindow** area and set its widget ID to the **XmNworkWindow** resource of **XmInputMethod**).
- Indirectly (The AIXwindows convenience creation dialog subroutines automatically create a widget that can serve as a **workWindow** area. For example, when you call the **XmCreateFileSelectionDialog** subroutine, it implicitly creates an **XmInputMethod** widget and an **XmFileSelectionBox** widget that can serve as the **workWindow** area of the **XmInputMethod** widget.)

The focusWindow Area

A **focusWindow** area is any window under the **XmInputMethod** widget subtree that has focus.

The XmInputMethod Widget Class Resource Set



XmInputMethod Widget with AuxiliaryWindow Structure

Notes:

1. In DBCS/SBCS AIXwindows Environment/6000, pre-editing of Japanese language input is undertaken in the following manner:
   - The string being pre-edited is underlined in the display. Underlined strings are divided into the following categories:
     - Current conversion targets – These strings can be re-converted. They are underlined and displayed in reverse video mode.
     - Not current conversion targets – These strings are underlined but not displayed in reverse video mode.
   - The user can change any string that is not a current conversion target into a current conversion target by moving the cursor so that the underlined string can be re-converted. A string that the user has pre-edited and confirmed cannot be reconverted. The confirmed string is moved from the **preEditWindow** area to the location where the user is entering characters in the **XmText** widget. The string is then displayed in normal mode without underlining.
2. The **auxiliaryWindow** area is a POPUP dialog to help Input Method. The figure titled **XmInputMethod** Widget with AuxiliaryWindow Structure shows an example that places an "all of the candidates" (ZENKOUHO) menu up for pre-editing of the Japanese Input Method `Jp_JP.pc932`. ZENKOUHO is helpful in pre-editing. It asks the user to select one word from all of the candidates provided.

The preEditWindow area

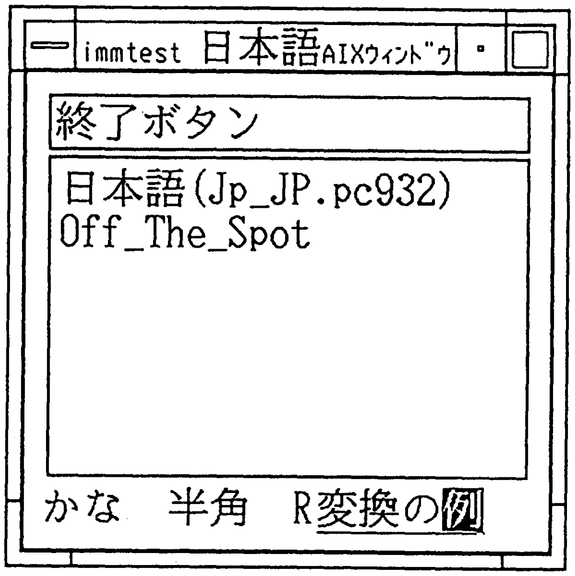
A **preEditWindow** area displays the string being pre-edited. The **XmInputMethod** widget class supports the following pre-editing modes:

- **OFF-THE-SPOT** pre-edit mode
- **POPUP** pre-edit mode.

The OFF-THE-SPOT Pre-edit Mode

To use the **OFF-THE-SPOT** mode, set the **XmNpreEditType** resource to an **XmInputMethod** resource using the Enhanced X-Windows **XtSetValues** subroutine or an appropriate entry passed from a resource file.

The **preEditWindow** area is always one line whose location is fixed at the bottom of the application dialog window and on the right side of a **statusWindow** area. Its visible line length is decided by the width of the dialog window. However, if a user exceeds the visible line length, input characters become invisible, but are still passed to the AIX Input Method and pre-edited. Consequently, the maximum number of characters is not limited by the visible line length of a **preEditWindow** area but cannot exceed 256 characters.



```
日本語(Jp_JP.pc932)
Off_The_Spot
```

OFF-THE-SPOT Pre-editing with the **XmInputMethod** Widget

Notes:

1. **OFF-THE-SPOT** pre-editing is one of the Japanese Input Method pre-editing styles. The location of pre-editing is fixed just below the **workWindow** area and on the right side of a **statusWindow** area.
2. In DBCS/SBCS AIXwindows, pre-editing of Japanese language input is undertaken in the following manner:
   - The string being pre-edited is underlined in the display. Underlined strings are divided into the following categories:
     - Current conversion targets – These strings can be re-converted. They are underlined and displayed in reverse video mode.
     - Not current conversion targets – These strings are underlined but not displayed in reverse video mode.
   - The user can change any string that is not a current conversion target into a current conversion target by moving the cursor so that the underlined string can be re-converted. A string that the user has pre-edited and confirmed cannot be reconverted. The confirmed string is moved from the **preEditWindow** area to the location where the user is entering characters in the **XmText** widget. The string is then displayed in normal mode without underlining.

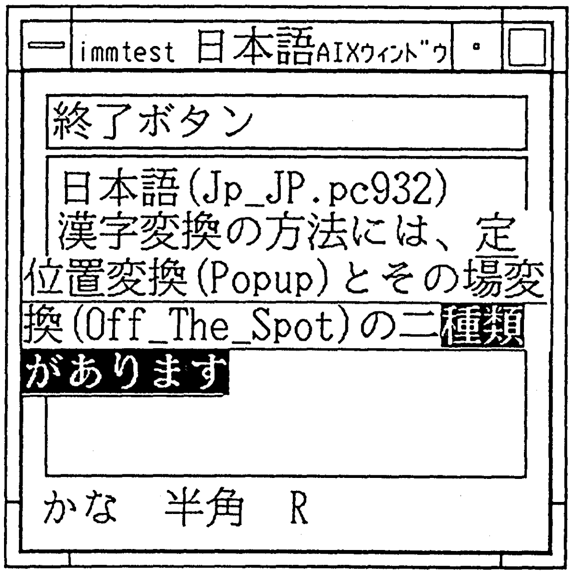
The POPUP Pre-edit Mode

To use the **POPUP** mode explicitly, set the **XmNpreEditType** resource to an **XmInputMethod** resource using the Enhanced X-Windows **XtSetValues** subroutine or an appropriate entry passed from a resource file.

In the POPUP pre-edit mode a **preEditWindow** area can have multiple lines whose length is decided by the width of the associated **workWindow** area. The maximum number of characters is limited to 256. The **preEditWindow** area is always located within the associated **workWindow** area and cannot cross its edges in any direction. If a pre-edited string reaches the bottom right corner of the **workWindow** area, the next character is displayed in the top left corner.

The current cursor position determines the location of the associated **preEditWindow area.** This approach ensures that the **preEditWindow** area is always located exactly where the user is entering characters.

Note: These characters always appear to the user to be part of the real data within an **XmText** widget.



日本語(Jp_JP.pc932)
漢字変換の方法には、定位置変換(Popup)とその場変換(Off_The_Spot)の二種類があります

POPUP Pre-editing with the **XmInputMethod** Widget

Notes:

1. POPUP pre-editing is one of the Japanese Input Method pre-editing styles.
2. In DBCS/SBCS AIXwindows, pre-editing of Japanese language input is undertaken in the following manner:
   - The string being pre-edited is underlined in the display. Underlined strings are divided into the following categories:
     - Current conversion targets – These strings can be re-converted. They are underlined and displayed in reverse video mode.
     - Not current conversion targets – These strings are underlined but not displayed in reverse video mode.
   - The user can change any string that is not a current conversion target into a current conversion target by moving the cursor so that the underlined string can be re-converted. A string that the user has pre-edited and confirmed cannot be reconverted. The confirmed string is moved from the **preEditWindow** area to the location where the user is entering characters in the **XmText** widget. The string is then displayed in normal mode without underlining.

The statusWindow Area

A **statusWindow** area reports the input/keyboard status of the AIX Input Method to the user. It is located in the lower left corner of the client area. It appears as part of the frame below the horizontal scrollbar.

If the pre-edit style is OFF-THE-SPOT, the **preEditWindow** area appears on the right side of the statusWindow structure. The **XmInputMethod** widget provides geometry management so that a **statusWindow** area is rearranged at the bottom corner of the client area when the client area is resized.

See the section titled XmInputMethod Widget Behavior Overview for Programming, page 19 for the coding steps required to ensure appropriate rearrangement of a **statusWindow** area.

The auxiliaryWindow Area

An auxiliaryWindow area helps the user with pre-editing. Based on the requirements of each national language, the AIX Input Method creates an **auxiliaryWindow** area, associates it with the **XmInputMethod** widget, and performs other necessary operations.

For example, the Japanese Input Method (JIM), "Jp_JP.pc932," creates the following types of auxiliaryWindow areas:

- ZENKOUHO
- JIS NUMBER
- Switching conversion method
  - SAKIYOMI-REN-BUNSETSU
  - IKKATSU-REN-BUNSETSU
  - TAN-BUNSETSU
  - FUKUGOU-GO.

Suggested Reading

Related Information

XmInputMethod Widget Behavior Overview for Programming contains additional information about the behavior and use of the AIXwindows **XmInputMethod** widget.

XmInputMethod Widget Behavior Overview for Programming

The **XmInputMethod** widget provides geometry and focus management for the AIX Input Method (IM) being used. The following topics are associated with **XmInputMethod** Widget Behavior:

- DBCS Languages Require Input Method and **XmInputMethod** Widgets
- **XmInputMethod** Widget Features
- Default Font Name for **XmInputMethod** Widgets

DBCS Languages Require Input Method and XmInputMethod Widgets

Asian languages, such as Japanese and Chinese, usually require DBCS encodings because their character sets have more than 256 characters (the maximum number of characters that can be expressed in SBCS). Languages such as these expressed in DBCS must be entered with pre-editing because their character sets have more characters than can reasonably be mapped to a terminal keyboard. For this reason, these DBCS languages require Input Methods (IM) that permits users to pre-edit (interactively compose each character in each language) all terminal keyboard input.

For example, AIX provides the following Japanese Input Method (JIM)

- `Jp_JP.pc932`

The interactive process of composing characters in such languages is called pre-editing.The DBCS/SBCS AIXwindows Environment supports the following types of pre-editing:

- **POPUP**
- **OFF-THE-SPOT**.

For this Input Method Interface, an **XmInputMethod** Widget is introduced in DBCS/SBCS AIXwindows. It provides an interface between an AIX Input Method and the DBCS/SBCS AIXwindows Environment, and supports geometry management and focus management.

XmInputMethod Widget Features

An XmInputMethod Widget provides the following unique features:

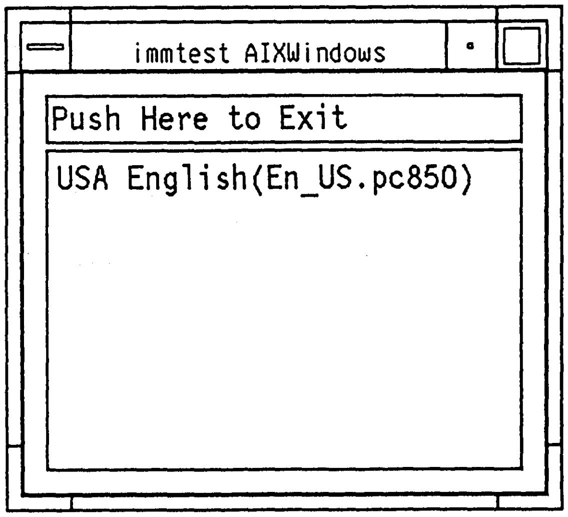
- Enables applications with an **XmInputMethod** interface to process DBCS/SBCS input/output supported by the AIX Input Method.
- Initializes an Input Method instance.

Note: If creation of the AIX Input Method fails, an **XmInputMethod** Widget cannot use AIX Input Method services. In this case, a key event is processed as an SBCS key in either the DBCS or SBCS environment.

- An AIX Input Method provide DBCS/SBCS input services according to its language-specific requirement. For example, the Japanese Input Method (JIM) "Jp_JP.pc932" provides the following pre-editing services using **X**, **Xt**, and **Xm** library subroutines:
  - Create a **preEditWindow** area
  - Associate it with an an **XmInputMethod** widget
  - Create a **statusWindow** area
  - Associate it with an an **XmInputMethod** widget
  - Create an **auxiliaryWindow** area
  - Associate it with an **XmInputMethod** widget
  - Draw the text being pre-edited
  - Change the **statusWindow** area.

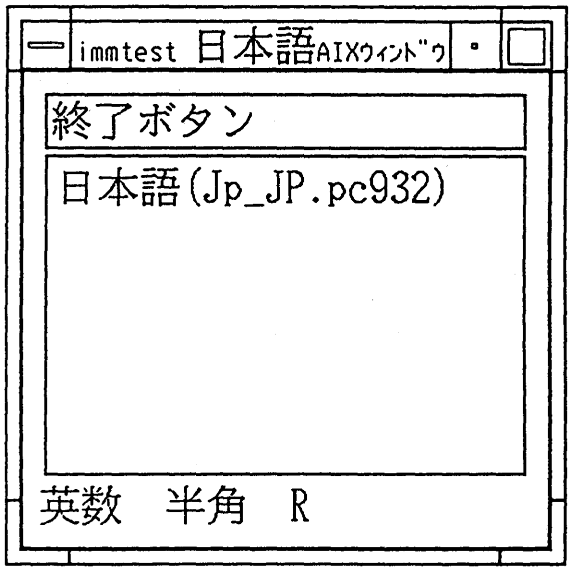
For example, the SBCS Input Method `En_US.pc850` will change keymaps.

- Enable the application to select an appropriate AIX Input Method dynamically at run time, provided the application is locale-sensitive.


immtest AIXWindows

Push Here to Exit

USA English(En_US.pc850)


immtest 日本語AIXｳｨﾝﾄﾞｳ

終了ボタン

日本語(Jp_JP.pc932)

英数　半角　R

Notes:

1. This locale-sensitive application can input/output the following languages without re-compilation:
   - English
   - Japanese.
2. This locale-sensitive application can also dynamically select an AIX Input Method and a resource file corresponding to its run-time environment as specified by the environment variables **LANG** and **XENVIRONMENT**.
   - English

   LANG is set to `En_US.pc850`, indicating one of the AIX Input Methods.

XENVIRONMENT is set to resE, which is the user-defined file name. No **statusWindow** area or **preEditWindow** area is created by the English Input method `En_US.pc850`.

- Japanese

LANG is set to `Jp_JP.pc932`, indicating one of the AIX Input Methods.

XENVIRONMENT is set to resJ, which is the user-defined file name. A **statusWindow** area and a **preEditWindow** area are created by the Japanese Input Method `Jp_JP.pc932`.

- Provide geometry management of the **statusWindow** area and a **preEditWindow** area when they are created by AIX Input Method.
  - For example,when the **applicationWindow** area is resized, the **statusWindow** area and **preEditWindow** area are resized and rearranged in the application window if the application program is made with a geometry resize policy. If the application program is not made with a resize policy, the **XmInputMethod** widget does not manage the **statusWindow** area at the lower left corner.

If the **statusWindow** area and **preEditWindow** area must be rearranged, the following coding is required:

When you create an **XmInputMethod** widget *directly* with either the AIXwindows **XmCreateInputMethod** subroutine, the Enhanced X-Windows **XtCreateManagedWidget** subroutine, or any equivalent subroutine, create an application shell as a parent of an **XmInputMethod** widget.

When you create an **XmInputMethod** widget *indirectly* by creating a main window with the AIXwindows **XmCreateMainWindowDialog** subroutine, create a widget that is a subclass of a **Shell** widget as a parent of a main window.

When you create an **XmInputMethod** widget *indirectly* by using one of the following AIXwindows convenience dialog creation subroutines, the dialog makes an **XmDialogShell** widget as a parent of an **XmInputMethod** widget:

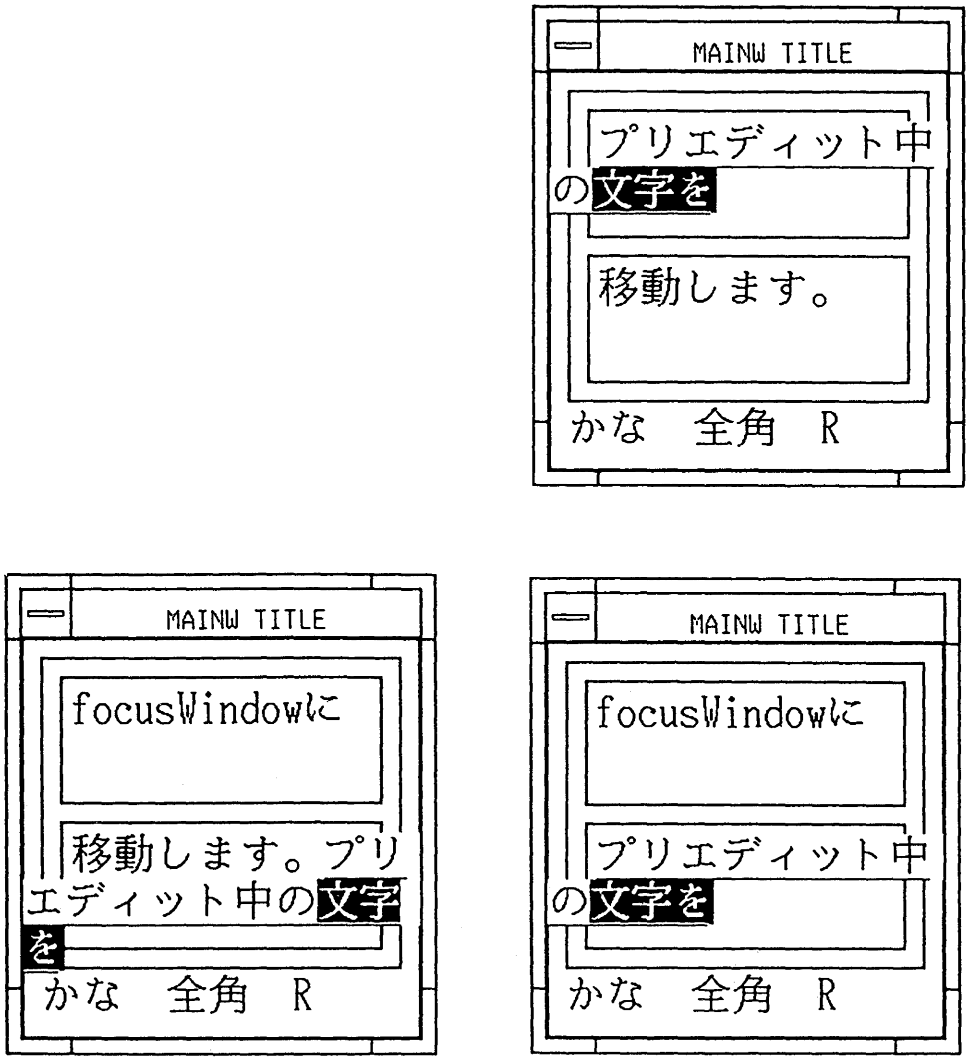
- **XmCreateBulletinBoardDialog**
- **XmCreateFileSelectionDialog**
- **XmCreateFormDialog**
- **XmCreateSelectionDialog**
- **XmCreatePromptDialog**.

Rearrangement of the children widgets is guaranteed by the **XmDialogShell widget**; there is no need to address explicitly the issue of geometry management.

- Provide an interface to the AIX Input Method.

A widget child that has the focus and requires keyboard mapping uses the **XmInputMethod** widget subroutines to handle keyboard input. Any "pre-editing" displays are performed by the **XmInputMethod** widget without **focusWindow** area involvement. Widgets treat the keyboard input as if a simple mapping is taking place even though several complex dialogs are actually occurring.

- Support the entry (by changing focus) of a string being pre-edited into any **XmText** widget within a widget tree.



プリエディット中の文字を

移動します。

focusWindowに

移動します。プリエディット中の文字を

focusWindowに

プリエディット中の文字を

Focus Change Management

Notes:

1. The string being pre-edited can be moved to the location of the focus by mouse clicking or any other appropriate means.

   - The string being pre-edited is underlined in the display. Underlined strings are divided into the following categories:

     - Current conversion targets – These strings can be re-converted. They are underlined and displayed in reverse video mode.

     - Not current conversion targets – These strings are underlined but not displayed in reverse video mode.

   - The user can change any string that is not a current conversion target into a current conversion target by moving the cursor so that the underlined string can be re-converted. A string that the user has pre-edited and confirmed cannot be reconverted. The confirmed string is moved from the **preEditWindow** area to the location where the user is entering characters in the **XmText** widget. The string is then displayed in normal mode without underlining.

2. A string that has already been pre-edited and confirmed cannot be reconverted.

For more information, see How to Create an **XmInputMethod** Widget on page 3–27, How to Create an **XmInputMethod** Widget Directly on page 3–28, and How to Create an **XmInputMethod** Widget Indirectly on page 3–29 .

Default Font Name for XmInputMethod Widget

If an application uses an SBCS AIX Input Method that does not require pre-editing, the **XmInputMethod** widget in the application does not require any fonts for the **statusWindow** area or the **preEditWindow** area.

If you want an application to use a DBCS AIX Input Method that requires pre-editing, the **XmInputMethod** widget in the application requires some fonts for the **statusWindow** area and the **preEditWindow** area.

To allow the **XmInputMethod** widget to use a specific font for pre-editing, specific font names should be provided to applications from the following resources in an appropriate resource file:

- fontList
- preEditFontList.

The **XmInputMethod** widget fontList resource is used for a **statusWindow** area. The **XmInputMethod** widget PreEditFontList resource is used for an OFF-THE-SPOT **preEditWindow** area. If nothing is set to the PreEditFontList resource (either by the Enhanced X-Windows **XtSetValues** subroutine or by the resource file), it is set to the same fonts as the **XmInputMethod** widget fontList resource at the time the **XmInputMethod** widget was created. The POPUP **preEditWindow** area and the **focusWindow** area use the same fontList resource. This resource requires keyboard input from the user in the form of an **XmText** widget.

For example, if the user wants to use Japanese fonts for pre-editing, fonts such as the following should be passed to the application:

```
*immwidget*fontList:        RomanKn12,Kanji12,Kana12,IBM_JPN12
*immwidget*preEditFontList: RomanKn12,Kanji12,Kana12,IBM_JPN12
```

Note: A default font, "fixed," is provided by the widget/gadget so that providing fontList resource is not necessarily required for an application using an SBCS language only. However, providing fontList explicitly by means of either a resource file or the **XtSetValues** subroutine is required for an application using a DBCS language because DBCS input/output requires DBCS fonts in a fontList resource and no widget/gadget provides DBCS fonts in its default fontList. The **XmInputMethod** widget always displays DBCS characters in its **statusWindow** area even if no widget/gadget in the widget tree has actual DBCS character data.

Suggested Reading

Related Information

XmInputMethod Widget Behavior Overview for Programming contains additional information about the behavior and use of the AIXwindows **XmInputMethod** widget.

The reference material for the following AIXwindows dialog creation subroutines contains additional information about using these subroutines to create **XmInputMethod** widgets indirectly: **XmCreateBulletinBoardDialog**, **XmCreateFileSelectionDialog**, **XmCreateFormDialog**, **XmCreateSelectionDialog**, **XmCreatePromptDialog**.

The **XmInputMethod** Widget Class reference material contains additional information about the **XmInputMethod** widget and its Resource Set.

How to Support a National Language in a libXm Application

Prerequisite Tasks or Conditions

1. Ensure that the include files and other header files listed in your source code are available and accessible.
2. Determine if your application requires Single-Byte Character Support (SBCS) or Double-Byte Character Support (DBCS).

Procedure

To ensure that your application is locale-sensitive (that is, it can easily adapt to local national language features at run time), follow the indicated steps in the order shown:

1. Add the C-language **setlocale** subroutine to your source code. This subroutine arranges for the **XmNinputMethodWidget** resource to be set to the current active locale.

   Syntax for the **setlocale** subroutine:

   char *setlocale(*locale_category*, *locale_name***);**
   int *locale_category*;
   char* *locale_name*;

2. Depending on the National Language Support (NLS) requirements of your application, you can call **setlocale** in either of the following ways:

   setlocale(LC_CTYPE, *locale***)**

   setlocale(LC_ALL, *locale***)**

   To determine the most appropriate form of this subroutine to use, keep in mind that the AIXwindows **libXm** library provides three features to help maintain locale-sensitivity:

   - Applications that invoke the **setlocale** subroutine can output one appropriate national language character set based on the locale.
   - Applications that invoke the **setlocale** subroutine can select the AIX Input Method of one appropriate language and input the language based on the locale.
   - To input/output a national language in applications with the AIXwindows **libXm** library, you must set LC_CTYPE. However, since LC_ALL includes LC_CTYPE, setting LC_ALL also allows applications to use a national language.

   The C library provides the following constant, which stands for Locale Character TYPE:

   #define LC_CTYPE 2

   The locale is established in the following manner:

   The *locale_name* is called by the **setlocale** subroutine. If the variable *locale_name* is set to any AIX Input Method explicitly, the locale is set to the current value of that variable. For example:

```
setlocale(LC_CTYPE, "En_US.pc850")   /*sets locale to US.pc850*/
```

   and

```
setlocale(LC_CTYPE, "Jp_JP.pc932")
    /*sets locale to Jp_JP.pc932*/
```

If the *locale_name* variable is set to "" in the following manner:

```
setlocale(LC_CTYPE,"");
```

the locale is set to the value of the environment variable **LANG**. Applications that invoke this setting can select one appropriate language corresponding to its run-time environment variable, **LANG**, without recompilation.

If the *locale_name* variable is set to **NULL** in the following manner:

```
setlocale(LC_ALL, NULL);
```

the locale is not changed and the current active locale is used. Applications that invoke this setting can select one appropriate language corresponding to its current active locale.

Programming Tips

Do not include language-specific resources, such as string characters, in application source code; use resource files or message catalogue files for language-specific resources.

Applications coded for locale-sensitivity in this manner can output DBCS/SBCS and input SBCS. If your application also supports DBCS input, you must create an **XmInputMethod** widget.

Suggested Reading

Related Information

The sample source code titled A Locale-sensitive Application: **loctest.c** in Appendix A: Sample Source Code contains an example of the use of the C-language **setlocale** subroutine to provide support for a national language in a **libXm** application.

The **setlocale** subroutine and **XmInputMethod** widget.

How to Set a Running Environment for Specific Languages

Procedure

Before a locale-sensitive application is executed, the environment should be appropriately declared. It must be set to allow the application to use the specific language codeset, Input Method, profile, dictionaries, and fonts. The application can use different languages without being recompiled if the user prepares multiple resource files in different languages.

To accomplish these tasks, complete the following steps in the order shown:

1. Set the **LANG** environment variable to a specific language by entering one of the following commands from the AIX command line:

   - If you are issuing your command from a Korn shell, enter:

     ```
     LANG=value
     ```

   - If you are issuing your command from a C shell, enter:

     ```
     setenv LANG value
     ```

   The *value* parameter is country-specific and language-specific. It supports the following languages as well as many others;

value	**Language (and Running Environment)**
En_US.pc850	Supports American English using an English keyboard
Jp_JP.pc932	Supports Japanese using a Japanese keyboard

2. To allow your application to use a specific resource file, prepare the resource file and select an appropriate file name. Resource files can be prepared under any of the following file names, but keep two things in mind as you choose an appropriate file name: resource files are searched in the order in which they are listed here, and if the same resource is found in more than one resource file, resources in files near the top of the list will always be overwritten by equivalent resources in files further down the list:

   - **$HOME/.Xdefaults**

   The resources in this file are used for any Enhanced X-Windows application. For example, if the **$HOME** variable is set to **/u/laura**, the resource file **/u/laura/.Xdefaults** is always read when the Enhanced X-Windows application is activated.

   - **$HOME/$LANG/%CLASSNAME**

   The resources in this file are selected corresponding to the **LANG** environment variable and the application class name. For example, if **$LANG** is set to `Jp_JP.pc932`, and **$HOME** is set to **/u/satomi**, the resource file **/u/satomi/Jp.JP.pc932/loctest** is always read when the application named **loctest** is activated.

   - **$XENVIRONMENT**

   Any file whose filename is set to the **XENVIRONMENT** environment variable is, in effect, a resource file. For example, if **$XENVIRONMENT** is set to **/u/satomi/resJ**, the resource file **/u/satomi/resJ** is read each time the application is initialized, provided the **XENVIRONMENT** is still active.

   Resource files containing strings must always be prepared and declared appropriately before the application is executed.

   If the same resource is found in more than one resource file, the resource in the file later in the list overwrites the earlier resource. For example, if the **fontList** resource is set in the file named **$HOME/.Xdefaults** and in the file named **$HOME/$LANG/%CLASSNAME**, the **fontList** resource in **$HOME/$LANG/%CLASSNAME** overrides the one in **$HOME/.Xdefaults**.

   There are several other ways to prepare the default resource file, including the following:

   /usr/lib/X11/$LANG/app-defaults/%CLASSNAME

   /usr/lib/X11/app-defaults/%CLASSNAME

   The **CLASSNAME** parameter is the name of the specific application to which the resource file passes values. For example, if there is a resource file for an application of the name **abc**, the resource file can be called **abc** and placed in **/usr/lib/X11/$LANG/app-defaults** or in **/usr/lib/X11/app-defaults**.

   This sample resource file is for the sample program **immtest.c**.

```
*fontList:                   Rom22
*dialogTitle:                loctest AIXwindows
*Exit Button*labelString:    Push Here to Exit
*textwidget*value:           USA English (En_US.pc850)
```

   If this sample resource file is named **$HOME/resE**, where **$HOME** is **/u/laura**, the **LANG** and **XENVIRONMENT** environment variables are set up as follows:

```
LANG=En_US.pc850
XENVIRONMENT=/u/laura/resE
```

   Enter the following at a Korne shell command line:

```
LANG=En_US.pc850
export XENVIRONMENT=/u/laura/resE
```

Or, enter the following at a C shell command line:

```
setenv LANG=En_US.pc850
setenv XENVIRONMENT=/u/laura/resE
```

If **immtest.c** is to be run with the **LANG** environment variable set to `Jp_JP.pc932`, a file named **$HOME/resJ** can be specified as follows:

In this case, the **LANG** and **XENVIRONMENT** environment variables can be set as described above, with **LANG** set to `Jp_JP.pc932` and **XENVIRONMENT** set to **/u/satomi/resJ** (where the Japanese user is "Satomi").

The **LANG** and **XENVIRONMENT** environment variables can be conveniently set vis shall scripts. For example, for the sample resource file **/u/laura/resE**, the corresponding Korne shell script of two lines reads:

```
LANG=En_US.pc850
export XENVIRONMENT=/u/laura/resE
```

If this file is named **goE**, enter the following from a Korne shell command line:

```
. goE
```

Similarly, a C shell script to set the environment for the sample resource file **/u/satomi/resJ**, contains the two lines:

```
setenv LANG—Jp_JP.pc932
setenv XENVIRONMENT=/u/satomi/resJ
```

If this file is named **goJ**, enter the following from a C shell command line:

```
source goJ
```

3. Determine the specific names of all the fonts you must pass to your application and specify this to the **fontList** resource in a resource file, as demonstrated in the following examples:

   - To pass a single English font to the application:

     ```
     *fontList: Rom22
     ```

   - To pass four Japanese fonts to the application:

     ```
     *fontList: RomanKn17,Kanji17,Kana17,IBM_JPN17
     ```

   Providing a list of specific fonts in the **fontList** resource within a resource file is not necessarily required for applications using an SBCS language only. This is because a default font ("fixed") is provided by each widget/gadget class that specifies fonts. However, you must provide font information explicitly (by means of a resource file or the Enhanced X-Windows **XtSetValues** subroutine) for any application using a DBCS language. DBCS input/output requires DBCS fonts in a **fontList** resource and no widget/gadget class provides DBCS fonts in its default **fontList** resource. An application using a DBCS language does not run without access to DBCS characters through the **fontList** resource. In addition, the **XmInputMethod** widget associated with `JP_Jp.pc932` always displays DBCS characters in its statusWindow even if no widget/gadget in the widget tree has actual DBCS character data.

   In the case of a Japanese locale, such as `Jp_JP.pc932`, four fonts must be specified for the **fontList** resource. All four fonts must be DBCS fonts. The first font should be an alphanumeric font, such as `RomanKn12`. (Note that this font covers only ASCII characters up to 127.) The second font should be a Kanji font, such as `Kanji12`. The third font should be a Katakana font, such as `Kana12`. The last font should be a Gaiji font, such as `IBM_JPN12`. Even if one of these fonts is not used, four must still be specified.

Note: If the **LANG** environment variable is set to `Jp_JP.pc932`, `Jp_JP`, `En_JP.pc932`, or `En_JP`, it is mandatory that the **fontList** resource gets four fonts even if only ASCII is to be displayed. For example, if `LANG=Jp_JP.pc932`, the **fontList** resource could be: fixed, fixed, fixed, fixed. (One of the fonts could be a Japanese font, of course.) However, even in the case of non-Kanji usage, if `LANG=Jp_JP.pc932`, four fonts in the **fontList** resource is mandatory.

4. If it is important to force an application to use the specific language of the menu, title, or label, the string in that language can be provided to applications from the following resources in a resource file.

   - dialogTitle
   - filterLabelString
   - applyLabelString
   - cancelLabelString
   - helpLabelString
   - okLabelString
   - listLabelString
   - selectionLabelString.

   The following example shows a sample SBCS resource file for an application named **loctest**. This resource file could also be named **$HOME/En_US.pc850/loctest**. It is a sample resource file for the sample program **loctest.c**. (Note: this resource file can also be used for the sample program **convtest.c**.)

```
*fontList:                Rom22
*dialogTitle:             loctest AIXwindows
*filterLabelString:       File Filter to Select Files
*applyLabelString:        Apply
*cancelLabelString:       Cancel
*helpLabelString:         Help
*okLabelString:           OK
*listLabelString:         File List
*selectionLabelString:    Selected File
```

Note: With this resource file, only SBCS characters can be supported. This is because the **fontList** resource is specified to be a single, PC850 (SBCS), font. Thus, the assumption here is that the **LANG** environment variable is set to one of the SBCS locales—such as `En_US.pc850` or `Fr_FR.pc850`, for example—and that the user is using an SBCS keyboard.

The next example shows a sample DBCS resource file for an application named **loctest**. This resource file could be named **$HOME/Jp_JP.pc932/loctest**. It is the DBCS version for the sample the sample program **loctest.c**.

```
*fontList:                *RomanKn17,Kanji17,Kana17,IBM_JPN17
*dialogTitle:             loctest AIXwindows
*filterLabelString:       File Filter to Select Files
*applyLabelString:        Apply
*cancelLabelString:       Cancel
*helpLabelString:         Help
*okLabelString:           OK
*listLabelString:         File List
*selectionLabelString:    Selected File
```

Note: This resource file specification assumes the **LANG** environment variable is set to `Jp_JP.pc932` or `Jp_JP`. It is also assumes that a Japanese keyboard and AIX for RISC System/6000 with Japanese Kanji Support is being used.

String data such as "File List" can be expressed in Japanese, as indicated in Figure 6. It is shown here in English because technical restrictions limit the display of Japanese in this document.

There is a restriction in specifying a backslash (x'5c') in a resource file. To pass a backslash as data from a resource file to an application, enter two contiguous backslashes in the resource file. This restriction requires special attention in DBCS because a DBCS character sometimes includes x'5c' in its last byte so that the backslash is not visible. For example, do not attempt to pass a backslash to the filterLabelString resource of an application as follows:

```
*filterLabelString: \
```

Tthe resource file cannot pass a backslash to the application correctly because of a restriction in the Enhanced X-Windows Toolkit. The following entry is correct:

```
*filterLabelString: \\
```

This restriction also applies to code points of DBCS (kanji) in general. If any byte in DBCS code has the same code as a backslash (x'5c'), you must insert an additional backslash next to the original backslash. The backslash appears as the currency yen symbol in the Japanese character set. So, one yen symbol (backslash) should be inserted immediately after the kanji character. For example, if you attempt to pass the string "DISPLAY" in Japanese (expressed in 2 DBCS characters, x'955c' and x'8ea6' in hexadecimal code and pronounced as HYOUJI) to an application resource filterLabelString as follows:

```
*filterLabelString: x'955c8ea6'
```

the above resource file would not pass x'5c' to the application correctly because of a restriction in the X toolkit (Xt). (This resource file is expressed in hexadecimal code because kanji cannot be written in this documentation. Note that a kanji character x'955c' contains x'5c').

To specify a kanji x'955c' in a resource file and have it passed to the application correctly, x'5c' should be inserted after the kanji x'955c' in the resource file as follows:

```
*filterLabelString: x'955c5c8ea6'
```

Even if the yen currency symbol is not visible, this restriction should be kept in mind. In order to investigate if a kanji

Note: In order to investigate if a kanji character contains x'5c', refer to "N:SN18–0101" (available to Japanese customers only.)

5. Determine if your application must make use of strings in different languages in menus, titles, or labels. If it must, you can avoid the need for re-compilation of your source code by instructing users to prepare multiple resource files in different languages.

Suggested Reading

Related Information

Refer to Changing the Appearance and Behavior of AIXwindows for a Specific National Language for the search order of resource file of the **mwm**.

For a full specification of the search order of the default resource file, refer to information regarding the **XtInitialize** subroutine.

Refer to Restriction on Retrieving Resources from Resource Files for additional information on this subject.

How to Create an XmInputMethod Widget

Prerequisite Tasks or Conditions

1. Ensure that the include files and other header files listed in your source code are available and accessible.
2. Determine if your application requires Single-Byte Character Support (SBCS) or Double-Byte Character Support (DBCS).

   Note: An application must contain an **XmInputMethod** widget to process DBCS input. However, an **XmInputMethod** widget is not necessarily required in applications that process SBCS input only.

   The **XmInputMethod** widget can also be used for SBCS input with the same application program interface. This feature helps programmers develop applications that can be ported across SBCS abd DBCS environments.

Procedure

To create an **XmInputMethod** widget, follow the indicated steps in the order shown:

1. If you decide to use a national language, add the C-language **setlocale** subroutine to your source code. This subroutine arranges for the output of NLS characters and for the **XmNinputMethodWidget** resource to be set to the **LANG** environment variable.
2. Before you add an **XmInputMethod** widget to your application source code, decide whether you should create the widget directly or indirectly. Then follow the steps indicated for the option you select:
   - How to Create an **XmInputMethod** Widget Directly
   - How to Create an **XmInputMethod** widget Indirectly

Suggested Reading

Related Information

The sample source code titled A Program that Creates an **XmInputMethod** Widget Directly: **immtest.c** in Appendix A: Sample Source Code contains an example of the explicit creation of an **XmInputMethod** widget.

The sample source code titled A Program that Uses **XmCreateFileSelectionDialog** to Create an **XmInputMethod** Widget Indirectly: **convtest.c** in Appendix A: Sample Source Code contains an example of the implicit creation of an **XmInputMethod** widget.

The **setlocale** subroutine and **XminputMethod** widget.

How to Create an XmInputMethod Widget Directly

Prerequisite Tasks or Conditions

1. Ensure that the include files and other header files listed in your source code are available and accessible.
2. Determine if your application requires Single-Byte Character Support (SBCS) or Double-Byte Character Support (DBCS).

   Note: An application must contain an **XmInputMethod** widget to process DBCS input. However, an **XmInputMethod** widget is not necessarily required in applications that process SBCS input only.

   The **XmInputMethod** widget can also be used for SBCS input with the same application program interface. This feature helps programmers develop applications that can be ported across SBCS abd DBCS environments.

Procedure

To explicitly create an **XmInputMethod** widget, follow the indicated steps in the order shown:

1. Use one of the following subroutines (or any equivalent AIXwindows creation subroutine) to create an **XmInputMethod** widget as a child of a widget that inherits from the **Shell** class:
   - **XtCreateManagedWidget**
   - **XmCreateInputMethod**.
2. Create any container widget as a **workWindow** structure.
3. Use the the Enhanced X-Windows **XtSetValues** subroutine to associate a **workWindow** structure with the **XmInputMethod** structure you created in step 1 of this procedure.
4. If necessary, control the AIX Input Method by altering the following resources:

Set the **XmNpreEditType** resource to one of the following pre-edit modes by using the Enhanced X-Windows **XtSetValues** subroutine or adding an appropriate entry to an appropriate resource file:

- XmPREEDIT_POPUP
- XmPREEDIT_UNDER

If all users of the application require a single language-specific Input Method, explicitly set the **XmNInputMethodWidget** resource to that language. For example, Japanese users require a Japanese Input Method that is obtained by setting the **XmNinputMethodWidget** resource to `Jp_JP.pc932`.

Notes:

1. If you do not set the **XmNinputMethod** resource explicitly, it is automatically set for the locale assigned by the C-language **setlocale** subroutine. This has the effect of making the application locale-sensitive. If you added the **setlocale**

subroutine to your source code as suggested in step 1, do not override the resulting locale-sensitivity by explicitly setting the **XmNinputMethod** resource to a specific language.

2. If the input method you request is not supported by the current active locale assigned by the **setlocale** subroutine, the requested **XmNInputMethod** resource is ignored when the **XmInputMethod** widget is created at run time. This results in a warning message being displayed and the AIX Input Method for the current active locale assigned by the **setlocale** subroutine being loaded.

Suggested Reading

Related Information

The sample source code titled A Program that Creates an **XmInputMethod** Widget Directly: **immtest.c** in Appendix A: Sample Source Code contains an example of the explicit creation of an **XmInputMethod** widget.

The **setlocale** subroutine, **XmCreateInputMethod** subroutine, **XtCreateManagedWidget** subroutine, **XtSetValues** subroutine; the **XmInputMethod** widget.

How to Create an XmInputMethod Widget Indirectly

Prerequisite Tasks or Conditions

1. Ensure that the include files and other header files listed in your source code are available and accessible.
2. Determine if your application requires Single-Byte Character Support (SBCS) or Double-Byte Character Support (DBCS).

   Note: An application must contain an **XmInputMethod** widget to process DBCS input. However, an **XmInputMethod** widget is not necessarily required in applications that process SBCS input only.

   The **XmInputMethod** widget can also be used for SBCS input with the same application program interface. This feature helps programmers develop applications that can be ported across SBCS abd DBCS environments.

Procedure

To indirectly create an **XmInputMethod** widget, create an instance of an **XmInputMethod** widget using any of the following AIXwindows Toolkit subroutines:

- Convenience Creation Dialog subroutines
- The **XmCreateMainWindow** subroutine.

Indirectly Creating an XmInputMethod Widget Using a Convenience Creation Dialog Subroutine

To implicitly create an **XmInputMethod** widget using a convenience creation dialog subroutine, complete the following steps in the order shown:

1. Create a dialog widget using one of the following convenience creation subroutines:

   XmCreateBulletinBoardDialog

   XmCreateFileSelectionDialog

   XmCreateFormDialog

XmCreatePromptDialog

XmCreateSelectionDialog

XmCreatePromptDialog.

2. Specify indirect creation of an **XmInputMethod** widget by using the Enhanced X-Windows **XtSetArg** subroutine to set the **XmNinputMethodWidget** resource to any language-specific AIX Input Method or to the XmDEFAULT_INPUT_METHOD value.

   Note: If all users of the application require a single language-specific Input Method—Japanese, for example—explicitly set the **XmInputMethodWidget** resource to that language constant (`Jp_JP.pc932`).

   If you do not set the **XmNinputMethodWidget** resource explicitly, it is automatically set for the locale assigned by the C-language **setlocale** subroutine. This has the effect of making the application locale-sensitive. If you added the **setlocale** subroutine to your source code as suggested in step 1, do not override the resulting locale-sensitivity by explicitly setting the **XmNinputMethodWidget** resource to a specific language.

   If the input method you request is not supported by the current active locale assigned by the **setlocale** subroutine, when the **XmInputMethod** widget is created at run time the requested **XmNinputMethodWidget** resource is ignored. This results in a warning message being displayed and the AIX Input Method for the current active locale assigned by the **setlocale** subroutine being loaded.

Indirectly Creating an XmInputMethod Widget Using the XmCreateMainWindow Subroutine

To implicitly create an **XmInputMethod** widget using a convenience creation dialog subroutine, complete the following steps in the order shown:

1. Create a widget that inherits from the **Shell** widget class and that is the parent widget of a **mainWindow** child widget.
2. Use the **XmCreateMainWindow** subroutine to create a mainWindow widget as a child of the widget that you created in step 1.
3. Specify indirect creation of an **XmInputMethod** widget by using the Enhanced X-Windows **XtSetValues** subroutine to set the value of the **XmNinputMethodWidget** resource to any language-specific AIX Input Method or to the XmDEFAULT_INPUT_METHOD value.

   Note: If all users of the application require a single language-specific Input Method—for example, Japanese—explicitly set the **XmInputMethodWidget** resource to that language constant (`Jp_JP.pc932`).

   If you do not set the **XmNinputMethodWidget** resource explicitly, it is automatically set for the locale assigned by the C-language **setlocale** subroutine. This has the effect of making the application locale-sensitive. If you added the **setlocale** subroutine to your source code as suggested in step 1, do not override the resulting locale-sensitivity by explicitly setting the **XmNinputMethodWidget** resource to a specific language.

   If the input method you request is not supported by the current active locale assigned by the **setlocale** subroutine, when the **XmInputMethod** widget is created at run time the requested **XmNinputMethodWidget** resource is ignored. This results in a warning message being displayed and the AIX Input Method for the current active locale assigned by the **setlocale** subroutine being loaded.

Suggested Reading

Related Information

The sample source code titled A Program that Uses **XmCreateFileSelectionDialog** to Create an **XmInputMethod** Widget Indirectly: **convtest.c** in Appendix A: Sample Source Code contains an example of the implicit creation of an **XmInputMethod** widget.

The sample source code titled A Program that Uses the **XmCreateMainWindow** Subroutine to Create an **XmInputMethod** Widget Indirectly: **maintest.c** in Appendix A: Sample Source Code contains an example of the implicit creation of an **XmInputMethod** widget.

AIXwindows Subroutines Overview for Programming provides additional information on Understanding AIXwindows Convenience Creation Subroutines.

The **setlocale** subroutine, **XmCreateBulletinBoardDialog** subroutine, **XmCreateFileSelectionDialog** subroutine, **XmCreateFormDialog** subroutine, **XmCreateInputMethod** subroutine, **XmCreateMainWindow** subroutine, **XmCreatePromptDialog** subroutine, **XmCreateSelectionDialog** subroutine, **XtSetArg** subroutine, **XtSetValues** subroutine; the **XmInputMethod** widget.

AIXwindows Widgets and Gadgets Overview for Programming

Individual widgets and gadgets are the building blocks of AIXwindows interfaces. Some widgets and gadgets display information interactively. Others display information without reacting to keyboard input or mouse input. Some widgets and gadgets can change their graphic appearance in response to input and can actively invoke a wide variety of appropriate subroutines. Others are passive containers for interactive widgets and gadgets.

All AIXwindows widgets and gadgets are divided into classes based on shared behavior characteristics and appearance characteristics. These characteristics are called *resources.* The shared behavior resources and appearance resources of each class are passed downward to every other class beneath it in the class hierarchy. A resource passed from one class to another in this manner is said to be *inherited* by each class that receives it.

Physically, an object class is a pointer to a specific data structure. An *instance* of a widget or gadget refers to a specific object derived from a general class. (An individual widget or gadget is said to be *instantiated* from its widget class or gadget class when it is created).

Every instance of an AIXwindows object consists of two data structures: a class data structure describing the appearance and behavior resources of every object in the class and an individual data structure describing additional properties of the specific widget or gadget itself

Every AIXwindows object can be *customized* to some degree by altering the default values governing appearance resources such as the size of the object, the size, thickness, and shading of its borders, the color of foreground and background elements, and the size and style of the text fonts. Resource values are stored in several different locations to promote efficiency and increase flexibility. These locations include:

- Data Structures

Many resource values are stored as default values in the object class data structures. This permits you to apply or alter values at the highest level (the class level). Alteration of these resource values automatically affects all subclasses in the hierarchy because they inherit the altered resources.

- Source Code

Resource values affecting a specific client application can be hard-coded into the argument list of each new object instantiated in the source code of that application.

- Default Files

Resource values affecting a general class of applications are stored in a single application defaults file (the **/usr/lib/X11/$LANG/app_defaults** file). Resources earmarked for customization by individual users typically include lists of resource values in each user's personal defaults file (the **$HOME/.Xdefaults** file).

The AIXwindows Toolkit provides more than 150 subroutines to help you generate, alter, and manipulate AIXwindows widgets, gadgets, and the compound objects that can be created with them. These subroutines are divided into the following categories based on shared functionality:

- Creation Subroutines

These subroutines create an instance of the single widget or gadget named in the subroutine. One creation subroutine is provided for each instantiable widget class and gadget class in the AIXwindows Toolkit. Creation subroutines provide access to more object resources than do other subroutines.

- Convenience Creation Subroutines

These subroutines create an instance of the compound object named in the subroutine.

- Support Subroutines.

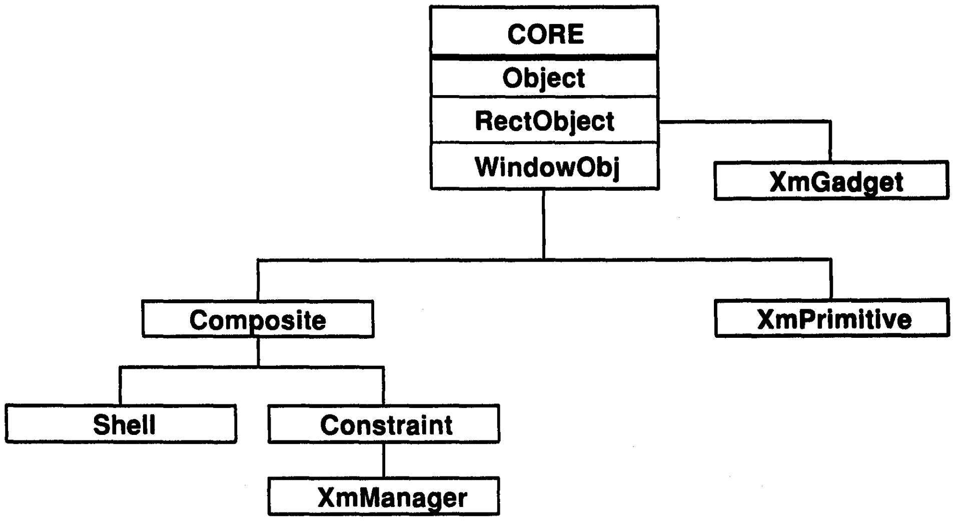
These subroutines provide a means of altering the appearance, functionality, or interactivity of objects created with other subroutines.

Class Hierarchies and Resource Inheritance

The path of inheritance for the nine basic object classes in the AIXwindowsToolkit begins with three X-Windows metaclasses:

- **Object** (All AIXwindows objects inherit resources from this metaclass.)
- **RectObject** (All rectangular AIXwindows objects inherit resources from this metaclass.)
- **WindowObj** (All window-based AIXwindows objects inherit resources from this metaclass.).

These three metaclasses are combined into a single **Core** class that is the uppermost class in the AIXwindows top-level class hierarchy.



AIXwindows Top-level Class Hierarchy

No objects are ever instantiated from the **Core** class; it serves as a supporting superclass that provides common resources to all other classes in the hierarchy.

The next two top-level classes in the hierarchy are layered beneath the **Core** class. They have no other top-level classes beneath them, but each has several subclasses that can inherit some or all of the new class resources. Together they provide the primary interactive objects required by all AIXwindows interfaces. They are known as:

- **XmGadget** (Six subclasses inherit resources from this class.)
- **XmPrimitive** (Ten subclasses inherit resources from this class.)

Each of the six subclasses in the **XmGadget** class produces gadgets that are the functional equivalents of the widgets produced by the **XmPrimitive** class. Although widgets and gadgets are similar in many ways, several significant differences are worth noting. These differences include the following:

- Windows

All widgets are window-based objects. They inherit resources from the **WindowObj** class. Gadgets do not inherit the **WindowObj** class resources (all visual resources must be referenced from the parent widget). For this reason gadgets are often described as *windowless* widgets. They do not support the windows, translations, actions, or popup children provided by the equivalent **XmPrimitive** widgets.

- Overhead

Gadgets improve relative performance and reduce data storage space requirements when compared to widgets. These improvements apply both to client (application) processes and server (X-Windows) processes, and are often quite significant.

Note: Use gadgets instead of their equivalent widgets whenever possible to reduce overhead and increase the efficiency of each interface.

Next in the hierarchy is the following metaclass:

- **Composite** (Twenty-five subclasses inherit resources from this class.).

Like the three metaclasses that make up the **Core** class, the **Composite** class does not instantiate any widgets. Instead, it provides resources to the subclasses that follow it in the hierarchy.

There are two additional metaclasses directly beneath the **Composite** class. These metaclasses are called:

- **Constraint** (Fifteen subclasses inherit resources from this class.).
- **Shell** (Eight subclasses inherit resources from this class.)

The **Constraint** class is a metaclass that does not instantiate any widgets. Instead, it provides common resources to the subclasses that follow it in the hierarchy.

The **Shell** class is internal and cannot be instantiated. However, six of the eight subclasses beneath the **Shell** class can be instantiated and provide a means of interfacing with the AIXwindows Window Manager. The X-Windows Toolkit provides some of the underlying shells, and the AIXwindows Toolkit provides the remaining shells.

The final top-level class in the AIXwindows hierarchy falls directly beneath the **Constraint** class. Like the **Constraint** class, it is also a metaclass that acts as supporting superclass for the subclasses beneath it. This class is called:

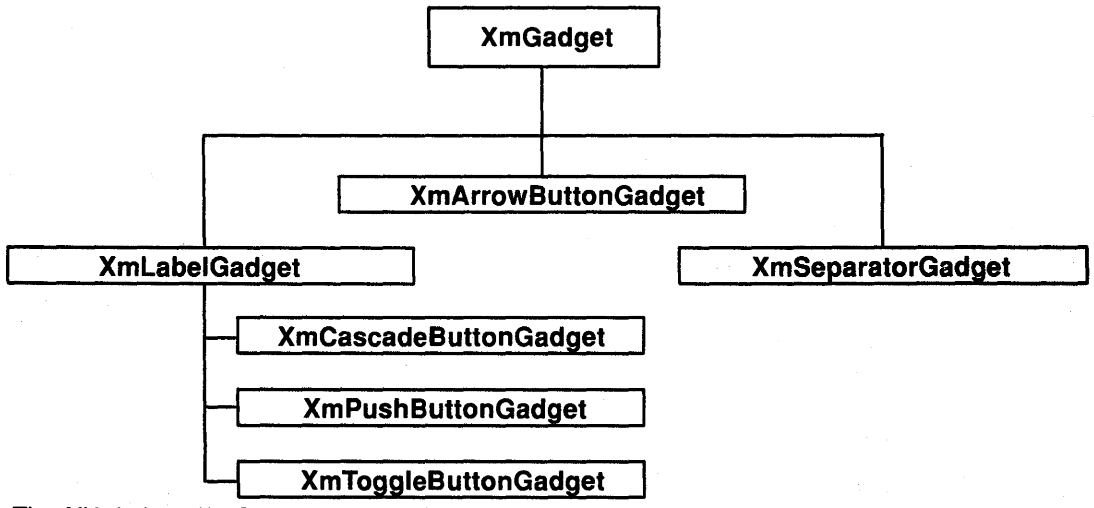
- **XmManager** (Fourteen subclasses inherit resources from this class.)

The **XmManager** class cannot be instantiated. It supports visual resources, graphics contexts, and traversal resources.

The **XmGadget** gadget class hierarchy consists of the six following gadget subclasses, each of which is essentially equivalent to its like-named **XmPrimitive** widget subclass:

- **XmSeparatorGadget**
- **XmArrowbuttonGadget**
- **XmLabelGadget**
- **XmCascadeButtonGadget**
- **XmPushButtonGadget**
- **XmToggleButtonGadget**.

Each subclass can inherit some or all of the resources of the classes that precede it in the hierarchy.



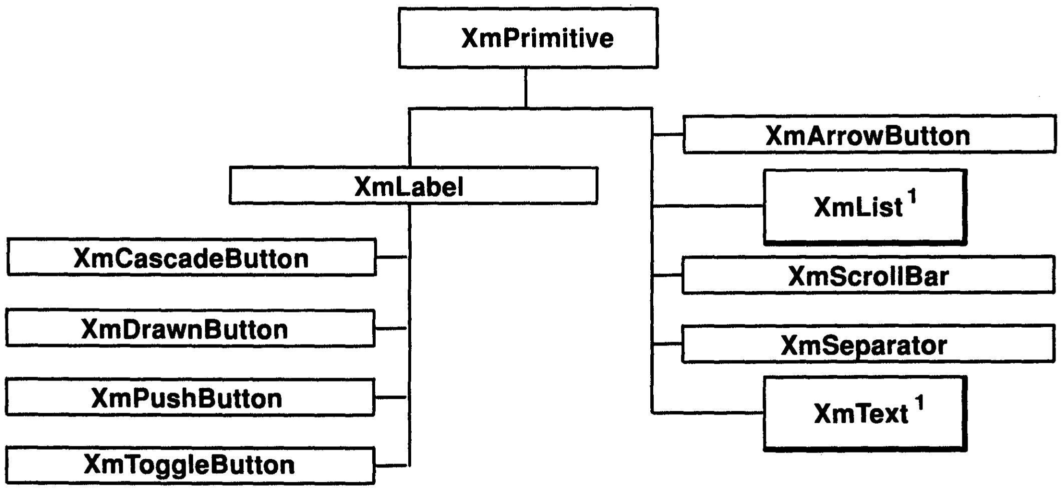
The AIXwindows **XmGadget** Gadget Class Hierarchy

The **XmPrimitive** class hierarchy consists of the ten following subclasses (the first six of which are essentially equivalent to the like-named **XmGadget** gadget subclasses):

- **XmSeparator**
- **XmArrowButton**
- **XmLabel**
- **XmCascadeButton**
- **XmPushButton**
- **XmToggleButton**
- **XmList**
- **XmScrollBar**
- **XmText**
- **XmDrawnButton**.

Each subclass can inherit some or all of the resources of the classes that precede it in the hierarchy.

Two of these subclasses (**XmList** and **XmText**) are considered *specialized subclasses.* This is because they are inherently more complex to use (and support more sophisticated interactivity) than the other subclasses in the **XmPrimitive** class hierarchy. For example, an **XmList** widget is a general purpose widget that allows you to make a selection from a list of items. The application defines an array of compound strings, each of which becomes an item in the list. You can set the number of items in the list that are to be visible. You can also choose to have the list appear with an **XmScrollBar** so that you can scroll through the list of items. Items are selected by moving the pointer to the desired item and pressing the mouse button (or a keyboard key defined as *select*). The selected item is displayed in inverse color.



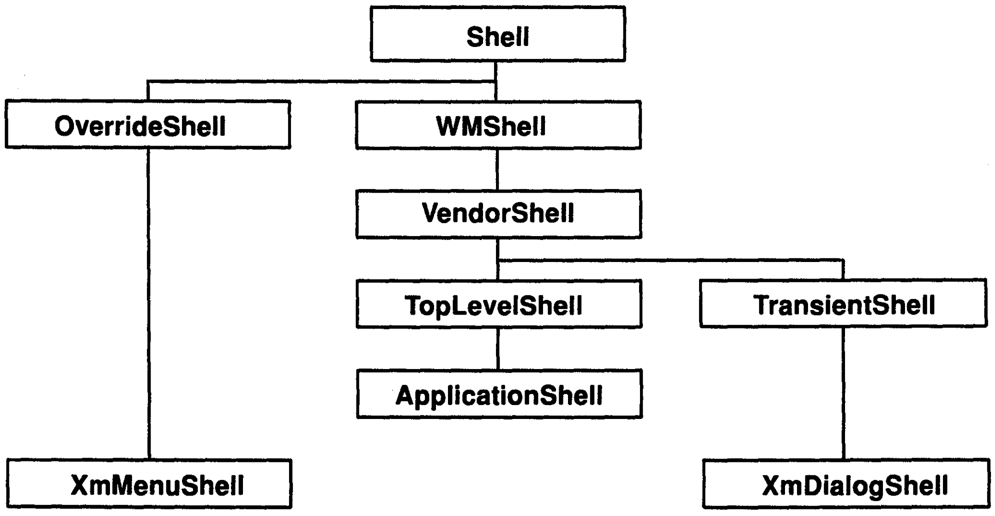
[1] specialized subclass

The AIXwindows **XmPrimitive** Widget Class Hierarchy

Shell widgets typically serve as the parents of the widgets at the top of a client application interface widget tree. Most Shell widgets are invisible but have a direct impact on how their children are displayed. The **Shell** widget class hierarchy consists of the eight following subclasses:

- **TransientShell**
- **XmDialogShell**
- **WMShell**
- **VendorShell**
- **TopLevelShell**
- **ApplicationShell**
- **OverrideShell**
- **XmMenuShell.**

The classes **Shell, WmShell,** and **VendorShell** are considered *private* because they cannot be instantiated. All other subclasses are considered *public* because they can be instantiated. Each subclass can inherit some or all of the resources of the classes that precede it in the hierarchy.



AIXwindows **Shell** Widget Class Hierarchy

The **XmManager** widget class hierarchy consists of the following thirteen subclasses:

- **XmScrolledWindow**
- **XmMainWindow**
- **XmRowColumn**
- **XmBulletinBoard**
- **XmDrawingArea**
- **XmPanedWindow**
- **XmFrame**
- **XmScale**

- **XmForm**
- **XmMessageBox**
- **XmSelectionBox**
- **XmCommand**
- **XmFileSelectionBox**.

Each subclass can inherit some or all of the resources of the classes that precede it in the hierarchy.

Two of these subclasses (**XmRowColumn** and **XmForm**) are considered *specialized subclasses.* This is because they are inherently more complex to use (and support more sophisticated interactivity) than the other subclasses in the **XmManager** class hierarchy. For example, an **XmRowColumn** widget is a general purpose row-column manager capable of containing any class of widget or gadget as a child. Although it requires no special information about the functioning of its children, it does provide support for several different layout styles involving changes in sizing, orientation, packing, children spacing, and margin spacing. The three types of menu systems provided by the AIXwindows Toolkit (Popup menu systems, Pulldown menu systems, and Option menu systems) are all based upon **XmRowColumn** widgets configured in different ways (as **Popup MenuPanes, Pulldown MenuPanes**, or as **Menu Bars** or **Option Menus**).

Understanding AIXwindows Creation Subroutines

There are 30 AIXwindows *creation subroutines.* Each creation subroutine creates a single instance of the widget or gadget named in the subroutine.

Widget Creation Subroutines

- **XmCreateArrowButton**
- **XmCreateBulletinBoard**
- **XmCreateCascadeButton**
- **XmCreateCommand**
- **XmCreateDialogShell**
- **XmCreateDrawingArea**
- **XmCreateDrawnButton**
- **XmCreateForm**
- **XmCreateFrame**
- **XmCreateFileSelectionBox**
- **XmCreateLabel**
- **XmCreateList**
- **XmCreateMainWindow**
- **XmCreateMenuShell**
- **XmCreateMessageBox**
- **XmCreatePanedWindow**
- **XmCreatePushButton**

- **XmCreateRowColumn**
- **XmCreateScale**
- **XmCreateScrollBar**
- **XmCreateScrolledWindow**
- **XmCreateSelectionBox**
- **XmCreateSeparator**
- **XmCreateText**
- **XmCreateToggleButton**.

Gadget Creation Subroutines

- **XmCreateArrowButtonGadget**
- **XmCreateCascadeButtonGadget**
- **XmCreateLabelGadget**
- **XmCreatePushButtonGadget**
- **XmCreateSeparatorGadget**
- **XmCreateToggleButtonGadget**.

Understanding AIXwindows Convenience Creation Subroutines

Eighteen *convenience creation* subroutines are included in the AIXwindows Toolkit. These subroutines are called convenience creation subroutines because each subroutine conveniently creates all of the widgets and gadgets required to make a single instance of the compound object named in the subroutine.

Note: Approximately half of the convenience creation subroutines (those ending with the **Dialog** suffix) are also known as *convenience creation dialogs* because each creates all of the widgets and gadgets required to make a single instance of a specific type of compound object called a *convenience dialog*.

The following subroutines create convenience dialogs or other compound objects:

- **XmCreateBulletinBoardDialog**
- **XmCreateErrorDialog**
- **XmCreateInformationDialog**
- **XmCreateFileSelectionDialog**
- **XmCreateFormDialog**
- **XmCreateMessageDialog**
- **XmCreatePromptDialog**
- **XmCreateQuestionDialog**.
- **XmCreateSelectionDialog**
- **XmCreateWarningDialog**
- **XmCreateWorkingDialog**
- **XmCreateMenuBar**

- **XmCreateOptionMenu**
- **XmCreatePopupMenu**
- **XmCreatePulldownMenu**
- **XmCreateRadioBox**
- **XmCreateScrolledList**
- **XmCreateScrolledText**.

Understanding AIXwindows Support Subroutines

AIXwindows *support subroutines* alter the appearance, functionality, or interactivity of the objects created with AIXwindows creation subroutines. Support subroutines are divided into 12 categories based on shared functionality. These categories include:

- Subroutines that Create and Manipulate Compound Strings
- Subroutines that Modify XmList Widget Lists
- Subroutines that Modify Clipboard Data
- Subroutines that Grab or Ungrab Keys or the Keyboard
- Subroutines that Associate an Image With a Name Through Pixmap Caching
- Subroutines That Provide for Resolution Independence
- Subroutines That Provide for Protocol Management and Window Manager Message Management
- Subroutines That Support the Keyboard Interface
- Subroutines that Manipulate Text
- Subroutines that Manipulate Font Lists
- Subroutines that Perform Miscellaneous Tasks
- Subroutines That Are Accessed Automatically

Subroutines that Create and Manipulate Compound Strings

Compound strings are designed to permit messages and other text to be displayed without the necessity of hard-coding language-dependent attributes such as direction, character set, and text. The AIXwindows Toolkit subroutines that create and manipulate compound strings include:

- **XmCvtStringToUnitType**
- **XmString**
- **XmStringBaseline**
- **XmStringByteCompare**
- **XmStringCompare**
- **XmStringConcat**
- **XmStringCopy**
- **XmStringCreate**
- **XmStringCreateLtoR**

- **XmStringDirectionCreate**
- **XmStringDraw**
- **XmStringDrawImage**
- **XmStringDrawUnderline**
- **XmStringEmpty**
- **XmStringExtent**
- **XmStringFree**
- **XmStringFreeContext**
- **XmStringGetLtoR**
- **XmStringGetNextComponent**
- **XmStringGetNextSegment**
- **XmStringHeight**
- **XmStringInitContext**
- **XmStringLength**
- **XmStringLineCount**
- **XmStringNConcat**
- **XmStringNCopy**
- **XmStringPeekNextComponent**
- **XmStringSegmentCreate**
- **XmStringSeparatorCreate**
- **XmStringWidth**.

Subroutines that Modify XmList Widget Lists

The following AIXwindows subroutines modify the lists displayed within **XmList** widgets:

- **XmListAddItem**
- **XmListAddItemUnselected**
- **XmListDeleteItem**
- **XmListDeletePos**
- **XmListDeselectAllItems**
- **XmListDeselectItem**
- **XmListDeselectPos**
- **XmListItemExists**
- **XmListSelectItem**
- **XmListSelectPos**
- **XmListSetBottomItem**
- **XmListSetBottomPos**

- **XmListSetHorizPos**
- **XmListSetItem**
- **XmListSetPos**.

Subroutines that Modify Clipboard Data

The AIXwindows Toolkit provides a clipboard to hold data that is being transferred between applications. The following cut and paste subroutines permit you to modify the type of data on the clipboard or the value of data on the clipboard:

- **XmClipboardCancelCopy**
- **XmClipboardCopy**
- **XmClipboardCopyByName**
- **XmClipboardEndCopy**
- **XmClipboardEndRetrieve**
- **XmClipboardInquireCount**
- **XmClipboardInquireFormat**
- **XmClipboardInquireLength**
- **XmClipboardInquirePendingItems**
- **XmClipboardLock**
- **XmClipboardRegisterFormat**
- **XmClipboardRetrieve**
- **XmClipboardStartCopy**
- **XmClipboardStartRetrieve**
- **XmClipboardUndoCopy**
- **XmClipboardUnlock**
- **XmClipboardWithdrawFormat**.

Subroutines that Grab or Ungrab Keys or the Keyboard

The AIXwindows Toolkit provides four subroutines that isolate and redirect keyboard events within a widget hierarchy. These subroutines call equivalent X-Windows subroutines to accomplish their purpose, but the equivalent subroutines should not be used directly. If they are used, the AIXwindows system is not notified of grabs or ungrabs and cannot process them correctly.

- **XtGrabKey**
- **XtUngrabKey**
- **XtGrabKeyboard**
- **XtUngrabKeyboard**.

Subroutines that Associate an Image With a Name Through Pixmap Caching

The AIXwindows Toolkit provides four pixmap caching subroutines to enable you to associate each bit-mapped image with a name. This association, once established for all widgets that have pixmap resources, permits the generation of pixmaps through references to a **.Xdefaults** file (by pixmap name) and through a list of parameters (by pixmap image). The AIXwindows system automatically maintains a cache of all pixmaps to improve performance and reduce data storage requirements. The following subroutines facilitate the process of associating each bit-mapped image with a name:

- **XmInstallImage**
- **XmUninstallImage**
- **XmGetPixmap**
- **XmDestroyPixmap**.

Subroutines That Provide for Resolution Independence

All widgets in the AIXwindows Toolkit support a mechanism called *resolution independence* that allows client applications to create and display images that are the same physical size regardless of the resolution of the display. Resolution independence ensures that AIXwindows interfaces are compatible with a wide range of systems and display screens. The following subroutines are instrumental in the creation of resolution independent widgets:

- **XmSetFontUnit**
- **XmConvertUnits**.

Subroutines That Provide for Protocol Management and Window Manager Message Management

The AIXwindows Toolkit provides a set of inter-client communication subroutines that augment the X-Windows protocols and subroutines. The following AIXwindows Toolkit subroutines provide for protocol management and window manager message management:

- **XmInternAtom**
- **XmAtomToName**
- **XmAddProtocols**
- **XmRemoveProtocols**
- **XmActivateProtocol**
- **XmDeactivateProtocol**
- **XmAddProtocolCallback**
- **XmRemoveProtocolCallback**
- **XmSetProtocolHooks**

Subroutines That Support the Keyboard Interface

The AIXwindows keyboard interface provides for interaction with a client application through a keyboard as well as (or in place of) a mouse. The keyboard interface includes keyboard focus and traversal as well as keyboard input processing for individual widgets. AIXwindows uses the the concept of *tab groups* to organize **XmPrimitive** widgets for more efficient traversal within and between groups. The following subroutines provide a means of defining tab groups within an application:

- **XmAddTabGroup**
- **XmRemoveTabGroup**.

Subroutines that Manipulate Text

The following subroutines manipulate the text contained in **XmText** widgets:

- **XmTextClearSelection**
- **XmTextGetEditable**
- **XmTextGetMaxLength**
- **XmTextGetSelection**
- **XmTextGetString**
- **XmTextReplace**
- **XmTextSetEditable**
- **XmTextSetMaxLength**
- **XmTextSetSelection**
- **XmTextSetString**.

Subroutines that Manipulate Font Lists

The following AIXwindows Toolkit subroutines manipulate font lists:

- **XmFontListAdd**
- **XmFontListCreate**
- **XmFontListFree**.

Subroutines that Perform Miscellaneous Tasks

The following AIXwindows Toolkit subroutines perform tasks that are described in tne reference material for each subroutine:

- **XmCascadeButtonHighlight**
- **XmCommandAppendValue**
- **XmCommandError**
- **XmCommandGetChild**
- **XmCommandSetValue**
- **XmFileSelectionBoxGetChild**
- **XmFileSelectionDoSearch**
- **XmGetMenuCursor**
- **XmMainWindowSep1**

- **XmMainWindowSep2**
- **XmMainWindowSetAreas**
- **XmMenuPosition**
- **XmMessageBoxGetChild**
- **XmOptionButtonGadget**
- **XmOptionLabelGadget**
- **XmResolvePartOffsets**
- **XmScaleGetValue**
- **XmScaleSetValue**
- **XmScrollBarGetValues**
- **XmScrollBarSetValues**
- **XmScrolledWindowSetAreas**
- **XmSelectionBoxGetChild**
- **XmSetMenuCursor**
- **XmToggleButtonGadgetGetState**
- **XmToggleButtonGadgetSetState**
- **XmToggleButtonGetState**
- **XmToggleButtonSetState**
- **XmUpdateDisplay**
- **XmIsMotifWMRunning**

Subroutines That Are Accessed Automatically

Additional functionality is provided by internal AIXwindows subroutines and environment parameters that are accessed automatically as required. Specific areas of automatic functionality include:

- Setting Resource Defaults Dynamically

An internal processing subroutine incorporated into each widget resource set permits widgets to set all *color* and *pixmap* default resource values dynamically (when the widgets are created at run time) rather than statically (in the object code). Color and pixmap resource defaults are set to black and white for a monochrome system, or to a default color scheme (or a color scheme based on the **XmNbackground** resource) for a color system. These subroutines set the foreground color and two shading colors automatically once the background color is specified.

- Using Localized Resource Defaults Files and Environmental Parameters

The AIXwindows system can make use of localized resource defaults files. In general, AIXwindows searches the following locations for resource default values in the following order:

1. Resource default values that are hard-coded within the client application C language code have top priority and take precedence over all other defaults.

2. Resource defaults files that are stored in the **/usr/lib/X11/$LANG/app_defaults** directory have secondary priority. (The names of these files are specified as parameters of the **XtInitialize** subroutine.)
3. An **.Xdefaults** file that is stored in the home directory (**$HOME**) of any given user is given priority only if no other defaults files are encountered.

The X-Windows **XtInitialize** subroutine determines the proper path to the localized resource defaults parameters.

- Using Global Parameters to Store Version and Revision Values

The AIXwindows **XmUseVersion** parameter is a global parameter. It is automatically set to a value equal to the current version value multiplied by 1000 plus the current revision value. (These values are returned by the **XmVersion** macro during the initialization of the widget classes for each interface.)

Understanding AIXwindows Protocol Management Subroutines

A *protocol* is a 32-bit tag that facilitates interactive communication. In the AIXwindows system, client applications communicate with the AIX window manager through the use of a protocol that is either an *X Atom* or an arbitrary long integer parameter whose value is shared by the parties to the protocol communication.

Protocol Management

The AIXwindows Toolkit contains several protocol management subroutines. These subroutines provide a means of interacting with properties that contain atom arrays, client messages, and associated callbacks. These subroutines support the existing entries for the **WM_PROTOCOLS** property (for predefined ICCC protocols) and the **_MOTIF_WM_MESSAGES** property (for the window manager).

Note: Use the AIXwindows subroutine **XmInternAtom** to convert a string (**motif_wm_messages**) to a 32-bit tag and obtain a corresponding *X Atom* (**_MOTIF_WM_MESSAGES**).

The window manager sends protocol messages in the form of client application message events with the *message_type* fields of **ClientMessage** set to the properties and the *data.l[0]* fields set to the protocols. This permits the client application to associate a callback list with the protocol that is invoked when each client message event is received.

Each shell can have one protocol manager per associated property. This protocol manager provides the following services:

- Tracks the state of the protocols, whether they are active or not.
- Tracks the state of the Shell, and creates and updates the protocol property accordingly.
- Processes client application messages and invokes the appropriate callbacks.

Each protocol manager can have multiple registered protocols, and each protocol (identified by its own tag) can have any number of associated client application callbacks in addition to pre-hook and post-hook callbacks (typically registered by the widget set).

Each client application uses the **_MOTIF_WM_MESSAGES** property to indicate to the window manager which messages it is currently handling. A client can add `f.send_msg` entries to the menu in one of the following ways:

- Through use of the **$HOME/.mwmrc** file
- Through use of the **XmNmwmMenu** resource in the **VendorShell** class.

Note: **XmNmwmMenu** is a string that is parsed by the window manager to determine what to display in the system menu and how to react to the selection of each menu item.

Protocol Management Macros

Each of the five general protocol manager subroutines has a corresponding macro. These macros are identified by the uppercase letters **WM** in their name. The primary difference between a general protocol manager subroutine and its corresponding macro is that the subroutine is passed a protocol property in all calls while the macro always forces this property to **WM_PROTOCOLS**.

Note: General protocol manager macros can be quite useful when a client application must interact with ICCC protocols such as **WM_DELETE_WINDOW** or **WM_SAVE_YOURSELF**.

Protocol State Information

It is sometimes useful to allow a protocol's state information (such as its callback lists) to persist, even if the client application resigns temporarily from the interaction. The main reason to do this is to gray out `f.send_msg` labels in the system menu. This is supported by allowing a protocol to be in one of the two following states:

- Active
- Inactive

If the protocol is active and the shell is realized, the property contains the protocol atom. If, however, the protocol is inactive, the protocol atom is not present in the property.

Protocol Callbacks

When the protocol manager receives a client application message associated with a protocol, the manager first determines whether the protocol is active or inactive. If the protocol is active, every callback associated with the protocol is called.

Three types of callback lists can be associated with a protocol, including the following:

- *Pre-hook* callback lists are intended for AIXwindows Toolkit use and are accessed by the **XmSetProtocolHooks** subroutine.
- *Post-hook* callback lists are intended for AIXwindows Toolkit use and are accessed by the **XmSetProtocolHooks** subroutine.
- All other callback lists are for client use. They are accessed by the **XmAddProtocolCallbacks** subroutine and the **XmRemoveProtocolCallbacks** subroutine.

Related Information

Additional information on the use of the **$HOME/.mwmrc** file can be found under Resource Description File Functions in Chapter 2.

For information about the AIXwindows desktop and the window manager associated with AIXwindows interfaces, refer to *AIX General Programming Concepts for IBM RISC System/6000.*

Reference information on the **VendorShell** Widget Class and the **VendorShell** Resource Set is located in the User Interface Volumns(3 and 4) of *AIX Calls and Subroutines Reference for IBM RISC System/6000.*

How to Create an Application Interface with the AIXwindows Toolkit

Prerequisite Tasks or Conditions

None.

Procedure

There are nine basic tasks involved in creating a client application interface using the widgets, gadgets, and subroutines in the AIXwindows Toolkit.

Note: **Tasks 1-3** and **7-9** must be completed once for each interface. **Tasks 4-6** must be repeated once for each object in the interface.

Task 1 Identify and include required header files

Related Functions:

#include <X11/Intrinsic.h>
#include <Xm/Xm.h>
#include <Xm/*widget***.h>**

Task 2 Initialize theX-Windows Toolkit Xt Intrinsics

Related Function:

XtInitialize(...)

Task 3 Add additional top-level windows.

Related Function:

XtAppCreateShell(...)

Task 4 Set up parameter lists for each widget and gadget in the interface

Related Function:

XtSetArg(...)

Task 5 Add appropriate callback routines.

Related Function:

XtAddCallback(...)

Task 6 Create each widget and gadget in the interface.

Related Functions:

XtCreateManagedWidget(
followed by
XtManageChild(*widget***)**

Task 7 Realize all widgets and gadgets.

Related Functions:

XtRealizeWidget(*parent***)**
XtMainLoop()

Task 8 Compile source code and link all relevant libraries

Related Function:

cc +Nd2000 +Ns2000 –o*application application***.c –lXm –lXt –lX11 –lPW**

Task 9 Create appropriate resource defaults files

Related Functions:

/usr/lpp/X11/app–defaults/*class*
$HOME/.Xdefaults

Sample Source Code for AIXwindows Programming Examples

- **option.c**
- **popup.c**

In addition to demonstrating general AIXwindows programming techniques, these sample programs can be used as a foundation for other, more complex AIXwindows interfaces. If you have access to the libraries, files, and toolkits required to create AIXwindows client application interfaces, you can compile, link, and run these sample programs with or without alteration and observe the effects of any changes you make in the source code, defaults files, or inherited resources. To create executable files from these sample source code files, complete the following steps:

1. Use an ASCII text editor such as **vi** or **ed** to create and open an empty file (named **sample1.c** or some other appropriate name) in your **$HOME** directory (or a similar directory created for testing new code).
2. Move the mouse pointer to the head of the first line of source code you intend to copy, and then highlight the remainder of the code by dragging the pointer over the code and releasing the button at the end of the last source code line. (Notice that a new menu option, **Edit**, appears on the menubar when you highlight text in this manner.)
3. Pull the **Edit** menu down and select the **Copy** option.
4. Make certain that your ASCII editor is in the insert mode. Move the pointer into the window containing the empty file. Press the left and right mouse buttons simultaneously to transfer a copy of the code segment from the Edit buffer into the empty file.
5. Repeat the copy process until you have a complete source code program, and then close the file.

Note: If a resource defaults file is commented into the beginning of any of the source code examples you copy, repeat the copy process, moving the contents of the resource defaults file into a new file of their own in the **/usr/lib/X11/app-defaults** directory. This new resource defaults file should have the same basename as the new file containing the sample source code.

6. Create and open a new file named **makefile** in your **$HOME** directory (or a similar directory created for testing new code) and repeat the copy process as necessary to copy the contents of the **makefile** at the end of the two sample programs into the new **makefile** file.
7. You are now ready to compile each program by entering the following command:

   make *ProgramName*

   Do not compile any of the source code files with the **make** command without first determining that all related source files and makefile files are in your **$HOME** directory (or a similar directory created for testing new code).

Task 1: How to Include Header Files for Interfaces Created with the AIXwindows Toolkit

Prerequisite Tasks or Conditions

1. Ensure that the include files and other header files that you intend to list in your C language source code are available and accessible.

Procedure

The first task involved in creating a client application interface is to list several specific include files in the *Include Files* section of the client application source code. To accomplish this task, complete the following steps:

1. List the following standard include files in the include file section of the client application source code, as demonstrated in the *Include Files* section of the sample source code:

– **#include <X11/Intrinsic.h>**	(X-Windows subroutines)
– **#include <Xm/Xm.h>**	(Resource names and defined values)
– **#include <Xm/XmP.h>**	(Information for widget writers)

2. List one pair of the following include files for each widget class name and gadget class name you reference in the client application source code:
   - **#include <Xm/widget_name.h>**
   - **#include <Xm/widget_nameP.h>**
3. If the client application interface makes use of protocol management subroutines and message management subroutines from the AIXwindows Toolkit, list the following include files:
   - **#include <X11/Protocols.h>**
   - **#include <X11/AtomMgr.h>**

Note: The naming conventions for include files differentiate between *public* include files and *private* include files. Public files are available for use with general client applications. They contain class and record pointers, external variables, resource names, and the defined values necessary for a specific client application. Private files typically contain class and instance structure definitions, variables, and the defined values necessary for writing widgets.

Task 2: How to Initialize theX-Windows Toolkit Xt Intrinsics

Prerequisite Tasks or Conditions

1. Ensure that the include files and other header files that you intend to list in your C language source code are available and accessible.
2. Identify and include all appropriate header files, including AIXwindows Toolkit include files.

Procedure

The second task involved in creating a client application interface is to initialize the Xt Intrinsics which are found in the X-Windows Toolkit. The Xt Intrinsics must be intialized before making any other calls to the XtIntrinsics subroutines. The **XtInitialize** subroutine establishes the connection to the display server, parses the command line that invoked the application, loads the resource database, and creates a shell widget to serve as the parent of your application widgets.

By passing the command line that invoked your application to IxInitialize, the subroutine can parse the line to allow users to specify certain resources, such as fonts and colors, for your application at runtime. The **XtInitialize** subroutine scans the command line and removes those options. The rest of your application sees only the remaining options.

There is a second subroutine that may be used to initialize the Xt Intrinsics, the **XtToolkitInitialize** subroutine. This subroutine allows you to decide the tyoe of shell you want to use. However, this subroutine only initializes the toolkit; it does not open the display or create an application shell. To open the display and create an application shell you must use the **XtOpenDisplay** and **XtAppCreateShell** subroutines.

1. Select one of the following initialization subroutines to initialize the Xt Intrinsics:
   - **XtInitialize** subroutine
   - **XtToolkitInitialize** subroutine.
2. If you decide to use the **XtInitialize** subroutine, add the appropriate **XtInitialize** syntax statement to the client application source code. No other initialization subroutines are required because the **XtInitialize** subroutine automatically establishes the connection to the display server and creates an **ApplicationShell** widget parent for the subsequent widget tree hierarchy.

   If you decide to use the **XtToolkitInitialize** subroutine, follow the instructions in step 3.
3. Add the appropriate **XtToolkitInitialize** syntax statement to the client application source code.

   Note: Unlike the **XtInitialize** subroutine, the **XtToolkitInitialize** subroutine does not automatically establish the connection to the display or create an **ApplicationShell** widget parent for widget-gadget children. These tasks are accomplished by the subroutines in steps 4 and 5.
4. Add the appropriate syntax statement for the **XtOpenDisplay** subroutine.
5. Add the appropriate syntax statement for the **XtAppCreateShell** subroutine.

Task 3: How to Add Top-Level Windows to an AIXwindows Interface

Prerequisite Tasks or Conditions

1. Ensure that the include files and other header files that you intend to list in your C language source code are available and accessible.
2. Identify and include all appropriate header files, including AIXwindows include files.
3. Initialize the X-Windows Toolkit Xt Intrinisics.

Procedure

The third task involved in creating a client application interface is to add one or more top level windows to the interface. To accomplish this task, complete the following steps:

1. Identify all of the top-level window widgets required for the client application interface you are coding. Typically, one instance of a **Shell** widget class or subclass is required for each top-level window.

   Note: Top-level window widgets are the widgets at the top level of the widget-instance hierarchy in a given interface. They are often referred to as *top-level windows* or *independent windows* because they manage other widgets but are not managed themselves. These windows typically are instantiated from the **XmMainWindow** widget class or the **XmMessageBox** widget class.

2. Declare all top-level windows in the Forward Declaration section of your source code.
3. Create and realize a main Application window.
4. Develop a widget tree to ensure that all the widgets and gadgets required for the client application interface are included in your client application source code. The existing **ApplicationShell** widget should be at the top of the tree

   Note: If you used the **XtInitialize** subroutine to initialize the X-Windows Toolkit, an **ApplicationShell** widget was automatically created. If you used the **XtToolkitInitialize** subroutine to initialize that Toolkit, you were required to create explicitly an **ApplicationShell** widget with the **XtAppCreateShell** subroutine.

   Each client application interface can have several windows. However, each of these windows needs to be created as the child of a top-level **Shell** superclass parent. Since only one **ApplicationShell** widget is permitted per client interface, **TransientShell** widgets or **TopLevel** widgets must be used for all other top-level windows.
5. List in the appropriate section of the client application source code all widgets and gadgets shown in the widget tree. Include an appropriate syntax statement in the client application source code for each top-level window widget, as provided in the reference material for each widget class.

Task 4: How to Set Up Parameter Lists for AIXwindows Widgets and Gadgets

Prerequisite Tasks or Conditions

1. Ensure that the include files and other header files that you intend to list in your C language source code are available and accessible.
2. Identify and include all appropriate header files, including AIXwindows include files.
3. Initialize the X-Windows Toolkit.
4. Add all top-level windows required for the client application interface.

Procedure

The fourth task involved in creating a client application interface is to set up an argument list for each widget and gadget in the interface. To accomplish this task, complete the following steps:

1. Determine which of the following variations of the X-Windows **XtSetArg** macro is most appropriate for each widget and gadget in the client application interface:

   Variation 1

   The first line of this variation declares an array named *args* whose size can range up to any predetermined maximum (ten elements in this example). Arrays of this type can be of unlimited size as long as the number of elements allocated is greater than or equal to the number of elements actually used. The first parameter (subscript 0) specifies the label for the created widget. (In this instance the label is actually a pointer to a compound string created by a call to a subroutine earlier in the code.) The second and third parameters (subscripts 1 and 2) specify that the widget has a width of 250 pixels and a height of 150 pixels. The following syntax statement defines variation 1:

```
Arg args[10];
XtSetArg(args[0], XmNlabelString, btn_text);
XtSetArg(args[1], XmNwidth, 250);
XtSetArg(args[2], XmNheight, 150);
        •
        •
        •
```

Variation 2

In this variation, a variable *n* is declared and used as a counter to to determine the size of the array. The use of a counter-variable rather than a hard-coded constant makes it easier to add and delete parameter assignments. The variable contains the total number of resources that have been set. It can be passed to the **XmCreate_** subroutines elsewhere in the source code as the parameter list count. The following syntax statement defines variation 2:

```
Cardinal n=0;
Arg args[10];
XtSetArg(args[0], XmNlabelString, btn_text); n++
XtSetArg(args[n], XmNwidth, 250); n++
XtSetArg(args[n], XmNheight, 150); n++
        •
        •
        •
```

Note: Do not increment the counter from inside the call to the **XtSetArg** macro. As currently implemented, the **XtSetArg** macro dereferences the first argument twice. A counter incremented from inside the call is incremented twice for each call.

2. Include an appropriate variation of the **XtSetArg** macro in every section of the client application source code in which a widget or gadget is created, as demonstrated in the *Create XmWidgetName* sections of the sample source code.

Note: There are three advanced programming alternatives to using the **XtSetArg** macro to set argument values for widgets and gadgets. The first of these is to assign each element of the type **Arg** individually as follows, being sure to keep name-value pairs synchronized:

```
XmString btn_text;
Arg args [10];
btn_text = XmStringCreateLtoR (Push Here,
                                   XmSTRING_DEFAULT_CHARSET);
args[0].name = XmNwidth;
args[0].value = (XtArgVal) 250;
args[1].name = XmNheight;
args[1].value = (XtArgVal) 150;
args[2].name = XmNlabelString;
args[2].value = (XtArgVal) btn_text;
```

The second alternative to using the **SetArg** macro is to initialize argument lists statically at compile time as follows:

```
static Arg Args [] = {
{XmNwidth, (XtArgVal) 250},
{XmNheight, (XtArgVal) 150},
};
```

The values of each argument are cast to the **XtArgVal** variable type. When the widget creation subroutine is invoked, the **XtNumber(args)** macro returns the number of elements in the argument list. You can create any number of lists as long as you declare each list as an **Arg** variable type.

The final alternative for creating argument lists involves initializing a list statically at compile time (as previously shown), and then modifying the values of the settings using regular assignment statements as follows:

```
XmString btn_text;
static Arg Args [] = {
{XmNwidth, (XtArgVal) 500},
{XmNheight, (XtArgVal) 150},
{XmNlabelString, (XtArgVal) NULL},
};btn_text = XmStringCreateLtoR ( Push Here ,
                                        XmSTRING_DEFAULT_CHARSET);
        args[1].value = (XtArgVal) 250;
        args[2].value = (XtArgVal) btn_string;
```

Task 5: How to Add Callback Routines

Prerequisite Tasks or Conditions

1. Ensure that the include files and other header files that you intend to list in your C language source code are available and accessible.
2. Identify and include all appropriate header files, including AIXwindows include files.
3. Initialize the X-Windows Toolkit Xt Intrinisics.
4. Add all top-level windows required for the client application interface.
5. Set up parameter lists for all widgets and gadgets used in the client application interface.

Procedure

The fifth task involved in creating a client application interface is to create specific callback procedures and add callback lists for interactive widgets and gadgets. To accomplish this task, complete the following steps:

1. Identify each widget and gadget that can be involved in an interactive event such as a mouse button press, a keyboard selection, or a cursor movement.
2. In the client application source code include the callback procedure to be invoked in response to each of these interactive events. Use the following syntax statement for each callback procedure:

   void CallbackProc (*w*, *client_data*, *call_data***)**
   Widget *w*;
   caddr_t *client_data*;
   caddr_t *call_data*;

   where the parameters have the following values:

w	Specifies the widget-gadget for which this callback routine is invoked.
client_data	Specifies the data that the widget-gadget passes back to the client when the widget-gadget invokes the callback procedure. Provides a means by which a client can define client-specific data to be passed to it, such as a pointer to additional information about the widget-gadget, a reason for invoking the callback, and so on.

   Note: The *client_data* parameter is set to **NULL** when all necessary callback information already exists in the widget-gadget data structure.

call_data	Specifies callback-specific data that the widget-gadget passes to the client.

Note: The *client_data* parameter is also widget-specific. It is set to **NULL** unless otherwise stated in the related widget-gadget reference material.

3. Use the X-Windows **XtAddCallback** subroutine to create a callback routine for the first user action. Use the following general syntax for this callback routine:

   void XtAddCallback(*w*, *callback_name*, *callback*, *client_data***)**
   Widget *w*;
   String *callback_name*;
   XtCallbackProc *callback*;
   caddr_t *client_data*;

   where the parameters have the following values:

w	Specifies the widget-gadget data structure to which the callback routine should be added.
callback_name	Specifies the callback list (within the widget-gadget) to which the callback routine should be appended.
callback	Specifies the callback procedure to add.
client_data	Specifies the client data to be passed to the callback when it is invoked by theX-Windows **XtCallCallbacks** subroutine. The *client_data* parameter is often set to **NULL**.

4. Continue building one or more lists of callback routines by using the X-Windows **XtAddCallback** subroutine to add individual callback routines for each previously-identified user action. Keep in mind that you might want a single user event to trigger multiple callback routines.

   Note: You can add *lists* of callback routines to your source code by using the X-Windows **XtAddCallbacks** subroutine in place of the **XtAddCallback** subroutine.

5. Determine which (if any) widgets and gadgets in the client application interface define one or more callback resources. This information is included in the reference material associated with each widget-gadget class.
6. Set the value of each applicable resource to the name of the callback list you created in step 2.

Task 6: How to Create Widgets and Gadgets With the AIXwindows Toolkit

Prerequisite Tasks or Conditions

1. Ensure that the include files and other header files that you intend to list in your C language source code are available and accessible.
2. Identify and include all appropriate header files, including AIXwindows include files.
3. Initialize the X-Windows Toolkit Xt Intrinisics.
4. Add all top-level windows required for the client application interface.
5. Set up parameter lists for all widgets and gadgets used in the client application interface.
6. Add appropriate callback routines for each widget class and gadget class, and develop all necessary callback lists.

Procedure

The sixth task involved in creating a client application interface is to create individual instances of all widgets, gadgets, and compound objects required by the client application interface. There are two ways you create widgets:

1. Use the X-Windows **XtCreateManagedWidget** subroutine to create a widget instance. Use the following general syntax for this widget creation routine.

   Widget XtCreateManagedWidget (*Name, WidgetClass, Parent, ArgumentList,*)
   String *Name*;
   WidgetClass *WidgetClass*;
   Widget *Parent*;
   Arglist *ArgumentList*;
   Cardinal *ArgumentCount*;

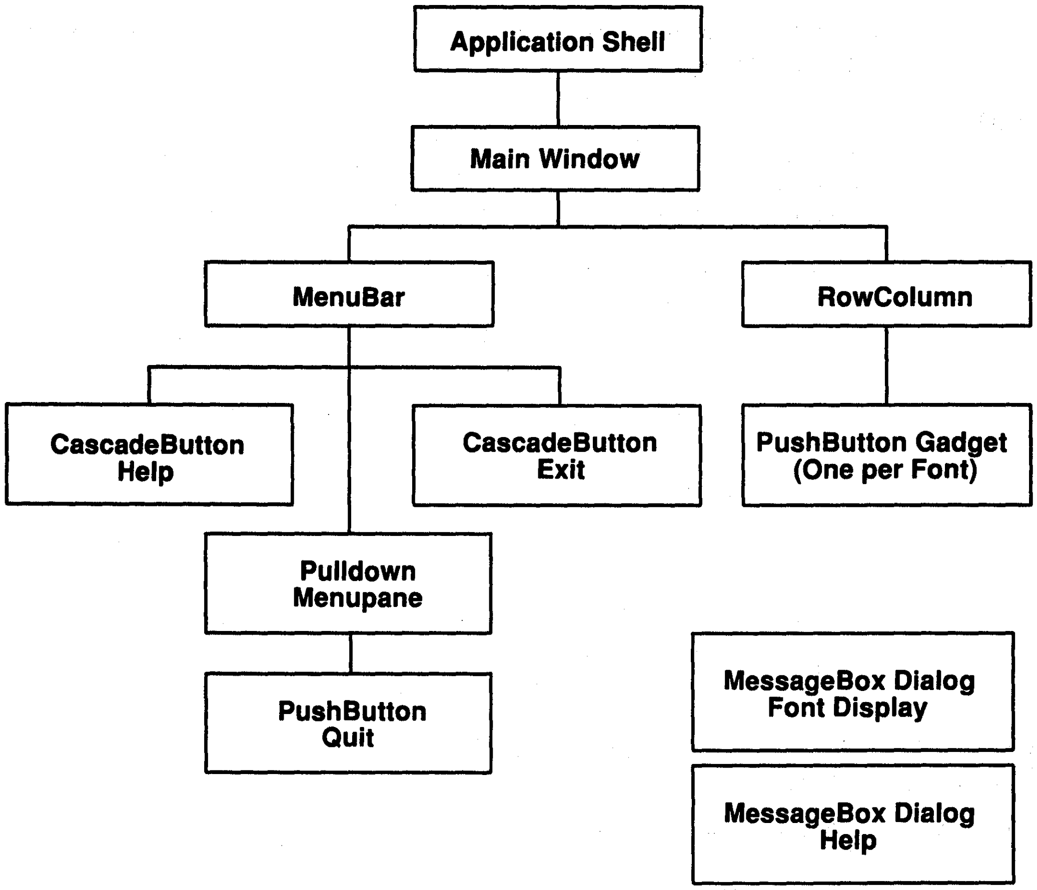
   where the parameters have the following values:

ArgumentCount	Specifes the number of resource/value pair arguments found in the argument list.
ArgumentList	Specifies the argument list used to override the resource defaults.
Name	Specifies the resource name for the created widget. The name is used for retrieving resources and if unique values are necessary, should not be named the same as a widget that is a child of the same parent widget. The only exception to this rule is a child that use identical resources.
Parent	Specifies the parent widget ID for the widget to be created.
WidgetClass	Specifies the widget class pointer for the created widget.

   This subroutine names the newly created widget specified by the *Name* parameter and defines it it be a widget of the class specified by the *WidgetClass* parameter. The class name or the widget name can be used in default files to refer to this widget. The widget's parent is the toplevel shell widget returned by the **XtInitialize** subroutine. The argument list and number of arguments specified by the *ArgumentList* and *ArgumentCount* parameters complete the call. This subroutine creates the widget and noitifies its parent, so that the parent can control its specific layout.

   The second way to create a widget does not automatically manage the widget. Instead, the widget is managed when the widget is displayed.

2. Each widget has its own **XmCreate** subroutine associated with it. A **XmCreate** subroutine creates the widget it is associated with but does not manage the widget. The new widget is managed using the **XtManageChild** subroutine.



Example of AIX Toolkit Parent-Child Widget Tree

Task 7: How to Realize AIXwindows Widgets and Gadgets

Prerequisite Tasks or Conditions

1. Ensure that the include files and other header files that you intend to list in your C language source code are available and accessible.
2. Identify and include all appropriate header files, including AIXwindows include files.
3. Initialize the X-Windows Toolkit Xt Intrinsics.
4. Add all top-level windows required for the client application interface.
5. Set up parameter lists for all widgets and gadgets used in the client application interface.

6. Add appropriate callback routines for each widget class and gadget class, and develop necessary callback lists.
7. Create an appropriate for the client application interface.
8. Create all necessary instances of widgets, gadgets, and compound objects as shown in the widget tree.

Procedure

The seventh task involved in creating a client application interface is to make the top-level widget in the AIXwindows interface widget tree (as well as all children managed by that widget) visible. To accomplish this task, complete the following steps:

1. Add an X-Windows **XtRealizeWidget** subroutine to the client application source code. At run time, this subroutine realizes (makes visible on screen) the top-level widget in the widget tree, as well as all children managed by that widget. The syntax for this subroutine is:

   void XtRealizeWidget (*Widget***)**
   Widget *Widget*;

   where the parameter have the following value:

Widget	Specifies the widget

2. Add an X-Windows **XtMainLoop** subroutine to the client application source code. This subroutine causes the client application to enter a loop, awaiting action by the user. The application passes control to the Xt Intrinsics and the AIXwindows widgets once the **XtMainLoop** subroutine is called.

Task 8: How to Link Libraries when Creating an Interface with the AIXwindows Toolkit

Prerequisite Tasks or Conditions

1. Ensure that the include files and other header files that you intend to list in your C language source code are available and accessible.
2. Identify and include all appropriate header files, including AIXwindows include files.
3. Initialize the X-Windows Toolkit Xt Intrinsics.
4. Add all top-level windows required for the client application interface.
5. Set up parameter lists for all widgets and gadgets used in the client application interface.
6. Add appropriate callback routines for each widget and gadget, and develop necessary callback lists.
7. Create an appropriate widget tree for the client application interface.
8. Create all necessary instances of widgets, gadgets, and compound objects as shown in the widget tree.
9. Realize the top-level widget in the and add the X-Windows **XtMainLoop** subroutine to cause the client application to enter a loop, awaiting action by the user.

Procedure

The eighth task involved in creating a client application interface is to link all necessary libraries in the appropriate order. To accomplish this task, complete the following steps:

1. After all source code has been compiled successfully, list these three libraries directly after the appropriate **link** command and flag:

   libX11.a
   libXt.a
   libXm.a

   Note: The order in which you list these libraries is important. Be certain you list the libraries in the order shown.

2. If the client application source code contains one or more **XmFileSelectionBox** widgets, add the **libPW.a** library to the link list, placing it between **libXm.a** and **libXt.a** as shown in the following list:

   libX11.a
   libXt.a
   libPW.a
   libXm.a

   libX11.a
   libXt.a
   libPW.a
   libXm.a
   libIM.a

3. Once all appropriate libraries are listed in the proper order in the link list, they are ready to be linked in the normal manner.

Task 9: How to Provide Resource Defaults Files for Interfaces Created with the AIXwindows Toolkit

Prerequisite Tasks or Conditions

1. Ensure that the include files and other header files that you intend to list in your C language source code are available and accessible.
2. Identify and include all appropriate header files, including AIXwindows include files.
3. Initialize the X-Windows Toolkit Xt Intrinsics.
4. Add all top-level windows required for the client application interface.
5. Set up parameter lists for all widgets and gadgets used in the client application interface.
6. Add appropriate callback routines for each widget class and gadget class, and develop necessary callback lists.
7. Create an appropriate widget tree for the client application interface.
8. Create all necessary instances of widgets, gadgets, and compound objects as shown in the widget tree.
9. Realize the top-level widget in the widget tree and add the X-Windows **XtMainLoop** subroutine to cause the client application to enter a loop, awaiting action by the user.
10. After the source code has been successfully compiled, link the resulting object code with all appropriate libraries.

Procedure

The ninth task involved in creating a client application interface is to decide whether or not to add one or more resource defaults files to the client application interface. To accomplish this task, complete the following steps:

1. Determine whether or not you want to provide the user with resource defaults files that permit the user to customize the AIXwindows interface at run time. Consider the following factors when determining whether to specify an argument in a defaults file or in the program itself:

   - Using a resource defaults file provides additional flexibility. The user can override preset defaults to reflect personal preferences, and the client application resource defaults file can also be modified for system-wide customization.
   - Specifying resource defaults settings in the source code rather than a resource defaults file gives the programmer greater control over the interface. These resource defaults cannot be overridden unless the source code is made available.
   - Using resource defaults files can speed application development. Changes in resource defaults can be accomplished with any ASCII text editor, without the need for recompilation of source code or relinking of libraries and files.
   - Resource defaults files can simplify the client application source code because resources in defaults files are specified as strings. The same resources listed in your source code might have to be in some internal format that requires several subroutines to compute.
   - Specifying resource defaults in the source code often results in more efficient performance of the interface due to reduced processing overhead.

2. You can use two files for customization of the resource defaults:

/usr/lib/X11/app-defaults/*File*	This file supplies resource defaults for an entire *class* of AIXwindows client applications executing anywhere in the computer system.

   Application programs specify the file that contains the application defaults when they call the **XtInitialize** subroutine.The *ApplicationClass* parameter of the **XtInitialize** subroutine specifies the name of the application defaults file. Several applications can point to the same file.

.Xdefaults	This file, placed in an individual user's home directory, supplies resource default values for all applications started by that user.

   The user defaults override application and system defaults and allow different users running the same program to specify their personal display preferences, such as color and font selection.

   Note: At run time, AIXwindows checks first for system resource defaults compiled into the widgets, then overrides them with the resource defaults in the client application defaults file **/usr/lib/X11/app-defaults/***File* (provided it exists). AIXwindows then looks for the **.Xdefaults** resource defaults file in the user's home directory and (assuming it exists) resets the resource default values as necessary. These resource defaults remain operative unless a different resource defaults has been hard-coded into the client application source code, in which case, the client application resource defaults overrides all resource defaults files.

Defaults File Interaction Example

The following example illustrates the interaction of the defaults files with each other and with parameters specified in programs.

To determine the color of the background, the Xt Intrinisics will do the following:

1. Look for the system defaults and initialize the background color to **white**. (These defaults are compiled into the widgets.)
2. Look for the *application* defaults file **/usr/lib/X11/app–defaults/***File* and set the background color to the color specified by the file, for example, **black**.
3. Look for the *user* defaults file **.Xdefaults** and set the background color to the color specified by this file, for example, **blue**.
4. If the program sets the background argument, **XmNbackground,** this overrides any defaults that may have been set.

Related Information

Chapter 1. AIXwindows Desktop contains information about the purpose of resource defaults and the format and contents of defaults files such as **/usr/lib/X11/app-defaults/***application_name* and **.Xdefaults**.

The User Interface Volumns(3 and 4) of AIX Calls and Subroutines for IBM RISC System/6000 contain information about the following:

- The **XmMainWindow** widget class, **XmMainWindow** widget class, **XmMessageBox** widget class, **ApplicationShell** widget class.
- The **XtArgVal** variable type and the **XtNumber(args)** macro, as well as information about using the **XtSetArg** subroutine as a macro.
- The X-Windows **XtInitialize** subroutine, **XtToolkitInitialize** subroutine, **XtAddCallback** subroutine, **XtAddCallbacks** subroutine, **XtGetValues** subroutine, **XtDestroyWidget** subroutine, **XtManageChild** subroutine, **XtManageChildren** subroutine, **XtUnmanageChild** subroutine, **XtUnmanageChildren** subroutine, **XtCreateManagedWidget** subroutine, **XtMainLoop** subroutine, **XtCreateWidget** subroutine.

Sample Source Code for AIXwindows Interfaces

The **makefile** at the end of the two sample AIXwindows programs must be used to compile these sample programs.

/* AIXwindows Interface Example #1: option.c */

```
/* list all standard include files */
#include <stdio.h>
#include <sys/signal.h>
#include <Xm/XmP.h>
#include <X11/Shell.h>
/* list an include file for each widget and gadget you use */
#include <Xm/BulletinB.h>
#include <Xm/CascadeB.h>
#include <Xm/CascadeBG.h>
#include <Xm/FileSB.h>
#include <Xm/Frame.h>
#include <Xm/Label.h>
#include <Xm/LabelG.h>
#include <Xm/List.h>
#include <Xm/MainW.h>
```

```
#include <Xm/MenuShell.h>
#include <Xm/MessageB.h>
#include <Xm/PanedW.h>
#include <Xm/PushB.h>
#include <Xm/PushBG.h>
#include <Xm/RowColumn.h>
#include <Xm/SelectioB.h>
#include <Xm/Text.h>

#define NUM 100

/* Forward Declarations - beginning with all widgets to be used */

Widget  shell,
        panel,
        btn1,
        btn2,
        btn3,
        label,
        button,
        optionMenu;

static XtCallbackProc hello1CB();
static XtCallbackProc hello2CB();
static XtCallbackProc hello3CB();

static void     die();

int             n;
Arg             args[NUM];
Display         *display;
XmString        string;

/* start main loop - main logic for AIXwindows interface*/

void  main(argc, argv)
    int     argc;
    char  **argv;
{
/* locals */

  Boolean       trace = False;

/* look for activity */

  signal(SIGINT, die);

/* initialize the Enhanced X-Windows toolkit and open display */

    XtToolkitInitialize();

    display = XtOpenDisplay(NULL, NULL, argv[0], "window",
                            NULL, 0, &argc, argv);
    if (display == NULL) {
        fprintf(stderr, "%s:  Can't open display", argv[0]);
        exit(1);
    }

/* Create Application Shell */

  n = 0;
   XtSetArg( args[n], XmNwidth, 400 ); n++;
   XtSetArg( args[n], XmNheight, 400 ); n++;
   XtSetArg( args[n], XmNallowShellResize, True ); n++;
   shell = XtAppCreateShell( argv[0], NULL, applicationShellWidgetCl
ass,
                                    display, args, n);
    XtRealizeWidget( shell );

    n = 0;
    panel = XmCreatePulldownMenu( shell, "panel", args, n );

    n = 0;
    btn1 = XmCreatePushButtonGadget( panel, "btn1", args, n );
    XtManageChild( btn1 );

    XtAddCallback( btn1, XmNactivateCallback, hello1CB, NULL );

    n = 0;
    btn2 = XmCreatePushButtonGadget( panel, "btn2", args, n );
    XtManageChild( btn2 );

    XtAddCallback( btn2, XmNactivateCallback, hello2CB, NULL );

    n = 0;
    btn3 = XmCreatePushButtonGadget( panel, "btn3", args, n );
    XtManageChild( btn3 );

    XtAddCallback( btn3, XmNactivateCallback, hello3CB, NULL );

    string = XmStringCreate( "options:", XmSTRING_DEFAULT_CHARSET );

    n = 0;
    XtSetArg( args[n], XmNlabelString, string ); n++;
    XtSetArg( args[n], XmNmnemonic, 'm' ); n++;
    XtSetArg( args[n], XmNsubMenuId, panel ); n++;
    XtSetArg( args[n], XmNorientation, XmVERTICAL ); n++;
    XtSetArg( args[n], XmNpacking, XmPACK_COLUMN ); n++;
    XtSetArg( args[n], XmNentryBorder, 10 ); n++;
    XtSetArg( args[n], XmNwhichButton, 1 ); n++;
    optionMenu = XmCreateOptionMenu( shell, "option", args, n );
    XtManageChild( optionMenu );

    label = XmOptionLabelGadget( optionMenu );
    button = XmOptionButtonGadget( optionMenu );

    printf( "ctrl-c to exit \n" );

    XtMainLoop();
}
static void die()
{
     printf("bye ... \n");
     exit(0);
}
static XtCallbackProc hello1CB(w, client_data, call_data)
    Widget   w;
    caddr_t  client_data;
    caddr_t  call_data;
{
        printf( "hello #1\n" );
        n = 0;
        XtSetArg( args[n], XmNwhichButton, 1 ); n++;
        XtSetValues(optionMenu, args, n);
}
static XtCallbackProc hello2CB(w, client_data, call_data)
    Widget   w;
    caddr_t  client_data;
    caddr_t  call_data;
{
        printf( "hello #2\n" );
        n = 0;
        XtSetArg( args[n], XmNwhichButton, 2 ); n++;
        XtSetValues(optionMenu, args, n);
}
static XtCallbackProc hello3CB(w, client_data, call_data)
    Widget   w;
    caddr_t  client_data;
    caddr_t  call_data;
{
        printf( "hello #3\n" );
        n = 0;
        XtSetArg( args[n], XmNwhichButton, 3 ); n++;
        XtSetValues(optionMenu, args, n);
}
```

/* AIXwindows Interface Example #2: popup.c */

```
/* list all standard include files */
#include <stdio.h>
#include <sys/signal.h>
#include <Xm/XmP.h>
#include <X11/Shell.h>
/* list an include file for each widget and gadget you use */
#include <stdio.h>
#include <sys/signal.h>
#include <Xm/XmP.h>
#include <X11/Shell.h>
#include <Xm/BulletinB.h>
#include <Xm/CascadeB.h>
#include <Xm/CascadeBG.h>
#include <Xm/FileSB.h>
#include <Xm/Form.h>
#include <Xm/Frame.h>
#include <Xm/Label.h>
#include <Xm/LabelG.h>
#include <Xm/List.h>
#include <Xm/MainW.h>
#include <Xm/MenuShell.h>
#include <Xm/MessageB.h>
#include <Xm/PanedW.h>
#include <Xm/PushB.h>
#include <Xm/PushBG.h>
#include <Xm/RowColumn.h>
#include <Xm/ScrollBar.h>
#include <Xm/ScrolledW.h>
#include <Xm/SelectioB.h>
#include <Xm/Text.h>

#define NUM 100

/* Forward Declaration - beginning with each widget used */

Widget  shell;
Widget  bb;
Widget  mbar;
Widget  text;
Widget  btn1;
Widget  btn2;
Widget  push1;
Widget  push2;
Widget  push3;
Widget  push4;
Widget  p1;
Widget  p2;

static XtCallbackProc   button1call();
static XtCallbackProc   button2call();
static XtCallbackProc   button3call();
static XtCallbackProc   button4call();

Display         *display;
static void     die();

void  main(argc, argv)
    int     argc;
    char  **argv;
{
/* locals */

   int  n;
    Arg           args[NUM];

/* look for activity */

    signal(SIGINT,  die);

/* initialize the toolkit */

   XtToolkitInitialize();
    display = XtOpenDisplay(NULL, NULL, argv[0], "window",
                            NULL, 0, &argc, argv);
    if (display == NULL) {
        fprintf(stderr, "%s:  Can't open display", argv[0]);
        exit(1);
    }

    n = 0;
    XtSetArg( args[n], XmNgeometry, "+600+800" ); n++;
    XtSetArg( args[n], XmNwidth, 300 ); n++;
    XtSetArg( args[n], XmNheight, 300 ); n++;
    XtSetArg( args[n], XmNallowShellResize, True ); n++;
    shell = XtAppCreateShell( argv[0], NULL,
                        applicationShellWidgetClass,
                        display, args, n);
    XtRealizeWidget( shell );

    n = 0;
    XtSetArg( args[n], XmNx, 50 ); n++;
    XtSetArg( args[n], XmNy, 50 ); n++;
    XtSetArg( args[n], XmNwidth, 50 ); n++;
    XtSetArg( args[n], XmNheight, 50 ); n++;
    bb = XmCreateRowColumn( shell, "rowcol", args, n );
    XtManageChild( bb );

/*
 ** create the menu pane
*/

   n = 0;
    mbar = XmCreateMenuBar( bb, "menu", args, n );
    XtManageChild( mbar );
/*
** create the pulldown menus
*/

   n = 0;
    p1 = XmCreatePulldownMenu( mbar, "popup", args, n );

  n = 0;
    p2 = XmCreatePulldownMenu( mbar, "popup", args, n );

/*
** create the cascade buttons
*/

   n = 0;
    XtSetArg( args[n], XmNsubMenuId, p1 ); n++;
    btn1 = XmCreateCascadeButton( mbar, "casc1", args, n );
    XtManageChild( btn1 );

  n = 0;
    XtSetArg( args[n], XmNsubMenuId, p2 ); n++;
    btn2 = XmCreateCascadeButton( mbar, "casc2", args, n );
    XtManageChild( btn2 );

/*
** create the pushbuttons for the pulldown menus
*/

  n = 0;
   push1 = XmCreatePushButtonGadget( p1, "button1", args, n );
   XtManageChild( push1 );

  XtAddCallback( push1, XmNactivateCallback, button1call, NULL );

  n = 0;
   push2 = XmCreatePushButtonGadget( p1, "button2", args, n );
   XtManageChild( push2 );

  XtAddCallback( push2, XmNactivateCallback, button2call, NULL );

  n = 0;
   push3 = XmCreatePushButtonGadget( p2, "button3", args, n );
   XtManageChild( push3 );

  XtAddCallback( push3, XmNactivateCallback, button3call, NULL );

  n = 0;
   push4 = XmCreatePushButtonGadget( p2, "button4", args, n );
   XtManageChild( push4 );

  XtAddCallback( push4, XmNactivateCallback, button4call, NULL );

  printf( "ctrl-c to exit \n" );

  XtMainLoop();
}
static void die()
{
   printf("bye ... \n");
   exit(0);
}
static XtCallbackProc button1call( widget, client_data, call_data )
Widget widget;
caddr_t client_data;
caddr_t call_data;
```

```
{
        printf( "hello, from button1\n" );
}
static XtCallbackProc button2call( widget, client_data, call_data )
Widget widget;
caddr_t client_data;
caddr_t call_data;
{
        printf( "hello, from button2\n" );
}
static XtCallbackProc button3call( widget, client_data, call_data )
Widget widget;
caddr_t client_data;
caddr_t call_data;
{
        printf( "hello, from button3\n" );
}
static XtCallbackProc button4call( widget, client_data, call_data )
Widget widget;
caddr_t client_data;
caddr_t call_data;
{
        printf( "hello, from button4\n" );
}
```

/* Makefile for Compiling Sample AIXwindows Programs */

```
CFLAGS=-DSTRINGS_ALIGNED
CC=/bin/cc
RM=/bin/rm -rf
DEPS=
LOCAL_LIBRARIES= -lPW -lXm -lXt -lX11 -lIM

.c.o:
        $(RM) $@
        $(CC) -c $(CFLAGS) $*.c
#####

all:: list

list: list.o
        $(RM) $@
        $(CC) -o $@ list.o $(LOCAL_LIBRARIES)

clean::
        $(RM) list
        $(RM) list.o

all:: option

option: option.o
        $(RM) $@
        $(CC) -o $@ option.o $(LOCAL_LIBRARIES)

clean::
        $(RM) option
        $(RM) option.o

all:: popup
```

```
popup: popup.o
        $(RM) $@
        $(CC) -o $@ popup.o $(LOCAL_LIBRARIES)
clean::
        $(RM) popup
        $(RM) popup.o
```

How to Create Menu Systems with the AIXwindows Toolkit

Prerequisite Tasks or Conditions

1. Ensure that the include files and other header files that you intend to list in your C language source code are available and accessible.
2. Identify and include all appropriate header files, including AIXwindows include files.
3. Initialize the Enhanced X-Windows Toolkit.
4. Add all top-level windows required for the client application interface.

Procedure

The AIXwindows Toolkit provides the following three types of menu systems for use in client application interfaces:

- *Popup* menu systems
- *Pulldown* menu systems
- *Option* menu systems

Each menu system is based upon an **XmRowColumn** widget configured to behave in a specific manner. To create a menu system for a client application interface, complete the following steps in sequential order:

1. Determine which of the three types of AIXwindows menu systems is best suited for your current purpose:
   - A *Popup menu system* usually consists of a single **XmRowColumn** widget (configured as a **Popup MenuPane** widget and containing a combination of **XmLabel** widgets, **XmPushButton** widgets or gadgets, **XmToggleButton** widgets or gadgets, and **XmSeparator** widgets or gadgets), plus its parent **MenuShell.** In addition, it can contain **XmCascadeButton** widgets or gadgets that access **Pulldown MenuPanes** functioning as submenus.
   - A *Pulldown menu system* typically consists of an **XmRowColumn** widget configured as a **MenuBar** widget , a set of **XmCascadeButton** widgets or gadgets that are children of the MenuBar, and an **XmRowColumn** widget configured as a **Pulldown MenuPane**. In addition, it can contain a combination of **XmLabel** widgets, **XmPushButton** widgets or gadgets, **XmToggleButton** widgets or gadgets, and **XmSeparator** widgets or gadgets.
   - An *Option menu system* typically consists of an **XmLabel** gadget describing the types of options available, an **XmRowColumn** widget (configured as a **Pulldown MenuPane** widget and containing **XmPushButton** widgets or gadgets that represent the available options), and a selection area consisting of an **XmCascadeButtonGadget** gadget that contains a label string reflecting the most recent option chosen.
2. Include the appropriate **XmCreate***Type***Menu** convenience creation subroutine in the client application source code. For example, the **XmCreatePopupMenu** convenience

creation subroutine creates an **XmRowColumn** widget (configured as a **Popup MenuPane** widget) and a parent **MenuShell**, and returns the widget ID for the **MenuPane** widget.

Note: You might find it useful to create a menu system without using the **XmCreate***Type***Menu** convenience creation subroutines. For example, if a client application requires access to individual **MenuShells**, the **MenuShells** and **MenuPanes** can be created with the Enhanced X-Windows **XtCreate** subroutines (or the creation subroutines for **MenuShells** and **XmRowColumn** widgets). If the **MenuShell** is created for a Popup menu system, the **MenuShell** must be the parent of the **Popup MenuPane**. If, on the other hand, the **MenuShell** is created for a **Pulldown** menu system utilizing a **MenuBar**, the **MenuShell** must be created as a child of the **MenuBar** widget.

3. If the menu system you create requires submenus, they can be created in the following manner:

   a. Create the main **MenuPane** widget as a child of a top-level widget.

   b. Create a **Pulldown MenuPane** widget and an **XmCascadeButtonGadget** gadget as children of the **Popup MenuPane**.

   c. Use the resource **XmNsubMenuId** to attach the **XmCascadeButtonGadget** gadget to the **MenuPane**.

   d. Repeat steps a through c as necessary to create additional submenus of the newly-created submenu, changing widget-gadget names for each submenu.

Chapter 4. AIXwindows Style Guide

AIXwindows Style Guide Overview

AIXwindows is a graphical user interface based on the Open Software Foundation's OSF/Motif user interface offering and is based on guidelines from the OS/2 Presentation Manager (PM) user environment. AIXwindows runs in the X–Windows environment. The components of the AIXwindows Environment are: AIXwindows and Enhanced X–Windows.

The AIXwindows runtime environment consists of the OSF/Motif window manager and a graphical OSF/Motif–based desktop that provides an iconic view of the file system.

The window manager works with the toolkit to manage the operation of windows on the screen. The window manager provides functions for moving and resizing windows, reducing windows to icons, restoring windows from icons, and arranging windows on the workspace. An additional AIXwindows window manager feature is the icon box. The icon box contains icons for all windows operating under the window manager.

The AIXwindows application development environment provides a high–level toolkit based on OSF/Motif and consists of the following tools: the OSF/Motif user interface toolkit bindings, a widget library that provides user interface objects using an X window, a gadget library providing windowless widgets, and the Enhanced X–Windows user interface toolkit bindings.

The AIXwindows toolkit is a collection of widgets and gadgets for building AIXwindows applications. The toolkit provides a standard graphical interface upon which the window manager is based. Toolkit widgets provide a 3–D reference appearance that gives users real–world, visual cues to the effects of their actions.

Enhanced X–windows is a network transparent windowing system designed to enhance the usability of the overall application processing environment.

User Control

A major software design goal should be user control of an application. User control gives the user the tools to get the job done and the ability to control those same tools. Users are in control of the application when the application is designed with these principles in mind:

- Consistency
- User–Object Interaction
- Flexibility
- User response to irreversable actions.

Because all applications differ in various ways, these principles may not always apply.

Consistency

Above all else, an application should be consistent. In addition to being consistent within its self, an application should be consistent with other applications that share the same environment.

Application consistency means the following:

- Similar controls operate alike or almost alike and have the same or similar uses. For example, since pulldown, popup, and cascading menus are similar controls, their operation and use should be similar.
- The same action should always have the same result. For example, click the **Select** mouse button on the **Window Menu** button of the window frame to display the window menu or click the mouse button twice (double click) to perform the default action.
- The location of the mouse pointer should be determined by direct manipulation and not positioned arbitrarily by the application.

An application should present its capabilities in an orderly manner. Necessary and commonly used functions are presented first and in a logical order. For example, essential functions could be included in a menu bar at the top of the client area where they are always visible and ready for selection.

More sophisticated or less frequently used functions can be hidden from immediate view. For example, dialog boxes provide a mechanism for hiding settings and functions that are infrequently used.

Decisions about the placement of functions are not easy to make. From the implementation standpoint, all functions are important. Often, however, a relatively small number of functions account for the majority of usage. These functions need to be prominently featured, but they can be prominent only if other functions are hidden.

Consistency among applications increases the user's sense of mastery. Experience with one application can be readily applied to another application, creating a positive transfer of knowledge. The focus of a computer session becomes the task at hand, not "learning a new application". When applications work in a manner that is consistent with other applications, users can enjoy a feeling of immediate confidence in their ability to master the new program. Also, they are pleasantly surprised when trying new functions because, although new, the functions seem familiar.

User–Object Interaction

User–object interaction describes the action transfer between a user and an object. This interaction connects a user's action to an observable response from an object. In this type of interface, the user experience the immediate visible result of an action.

The immediate visual response is important to the success of an application. Performance problems caused by inefficient program design and implementation make it difficult for users to concentrate on the task at hand and can render an otherwise well–designed application unusable.

User–object interaction simulates actual human actions where users use various tools to perform tasks on physical objects. Users control their AIXwindows environments by direct interaction with graphical controls similar to controls they use in real life. For example, buttons *push* to start an action and the slider on a scale actually *slides* to select a setting.

Another feature of user–object interaction is that the output of the application is also available as input. For example, a list of files is not only the result of a command, but also a collection of screen objects that a user can act upon.

User–object interaction gives the users control by enabling them to manipulate objects by *grabbing*, *pointing* or *selecting* the object, rather than by typing a command on a command line. Giving users control through user–object interaction also means reducing, wherever possible, the amount of information the user must memorize.

Flexibility

Flexibility should be apparent in both the operation and configuration of an application.

Providing multiple ways for users to access application functions and accomplish their tasks increases their sense of control. For example, a function could be accessed through a pull–down menu, a mnemonic key press, or a keyboard accelerator. Giving users control through flexibility enables them to select the best method for accessing a function based on criteria they choose,for example,experience level, personal preference, unique situation, or simply habit.

Allowing users to configure settings and select personal preferences enhances their sense of control and encourages them to take an active role in understanding the product and how it works. To be effective, the configurability of your application must be easily accessible.

User Response to Irreversable Actions

If a task has irreversible negative consequences, it should require users to make an explicit action before performing the task. For example, a worksheet could be saved by clicking the **Select** mouse button or typing the **Select** key on a **Save** push button. To delete the same worksheet should require clicking the **Select** mouse button or typing **Select** on an **Erase** push button and answering some type of a question with another selection action. For example, `"Are you sure you want to erase this worksheet?"`.

Warnings help protect users from inadvertent destructive operations, yet allow them to remain in control of the application. Operations that may cause a serious or unrecoverable loss of work should warn users of the consequences and request explicit confirmation.

By anticipating errors, you enable the support of recovery attempts and can provide messages informing users of the proper corrective action. Part of this support can be context–sensitive help and provisions for undoing an action. To avoid excessive errors, controls can be temporarily disabled when it would be inappropriate to use them. Disabled controls should provide a visual cue that they are not currently operable.

Context–sensitive help aids understanding, reduces errors, and eases recovery efforts. Help information text should be clear, concise, and written in everyday language. Help information should be readily accessible and just as readily removable.

Many users are most comfortable with learning how to use software applications when they use a natural, trial–and error method. An undo function supports learning by trial–and–error by minimizing the cost of errors. The undo function allows users to retract previous actions, and helps foster a spirit of exploration and experimentation.

The Elements of an AIXwindows Program Environment

At the highest level, the AIXwindows program environment is composed of the following four elements:

- An input selection model.
- A window manager.
- Other application programs.

The Input Selection Model

The AIXwindows environment is based on an object–action input selection model. The selection model defines the actions that users performs to control the window manager and applications in the AIXwindows environment.

The selection model follows a point–and–click paradigm. Users first point at and select an object with which to work, and then point at and select an action to perform on the selected object. The AIXwindows selection model is discussed in Understanding Consistent Behavior in an AIXwindows User Application.

Window Management

The window manager provides users with a way to manipulate the windows displayed in their AIXwindows environment. Typical manipulations include moving, resizing, enlarging, reducing, and arranging windows as needed.

The AIXwindows window manager, **mwm**, frames application windows with an eight–segment border that can be stretched to resize the window. A title area, supplied by the window manager, displays a title for the window and can be used to move the window. Graphical buttons embedded in the window manager frame provide a window management menu and other window controls. Additionally, the AIXwindows window manager has a three–dimensional appearance so that the control buttons, when pressed by the mouse pointer, actually look like they have been pressed. The window manager helps provide for consistent behavior from one application to the next and is discussed in Understanding the AIXwindows Window Management Environment.

Application Programs

Application programs fill the space inside the window frame. By following the selection model and the guidelines in this guide, any applications you create should be consistent with the behavior of other applications in the AIXwindows environment. Designing applications with consistent behavior and the proper use of controls are discussed in Understanding AIXwindows Client Area Design., Understanding AIXwindows Application Graphic Controls, Understanding AIXwindows Menus, and Understanding AIXwindows Dialog Boxes.

Understanding the AIXwindows Window Management Environment

The AIXwindows user interface provides an environment designed to facilitate communications between the user and your application. This environment is composed of discrete graphical elements.

The graphical elements of the AIXwindows user interface facilitate communication by providing the user with a metaphor, a figurative concept suggestive of real world objects. Through this metaphor, interaction with your application should be more familiar (thus more intuitive) and less technical than the traditional user interface provided by the command–line prompt. This metaphor is usually referred to as a *desk*, *desktop*, *workspace*, or *workbench*. While desktop is perhaps the more widely known term, this guide uses the term workspace to emphasize that applications in the AIXwindows environment need not be office oriented and that the functionality and graphical elements of the user interface are tools that allow users to accomplish tasks with their computers.

The elements of the user interface, the objects that the user sees (for example, windows, icons, menus, and dialog boxes), appear on the workspace and can be stacked on top of each other like papers on a desk or tools on a workbench.

The following major points about the graphic elements of the AIXwindows workspace are described:

- Types of windows
- Window anatomy
- The icon box.

Types of Windows

In the AIXwindows environment, the user communicates with your application using windows. A **window** is an area of the screen (usually rectangular) that provides the user with the functional means to communicate with your application and through which your application can communicate with the user.

A typical AIXwindows environment may have several applications in operation simultaneously. Each application typically has a main or primary window that displays data and in which the user has their primary interaction with the application. Additionally, applications usually have one or more secondary windows (dialog boxes) that carry on context–specific dialogues with the people using the application.

While your application can be made up of many windows, each window will be one of only two basic types:

- A primary window, or
- A secondary window.

Primary Windows

A primary window is the window by which all the other windows used by your application are generated. A primary window may be obscured by overlapping windows. Your application can have more than one primary windows.

A primary window is the only window by which an application can be closed. That is, when users close the last primary window of an interactive application, the application session should end. Closing a primary window causes all secondary windows associated with that window to go away.

When the user invokes your application, the application's first task is usually to display a primary window. Because of this, the user often thinks of the primary window as the main window. This provides the user with a valuable point of reference: they feel in control knowing there is a main window from which all else follows and to which they can return.

You should design your application to encourage this sense of control, and so that, as the user opens and closes windows in their dialog with your application, they can always return to a primary window. A primary window should remain consistent in appearance and behavior with the last time they were there.

Secondary Windows

Context–specific dialogs usually occur inside secondary windows called dialog boxes. When the dialog is completed, the secondary window usually disappears.

Secondary windows are always related to a parent window. Sometimes the parent is a primary window, sometimes another secondary window. A primary window can have any number of secondary windows as its children.

Secondary windows are not constrained to be inside the primary window, but they will always appear on top of that parent window in the window hierarchy. Think about how your application will distinguish between primary and secondary windows. One method is to include identifying information in the title area. For application–oriented programs, the application name is followed by the file name in the title bar; for object–oriented programs, the object name is followed by the function. Secondary windows then have title areas that include the application or object name and the implied action.

When a primary window is minimized, its secondary windows are temporarily removed from the display.

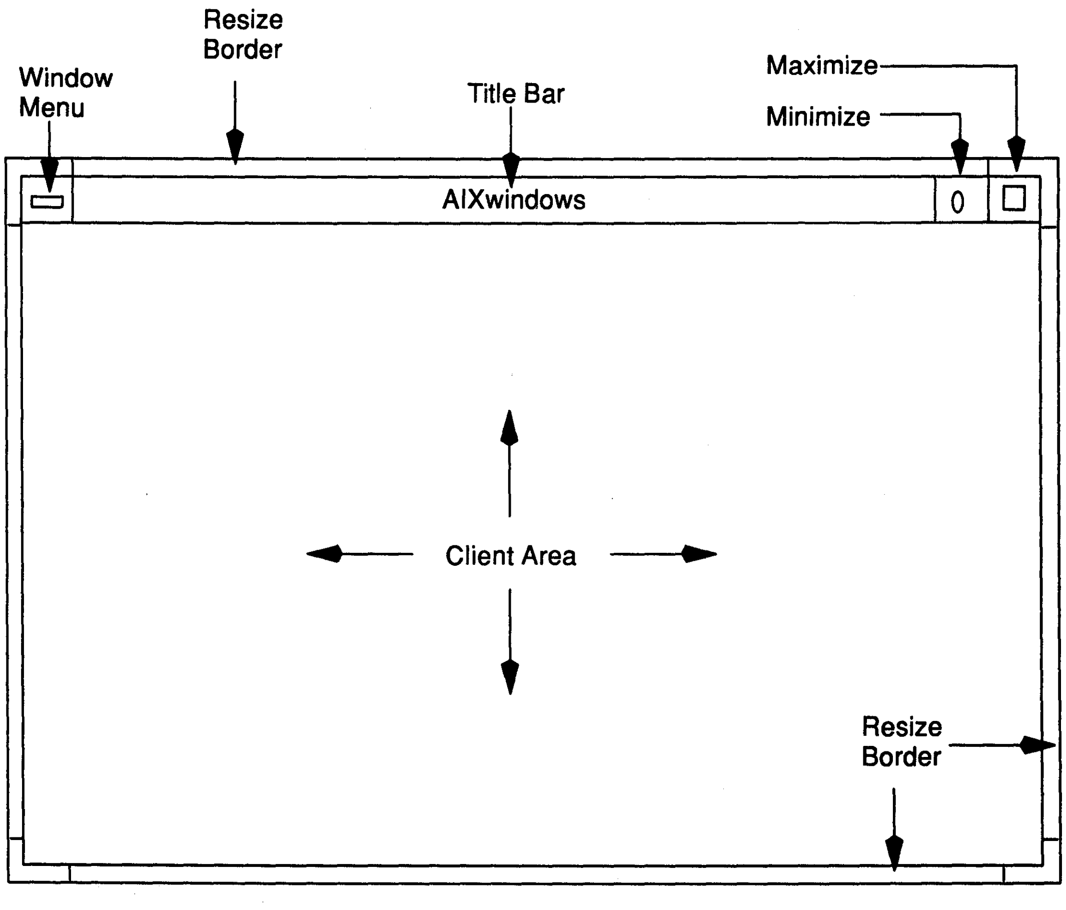
Window Anatomy

The AIXwindows window manager (**mwm**) provides windows with a window frame that contains various components. These components are functional, providing a convenient way for mouse users to invoke window management functions. Keyboard–only users use the window menu provided by **mwm** to invoke window management functions. Window frame components are sometimes called decorations.

Along with window frame components, the AIXwindows window layout includes a **client area**, the area inside the window frame, for your application to use.

In general, a window consists of the following components:

- Window menu button
- Window control buttons
- Title area
- Resize border
- Client area.

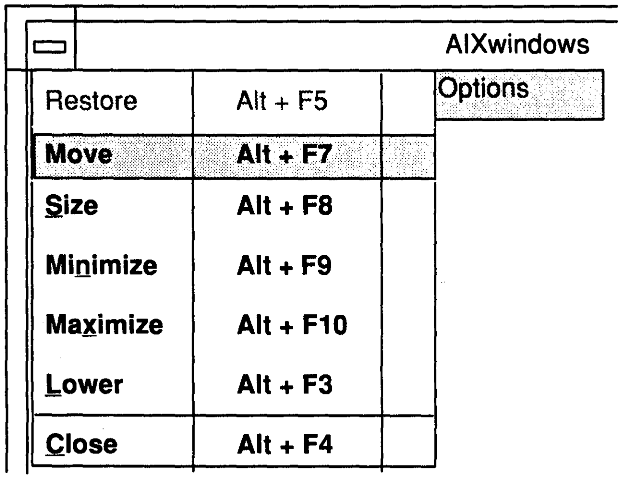


The components of the standard window layoutmay require some amount of modification for specific implementations. For example, it is inappropriate to resize some windows. These should have their resize borders removed.

Window Menu and Window Menu Button

The window menu button is located in the title bar, on the left side of the title area, and is used to display the window menu. Double–clicking the Select mouse button with the mouse pointer over the window menu button closes the window. The window menu provides a standard location for important window management functions. Users can browse the menu to see what actions are available. The window menu pulls down from the upper left corner of the window frame. Mouse users display the window menu by pressing the Select mouse button with the mouse pointer on the window menu button. Keyboard users display the window menu by pressing the Shift+Esc keys when the input focus is in the window.

The window menu and window menu button are sometimes called the *system menu* and *system menu button* respectively.



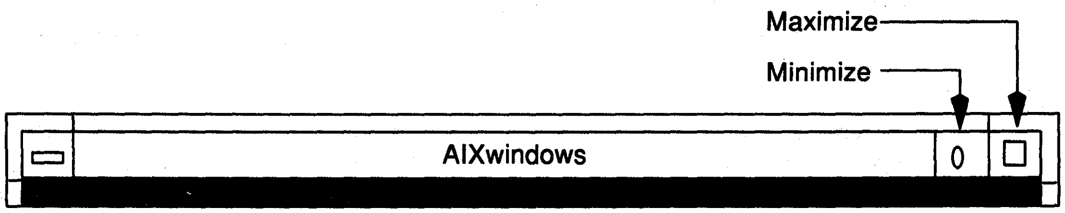
The standard selections of the window menu are shown in the figure. In the figure, the **Move** selection is being chosen. Also, the **Restore** function is not highlighted to provide a visual cue that, in the present context, the function is unavailable. The selections of the window menu have the following functions and accelerators:

Function	Accelerator	Action
Restore	Alt+F5	Restores a minimized or maximized window to its normal size. This selection is de–emphasized (grayed out) when the window is in its normal state.
Move	Alt+F7	Moves a window around on the workspace.
Size	Alt+F8	Stretches or shrinks a window in the direction indicated by the mouse pointer.
Minimize	Alt+F9	Changes a window into an icon.
Maximize	Alt+F10	Enlarges a window to its maximum specified size.
Lower	Alt+F3	(Optional) Moves a window to the back of the workspace (the bottom of the window stack).
Close	Alt+F4	Closes a window and removes it from the workspace.

Keyboard accelerators for the window menu are optional. If you decide to use accelerators, use the accelerators suggested above.

Window Control Buttons

Window control buttons are push buttons located in the upper right corner of the AIXwindows window frame. They provide a short–cut for invoking window management functions without pulling down the window menu. Users invoke window management functions by clicking on the appropriate window control button.



The functions chosen for window control buttons are implementation–dependent and include such functions as **Maximize** and **Minimize**.

Minimize Button

The minimize button is located to the immediate right of the title area. It provides the same function as the **Minimize** selection in the window menu. Users click the Select mouse button with the mouse pointer on the minimize button to shrink a window to an icon.

Maximize Button

The maximize button is located between the minimize button and the resize border. It provides the same function as the **Maximize** selection in the window menu. Users click the Select mouse button on the maximize button to enlarge a window to its maximum size. The maximize size of a window is established by the application. Clicking the Select mouse button with the mouse pointer on the maximize button of a maximized window restores the window to its original size, the same function as the **Restore** selection of the window menu.

Title Area

The **title area** is the horizontal bar that lies between the window menu button and the window control buttons, and, as part of the window frame, highlights when the window has the input focus. The title in the title area identifies the window.

Pressing the Select mouse button with the mouse pointer on the title area and dragging the mouse pointer on the screen will move the window to a new location. Clicking the Select mouse button with the mouse pointer on a title area (or frame) raises that window to the top of the window stack.

The window manager displays an **application title** in the title area. The application title is supplied by the application and clearly identifies the window and its role within your application.

You provide the window manager with a title. In object oriented environments, this title should be the name of the object followed by the application name; in file–based environments, the title could be the name of the application followed by the file name. Multiple windows of the same application should have titles that identify them with the application in some way, but can otherwise be distinct from one another.

The title area should not display the version number of your application. Nor should you use it to display system messages. Use the title area for information that stays relatively constant throughout the work session of your application.

Resize Border

Your application suggests the initial size of its windows to the window manager. Window sizes vary according to the work users perform in them. At any time, users should be able to alter the size of a primary window.

The window manager provides a functional window frame which surrounds the client area. The window frame highlights when input focus is passed to the window. Resizable windows have a wide frame border that users can drag when they want to change the window's size.

Client Area

The **client area** is the portion of the window in which users perform most application–level tasks. For example, if users are working with a graphics editor or a text editor, the client area contains the figure or document being edited. The client area is inside the window frame and can be composed of multiple work areas.

The Icon Box

By default, minimized windows (icons) are placed on the workspace in a row beginning at the lower left corner of the screen. However, under the management of the window manager, users can choose to group minimized windows in an icon box. They can also choose to have minimized windows placed at the location the normal window occupied.

An icon box acts like a typical window in the sense that it has a window frame and frame controls. Like other windows it can be sized, moved, minimized, maximized, restored, and lowered. However, an icon box cannot be closed.

Understanding Consistent Behavior in an AIXwindows User Application

The AIXwindows environment provides users with a behaviorally consistent graphical user interface based on a number of discrete but related operational models. Using these models, much of the interaction with an application program can be reduced to the following scenario: users select an object using the input device with which they feel most comfortable; they then select an action to perform on the selected object.

Consistent behavior enables the user to perform a task by focusing on the task itself rather than on the tools or the method in which they perform the task. Just as an automobile is a tool for transportation that does not require the average person to be a mechanic, a user–oriented software application provides users with a productivity–enhancing tool without requiring them to become engineers.

An understanding and consistent use of the following four operational models will help enable consistent behavior of your applications.

- The input focus model.
- The input device model.
- The navigation model
- The object–action selection model.

The Input Focus Model

A typical AIXwindows workspace can contain many windows. Like sheets of paper stacked on a desk, windows can be stacked on the workspace. Some windows may be partially or completely obscured by other windows. At any given time, however, only one window has the **input focus**. The window with the input focus is known as the **active window** and is the window where keyboard input will appear. The active window is also the only window that has the location cursor. Input focus can be moved from window to window.

When a window receives the input focus, the default behavior moves the window automatically to the front of the workspace so that no part of the window is obscured. This behavior can be modified to suit the needs or preferences of the users of your application. Additionally, the window with the input focus has a highlighted frame, supplied by the **mwm** window manager. This highlighted frame provides a visual cue that the window is active and further distinguishes the active window from inactive windows in the workspace.

The input focus model can use either an explicit or an implicit policy for moving the input focus from one window to another. Explicit focus is the default.

Explicit Focus

In explicit focus (the default). users explicitly select which window receives the input focus. With a mouse or other pointing device, users move the mouse pointer into a window and press the Select mouse button to move the input focus to that window. With a keyboard, users press the Alt+Esc keys to move the input focus sequentially through the windows stacked in the workspace. Explicit focus is sometimes known as click–to–type.

Implicit Focus

In implicit focus, the input focus moves to the window into which users move the mouse pointer. No explicit selection action is performed. The focus policy that a user chooses to implement is a matter of personal preference and need. Some mouse users find implicit focus more convenient. When using only keyboards, there is no distinction between explicit and implicit focus. Implicit focus is sometimes referred to as pointer, track pointer, track listener, or as being real estate driven.

The Input Device Model

The AIXwindows user interface enables users to communicate with your application using a keyboard and, optionally, a mouse or other pointing device.

Dual Access via Mouse and Keyboard

Some users prefer controlling software programs by pointing and clicking a mouse. Other users dislike removing one hand from the keyboard to “catch” a mouse and prefer instead to control programs solely from the keyboard. Other users lack the room for a pointing device on their desk. Still others prefer to mix mouse usage with keyboard usage: for frequently performed functions, they use keyboard accelerators; for other functions, they point and click with the mouse.

In AIXwindows applications, the keyboard is virtually interchangeable with the mouse. This enables users to pick the appropriate tool for the job or to choose the tool with which they feel the most comfortable.

You should design your application so that users can control it using a pointing device, the keyboard, or both. Although you may decide to make the pointing device the primary means of control, you should not constrain users from using the keyboard for application control.

Designing your application for dual accessibility gives users control by enabling them to choose the input device they find best suited to their needs, given their particular work situation and personal preferences.

Keyboard Conventions

The keyboard conventions outlined in this section work with ANSI keyboards. Keyboards, however, can differ greatly among manufacturers, and even among models made by the same manufacturer. Therefore, exact key labels as described here may differ from those that you are used to seeing. For example, the Alt key is sometimes labeled Meta, Extend or Extend char.

In general, all keyboards have the following types of keys:

Alphabetic keys
Keys representing the letters of the alphabet, the marks of punctuation, and text formatting functions such as the Tab, Return and spacebar keys.

Numeric keys Keys that represent the numbers from 0 to 9. These are included near the top of the keyboard or, in a numeric keypad on one side.

Navigation keys
Keys that are pressed to move the insertion cursor. These keys are commonly called the arrow keys and include the up arrow, the down arrow, the right arrow and the left arrow. On some keyboards they are separate keys, on others they are included as part of the numeric keypad. Also included in this category are the Home, End, PageUp, and PageDown keys.

Modifier keys Keys that are used in conjunction with other keys to modify the meaning or effect of those keys. The modifier keys include the Ctrl, Shift, and Alt keys. If your application designates a particular key as a modifier key, that key should not have any other function associated with it.

Special–purpose keys
Keys that have particular functions and frequently have labels stating their purpose. Among these keys are Help, Menu, Esc, Select, Enter, Delete, Backspace, and Insert.

Function keys Keys that most keyboards provide for extra or general functions. Usually these keys are labeled F1, F2, and the like. Some keyboards have ten function keys, others twelve or more. The AIXwindows environment assumes a keyboard with at least ten function keys. Function keys are usually placed either across the top of the keyboard or on one side, often the left.

As noted, keyboards differ greatly and some may not have all of the keys just mentioned. The table lists some common substitutions.

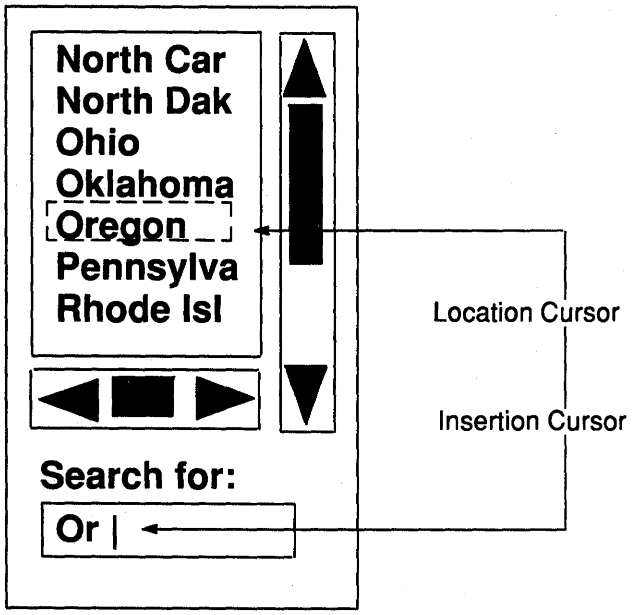
Key	Description	Substitution
Select	Makes a selection from the keyboard	Spacebar or Enter
Menu	Invokes a pop up menu	F4
Help	Invokes the help function	F1
Alt	Modifies the meaning of another key	Extend
Esc	Cancels current action	F12
Enter	Invokes the default action	Return
Next Field	Moves location cursor to next field	Tab or Ctrl+Tab
Previous Field	Moves location cursor to previous field	Shift+Tab or Ctrl+Shift+Tab

Also, some keyboards may have duplicate sets of keys. Numeric keys, for example, are often found both in the top row of alphabetic keys and in a numeric keypad and function keys are sometimes found both at the top of the keyboard and as a separate keypad on one side. These duplicate keys may have different keycodes associated with them. In such a case, you can design your application to provide some special functionality on the extra keys. For example, you might retain the top row of alphabetic keys as numeric keys and use the numeric keypad for special functions.

Cursor Shapes

While the traditional use of a keyboard has been for text entry, the keyboard has also been adapted for use as a pointing device. Because of this, the keyboard has two cursors, one for location, the other for text insertion.

The **location cursor** reflects movement on the screen using the keyboard's cursor navigation keys and indicates the current location of the keyboard input focus. The position of the location cursor gives users a visual cue to which object they are about to select. The location cursor is displayed whether or not a mouse is attached and is in addition to the mouse pointer. For example, the location cursor can indicate which item in a list box a user wants to select. The location cursor is often shown as a rectangle around a control. The location cursor is sometimes called the selection cursor.



The **insertion cursor** is used in edit controls such a text entry box. It indicates the point in text or graphics where new characters will be inserted. In text entry, the insertion cursor's default shape is a pipe (|) or bar for inserting, and a block character cell or underscore (_) for overstriking. In graphics entry, the insertion cursor's shape might be a pencil, paintbrush, or some other graphically descriptive image.

	Insertion	Overstrike
Active	|	█ or —
Inactive	| or ^	▒ or —

The insertion cursor is only displayed where graphics or text entry is allowed. The insertion cursor should change size to match the size of the current font.

Pointing Devices

Direct manipulation allows users to control an application by choosing selections from a menu and by setting controls following a point–and–click method. In the point–and–click method, users move the mouse until the mouse pointer is over (points to) the desired object. They then click the Select mouse button.

Direct manipulation usually requires a pointing device: a mouse, graphics tablet, track ball, joystick, or some other such device. Typically, keyboards with two sets of arrow keys use one set for mouse pointer navigation and the other set for location cursor navigation. Using the pointing device, users can navigate rapidly around the screen and can point at and choose objects to work on and actions to perform.

Throughout this guide we use the term mouse to refer to all pointing devices. You can use any pointing device in place of a mouse.

Mouse Buttons

You can use mouse buttons in combination with the mouse pointer to make selections, move the input focus, or position the insertion cursor.

This guide assumes a three–button mouse and gives the following names and functions to mouse buttons.

Button	Name	Description
1	Select	Used for selection.
2	Menu	Used for direct manipulation of objects.
3	Custom	Used to display pop up menus and otherwise as needed for application–specific functionality.

Note: The position of mouse buttons can vary depending on whether the mouse is configured for left– or right–handed operation.

You can perform the following mouse button operations:

Pressing	Holding down a mouse button.
Releasing	Releasing a mouse button after it has been pressed.
Clicking	Quickly pressing and releasing a mouse button without moving the mouse.
Dragging	Moving the mouse while a mouse button is pressed.
Double–Clicking	Clicking a mouse button twice in rapid succession without moving the mouse pointer.

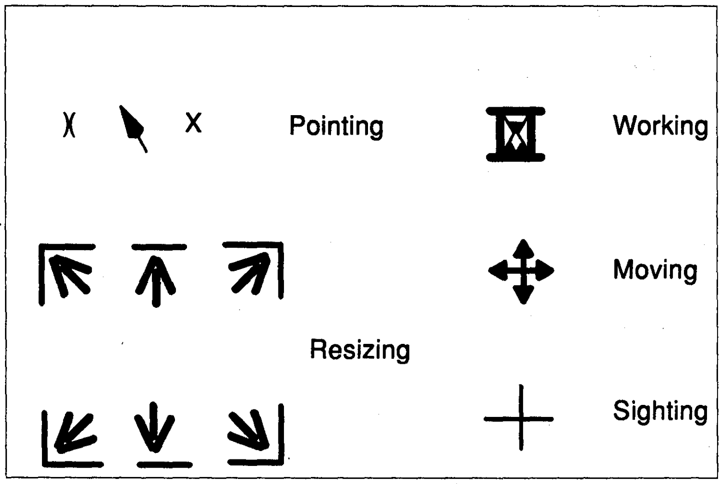
The Pointer

The mouse is associated with one and only one **mouse pointer**. The mouse pointer appears on the workspace and represents the location of the mouse. Movements of the mouse pointer correspond to movements of the mouse. The mouse pointer is not confined to any specific application. Users can move it anywhere on the workspace. The mouse pointer is sometimes known simply as the pointer.

Your application should only interpret the mouse pointer position; it should not attempt to change it. To do so would violate users' trust in the consistency of your program and their sense of control. Also, changing the mouse pointer location may create problems in applications that use absolute location devices (like graphics tablets).

Pointer Shapes

The shape of the mouse pointer provides users with an important visual cue indicating the functionality of the area in which the mouse pointer is currently located. The figure illustrates the standard mouse pointer shapes. While you should not create new mouse pointer shapes for functions that already have mouse pointer shapes associated with them, you can create new mouse pointer shapes for functions not already associated with a pointer shape. Also, do not use a predefined shape to symbolize a function it was not designed to represent.



If you decide to use other mouse pointer shapes in your applications, avoid shapes that are hard to see, hard to comprehend, create visual clutter, or flicker excessively when changing shape repeatedly.

Ensure that any mouse pointer you create has an obvious **hot spot** (active point). This is the area of the pointer image that marks the location of the mouse pointer. This is particularly critical for mouse pointer shapes that point to objects or specify positions. Users should be able to intuitively locate the hot spot (for example, the tip of an arrow or the center of a crosshair).

Design your application so that users can set the ratio of mouse pointer speed to mouse speed. This ratio is called the **gain**. Mouse pointer speed can be constant or accelerating. The gain is typically set globally in the AIXwindows environment. Therefore, if your application needs to adjust the gain, implement a zoom feature rather than change the gain. A zoom feature (similar to the zoom lens of a camera) adjusts the gain by varying the magnification of the application.

The Navigation Model

Regardless of whether they use a mouse, a keyboard, or both, users will need to move the mouse pointer and the location cursor to new positions. That is, they will need to navigate around the workspace.

The mouse pointer is a graphical representation of the current location of the mouse. The only way to move the mouse pointer is to physically move the mouse.

The location cursor shows the current location of the keyboard focus. A user can control the location cursor with either the mouse or the keyboard.

The Object–Action Selection Model

In **object–action selection**, users first select an object, and then select an action to perform on that object. Object–action is patterned after real life and provides users with a readily comprehensible operational model for AIXwindows applications.

It is helpful to note that the term **object** includes not only recognizable objects like windows and push buttons, but also objects such as the individual letters of a text file, that are perhaps less often thought of as discrete entities.

The AIXwindows selection model employs the following kinds of selection:

- The selection of a single object.
- The selection of a range of objects.
- The selection of additional (non–contiguous) objects, including multiple ranges.

To make selections in AIXwindows applications, users always use the same basic steps. First, they place the pointer (mouse) or location cursor (keyboard) on the object they wish to select. Second, they perform a specific selection action. Thus the kinds of selection listed here and explained below are not separate types of selection. Rather, they are variations of the one selection model theme. Which variation they use depends on whether they wish to select a single object, a range of objects, or several additional (non–contiguous) objects.

Some controls, as selection objects, make specific assumptions about the selection model. For example, a set of radio buttons assumes that each button's selection is mutually exclusive. Thus, while radio buttons follow the selection model for single objects, they do not allow any other type of selection. Check buttons, on the other hand, assume that each button's selection is not mutually exclusive. While they also follow the selection model for single objects, they allow the selection of multiple buttons without deselecting the prior selection (as is the case in strict single selection). Check buttons are an example of what is called multiple selection.

The following table lists the mouse button and keyboard operations of the AIXwindows selection model.

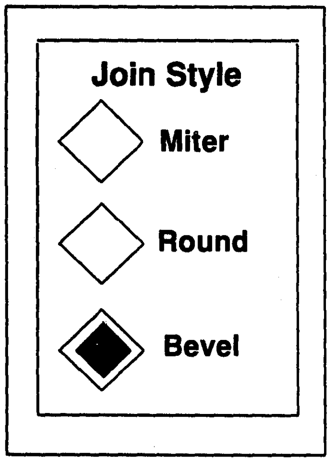
Selection Task	Mouse Button Operation	Keyboard Selection Operation
Select a single object (set the anchor point) and deselect all other objects.	Click Select	Press the Select key
Select a range of objects (set the anchor point at range beginning) and deselect all other objects.	Drag Select	Press the Shift+navigation keys
Toggle the selection of all objects between current location and the anchor point.	Shift+click Select	Press the Shift+Select keys
Toggle the selection of an additional object.	Ctrl+click selection operation	Press the Ctrl+selection operation keys

Single Selection Selects One Object Only

Single selection is probably the most common type of selection. In single selection, users select a single object upon which to perform an action.

In single selection, users move the mouse pointer or location cursor to a selectable object and then make their selection. Selecting an object changes the object's appearance. This provides users with the necessary visual cue to reinforce their sense of control over the selection process. For example, as shown in the figure, the insertion cursor in a text entry box is emphasized when the box is selected and de–emphasized when the box is not selected. In single selection, when one object is selected, other objects previously selected are deselected.

You can use single selection for such actions as selecting the active window (when the input focus policy is explicit selection) or selecting a push button or other type of control. In a text entry area, they use single selection to position the insertion cursor.



Single Selection with a Mouse

Users working with a mouse perform the following steps to select a single object:

1. Move the mouse pointer until it lies over the object they wish to select.
2. Click the Select mouse button to select the object.

Pressing the Select mouse button changes the object's appearance providing the visual cue to which object is about to be selected. Releasing the button selects the object and completes the single selection process.

Single Selection with a Keyboard

Keyboard users perform the following steps to select a single object:

1. Move the location cursor using the cursor navigation keys until it lies over the object they wish to select.
2. Press the Select key.

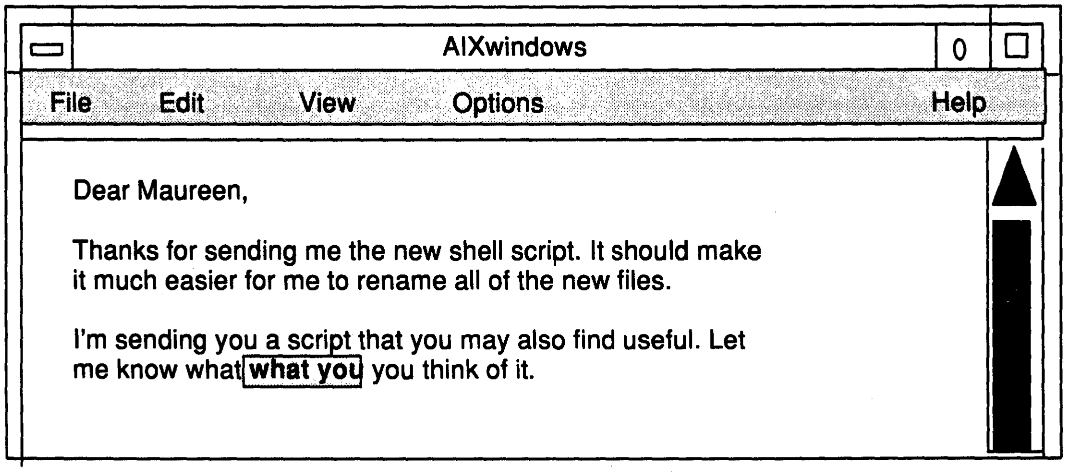
Pressing the Select key has the same effect as a mouse user's pressing the Select mouse button; it changes the object's appearance and provides the visual cue that the object is about to be selected. Releasing the Select key, selects the object and completes the single selection process.

Range Selection Selects Contiguous Objects

In **range selection**, users select a range of contiguous objects upon which to perform some action. The range is based on, but not limited to, a rectangular area. To be included in the range, objects must be completely included in the selection action.

In range selection, users move the mouse pointer or location cursor to the selectable graphics object and then make their selection. The object's appearance changes to provide a visual cue, just as it does in single selection.

In range selection, as the next object in the range is selected, the previous object in the range remains selected. Both objects become part of the range being selected. Any number of objects can be selected in range selection as long as they form a contiguous group.



Dear Maureen,

Thanks for sending me the new shell script. It should make it much easier for me to rename all of the new files.

I'm sending you a script that you may also find useful. Let me know what **what you** you think of it.

You can use range selection during cut–and–paste operations on text. As illustrated, words are selected as a range of ASCII characters and then acted upon (cut or pasted).

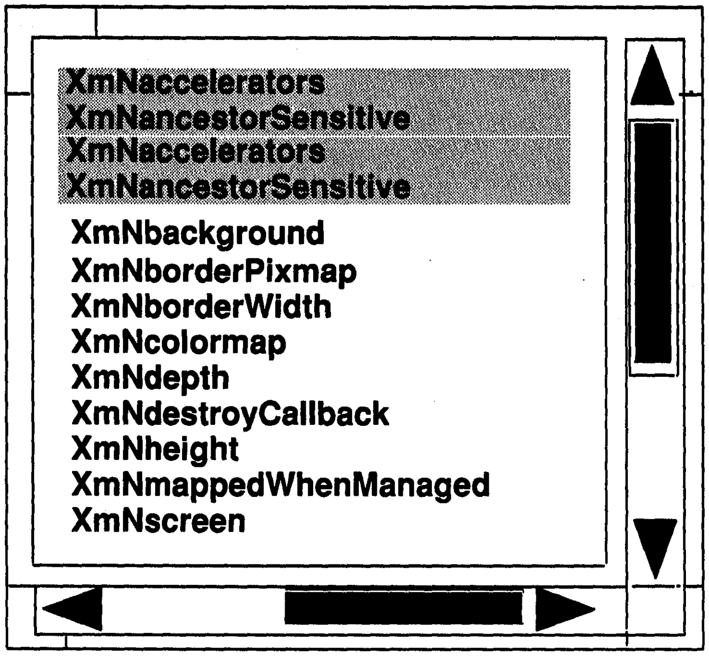
Range Selection Using a Mouse

Mouse users for range selection perform the following steps:

1. Position the mouse pointer over the object that starts the range.
2. Press the Select mouse button.
3. Drag the mouse pointer to the object that ends the range.
4. Release the Select mouse button to complete the range selection.

For example, to select four contiguous items from a list box, users position the mouse pointer on the first item, press the Select mouse button, drag the mouse pointer to the fourth item, then release the Select button. The appearance of each selected item changes as it is selected, providing a visual cue. Releasing the Select mouse button completes the range selection.

An alternative to dragging the mouse pointer from the start to the end of the range is to click the Select mouse button on the start, move the mouse pointer to the end of the range, and press the Shift+Select keys to complete the range selection.



Range Selection Using a Keyboard

Keyboard users for range selection perform the following steps:

1. Position the location cursor (insertion cursor for editable areas) using the navigation keys so that it is over the object that starts the range.
2. Hold down the Shift key and press the navigation keys to drag the cursor to the object that ends the range.
3. Release the Shift key to complete the range selection.

As with range selection using the mouse, each object's appearance changes providing the visual cue of selection. When the Shift key is released, the selection is complete.

For example, to select five contiguous items from a list box, users position the location cursor on the first item, press the Shift key, press the navigation keys to drag the location cursor to the fifth line, then release the Shift key.

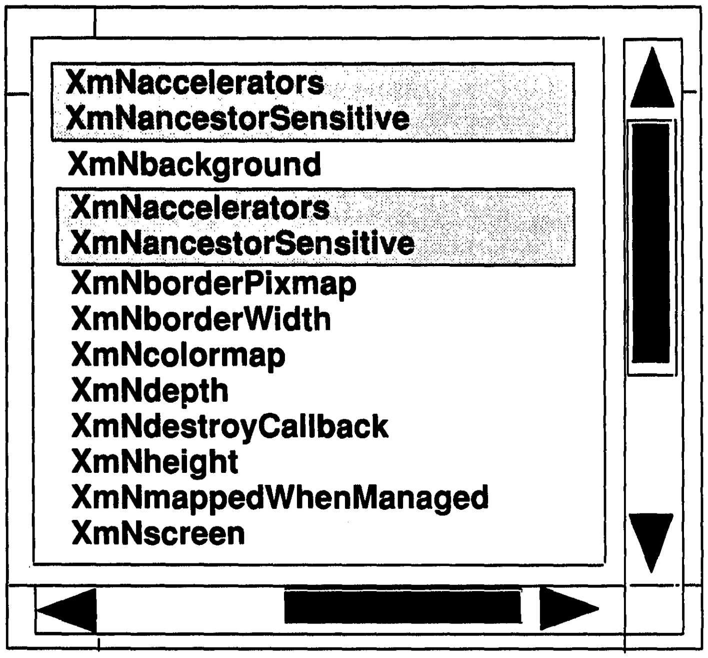
Alternatively, press the Select key to start the selection, press the navigation keys to move the location cursor to the end of the range, and then press the Shift+Select keys to complete the range selection.

Selecting Additional Non–contiguous Objects

Users can make one or more additional selections, forming a non–contiguous group of objects.

To begin an additional selection, users select a first object using single selection or a first range of objects using range selection. Other objects can be added to the selection group by repositioning the mouse or location cursor, and selecting the objects.

Like single and range selection, objects or ranges that are part of a non–contiguous selection provide a visual cue that they are part of the selection.



Users find the selection of additional non–contiguous objects, especially the selection of non–contiguous ranges, useful in text processing. Note also that the operation that performs a selection of additional non–contiguous objects works as a toggle. That is, if the object is not selected, the operation selects it; if the object is selected, the operation deselects it.

Making Additional Non–contiguous Selection Using the Mouse

Mouse users make an additional selection perform the following steps:

1. Select the first single object or range using the methods described in the preceding sections.
2. Position the mouse pointer on the next object they wish to select.
3. Press the Ctrl key and the Select mouse button to mark the next object or next range in the non–contiguous selection.
4. (For range selection only) Drag the mouse pointer to the end of the range, then release the Ctrl key and the Select mouse button.
5. Repeat steps 2–4 for each additional object or range in the non–contiguous selection.

For example, to select several non–contiguous items from a list box, users position the mouse pointer on the first item, click the Select mouse button to select it, move the mouse pointer to the next item for selection, press the Ctrl key and click the Select mouse button to select that item. The appearance of the selected list items changes as the item is selected, providing a visual cue to the selection.

An alternative to the press–drag–release operation is to press the Ctrl key and click the Select mouse button to begin the selection, reposition the mouse pointer at the end off the selection, then press the Shift key and click the Select mouse button. This alternate method for selecting multiple ranges is similar to the alternative method for selecting a single range described above.

Making an Additional Non–contiguous Selection Using the Keyboard

Keyboard users make additional (non–contiguous) selections perform the following steps:

1. Select the first single object or range using the methods described in the preceding sections.
2. Position the location cursor on the next object they wish to select.
3. Press the Ctrl+Select keys to mark the next object or start of the next range in the non–contiguous selection.
4. (For range selection only) Press the Shift key and press the navigation keys to drag the location cursor to the end of the range. Release the Shift key to mark the end of the range.
5. Repeat steps 2–4 for each additional object or range in the non–contiguous selection.

For the same example as above, to select several noncontiguous items from a list box, users position the location cursor on the first item, press the Select key to select it, move the location cursor to the next item for selection, press the Ctrl key and press the Select key to select that item. As with mouse selection, the appearance of the selected list items changes as the item is selected, providing a visual cue to the selection.

Alternatively, press Ctrl+Select keys to start the range, move the location cursor using the navigation keys to the end of the range, and then press Shift+Select keys to complete the range selection.

Deselecting an Object

A selection operation can be undone (canceled) by typing the Esc key before the operation is completed. Once completed, an entire previous selection can be deselected by making a single–object selection.

A previously selected object can also be deselected by toggling its selection state. Usually this requires positioning the mouse pointer or location cursor on the object and then pressing the Ctrl key and the Select mouse button (for mouse users) or by typing the Ctrl key and the Select key (for keyboard users). This deselection method is commonly used to deselect one object in a range of selected objects without deselecting the rest of the range.

A previously selected range of objects can be deselected by pressing the Ctrl key and using a drag operation to toggle their selection state.

Selecting the Default Action

Some objects have default actions associated with them. For example, An icons' default action is restoring itself to a normal window.

For mouse users, double–clicking the mouse with the mouse pointer over an object selects the default operation for that object. For example, double–clicking the Select mouse button with the mouse pointer over an icon restores the icon. Double–clicking the Select mouse button with the mouse pointer over the window menu button closes the window. Double–clicking the Select mouse button with the mouse pointer in a dialog box performs the default action associated with that dialog box.

For keyboard users, typing the Enter key with the location cursor over an object selects the default operation for that object. For example, typing the Enter key with the location cursor over an icon, normalizes the icon. Typing the Enter key with the location cursor in a dialog box performs the default action associated with that dialog box.

Auto Selection

Auto selection combines the act of moving the location cursor with the act of selecting the object. In the selection model presented in this chapter, users move the location cursor to an object and then explicitly select the object by pressing the Select mouse button or the Select key. When auto–selection is employed, users simply move the location cursor to an object and the object is selected; no explicit selection action is required.

Understanding AIXwindows Client Area Design

The AIXwindows window manager is responsible for providing window management services for the windows of all applications in the AIXwindows environment. Your application is responsible for organizing the client area of the main window, any subareas, and any secondary windows.

Organizing client areas is an important part of your design process. Depending on the nature of your application, you may choose to divide the client area into one or more subareas. Additionally, you may choose to design your application with secondary windows (dialog boxes). subareas and secondary windows visually reinforce the organization of your application and increase the user's sense of control over its operation.

- Client Subareas
- Grouping similar controls.
- Presenting multiple controls.
- Laying out AIXwindows application client areas.

Other articles discuss the use of controls, menus, dialog boxes, and help in more detail.

Client Subareas

As mentioned, you can divide the client area of your application window into one or more subareas. Client subareas are very application specific, with the possible exception of the menu bar.

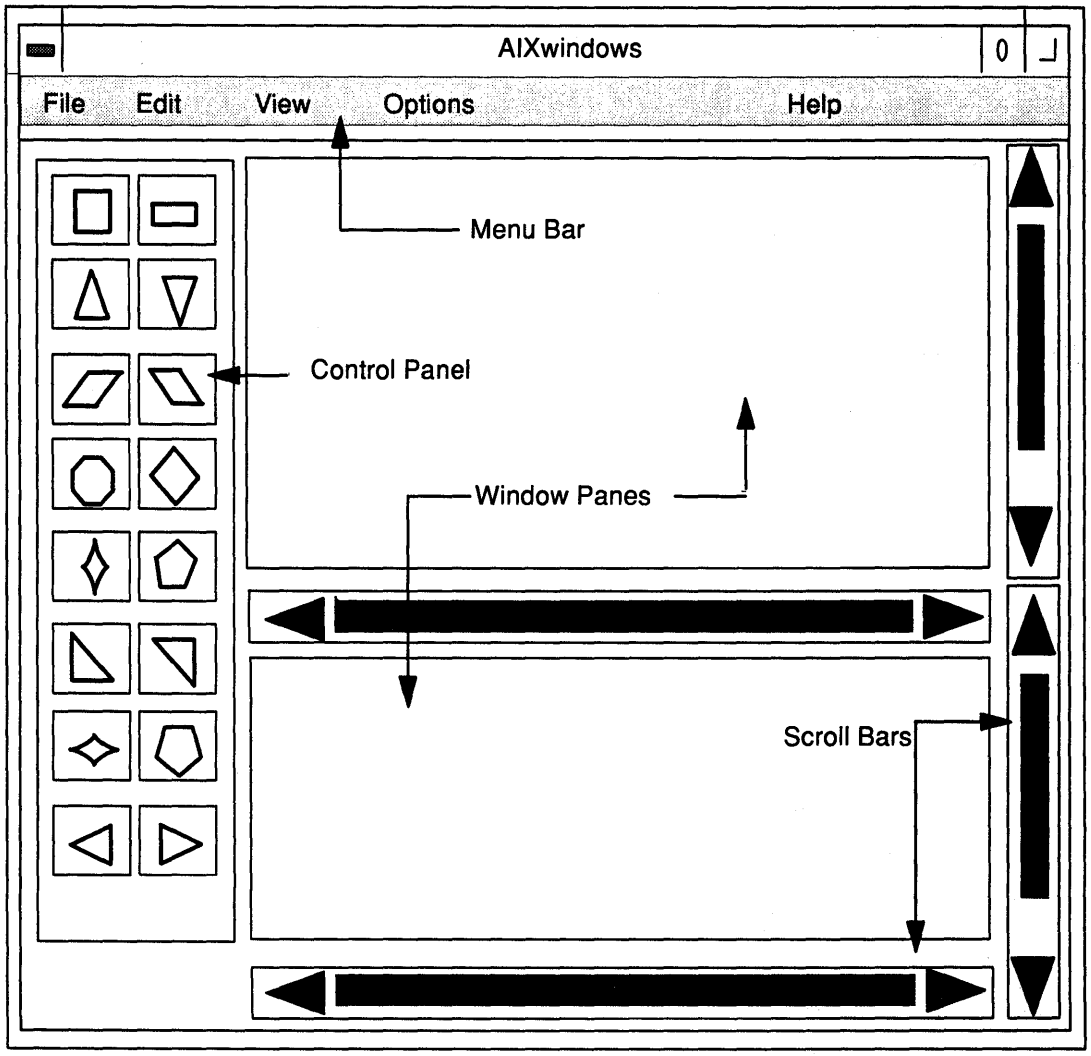
Menu Bar

The **menu bar** is the horizontal bar that appears just below the title area. It contains a list of menu topics from which users can select. A single letter mnemonic for each menu topic is underlined.

Keyboard users select a topic by pressing the F10 key to move the location cursor to the menu bar and then typing the mnemonic letter of the topic. Mouse users select a topic by positioning the mouse pointer over that topic and selecting it with the Select mouse button. Selecting a topic causes a pulldown menu to display selectable items related to that topic.

Note that commands are not included as topics in the menu bar because they would prohibit users from browsing the menu topics.

Because menus are a principal method of interaction between users and AIXwindows applications, most applications require a menu bar. Refer to AIXwindows Menus Overview for more detailed information about menus.



Control Panels

Some applications benefit by organizing part of the client area into a subarea called a **control panel**. A control panel is a group of like controls having similar functions. Control panels can either be part of the client area, if their use is required frequently, or part of a dialog box, if their use is required occasionally. Control panels are made up of the controls discussed in Understanding AIXwindows Application Graphic Controls.

Message Area

You may decide that it is more efficient for the user to view messages, but not warnings or messages requiring immediate action, within a subarea of the client area rather than in a separate message box. If so, the messages should appear at the bottom of the client area so that messages are not obscured by pulldown menus. The message can be either in a line reserved for messages or in a line temporarily used for the message and then returned to its previous use. Warnings and messages that require immediate action are not displayed this way. Rather, these are always displayed in message boxes to give them greater visibility in the workspace.

Command Line

Although the AIXwindows environment provides a graphics oriented, object–action selection model, your particular application may permit the use of typed commands to enhance the control users have over your application.

Command lines should generally run from border–to–border across the bottom of the client area, just below the message area, or directly below the menu bar at the top of the client area.

Client Controls

Each subarea can contain a variety of controls enabling the user to manage that subareas. Like subareas, the controls chosen for a particular sub–area, are very application specific. The controls you choose depend on the needs of the people who will use your application, and your application design.

Window Panes

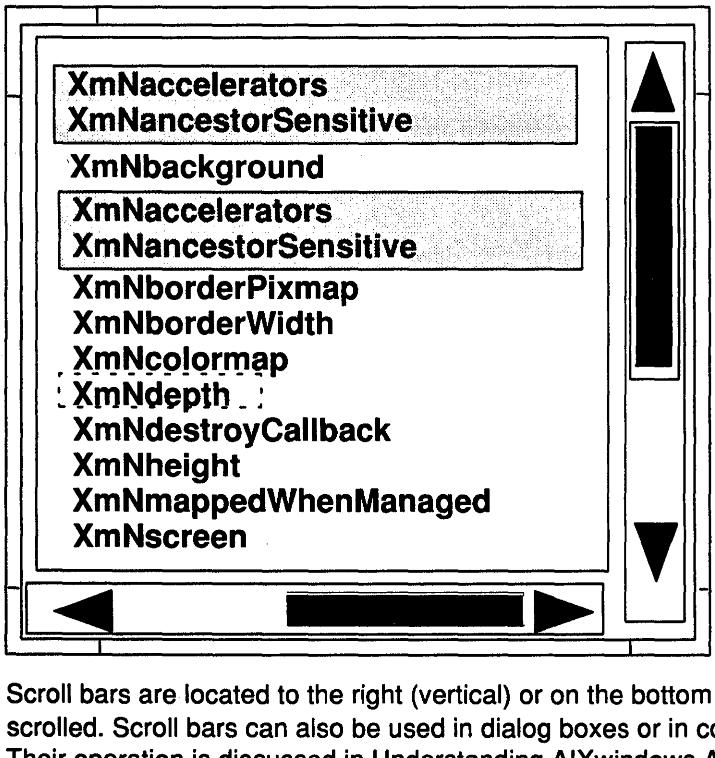
Depending on the needs of your application and the people who use it, you may decide that it is better to divide your client area into **window panes** rather than into fixed partition subareas.

Panes can be either vertical (one on top of the other) or horizontal (side by side). The user can resize panes by dragging the boundary between the panes. Making one pane bigger makes the other pane smaller, while the overall size of the window remains the same.

Scroll bars

People use scroll bars to scan rapidly through the contents of a window. The current location of the scroll bar is shown by the position of the slider in the scroll bar area.

Resizable windows and window panes require scroll bars if the information in them will require more space or become obscured by the border of the subareas. The following figure shows vertical and horizontal scrollbars.



Scroll bars are located to the right (vertical) or on the bottom (horizontal) of the area to be scrolled. Scroll bars can also be used in dialog boxes or in combination with other controls. Their operation is discussed in Understanding AIXwindows Application Graphic Controls.

Other Controls

The AIXwindows object–action selection model uses a number of other controls. These controls are graphical representations of real life controls and include the following:

- Push buttons
- Radio buttons
- Check buttons
- List boxes
- Entry boxes
- Scales
- Scroll bars.

Other Client Areas

In addition to organizing your application's client area into subareas, you may choose to organize your application and its operation using some of the other methods provided by the AIXwindows environment. These include pop–up menus, cascading menus, and dialog boxes (secondary windows). Like subareas, these other areas visually reinforce the organization of your application and increase the user's sense of control over its operation.

Menus

AIXwindows menus work just like real life menus. Menus enable users to choose from a list of possible selections. Besides the pulldown menus on the menu bar, the AIXwindows environment has the following types of menus:

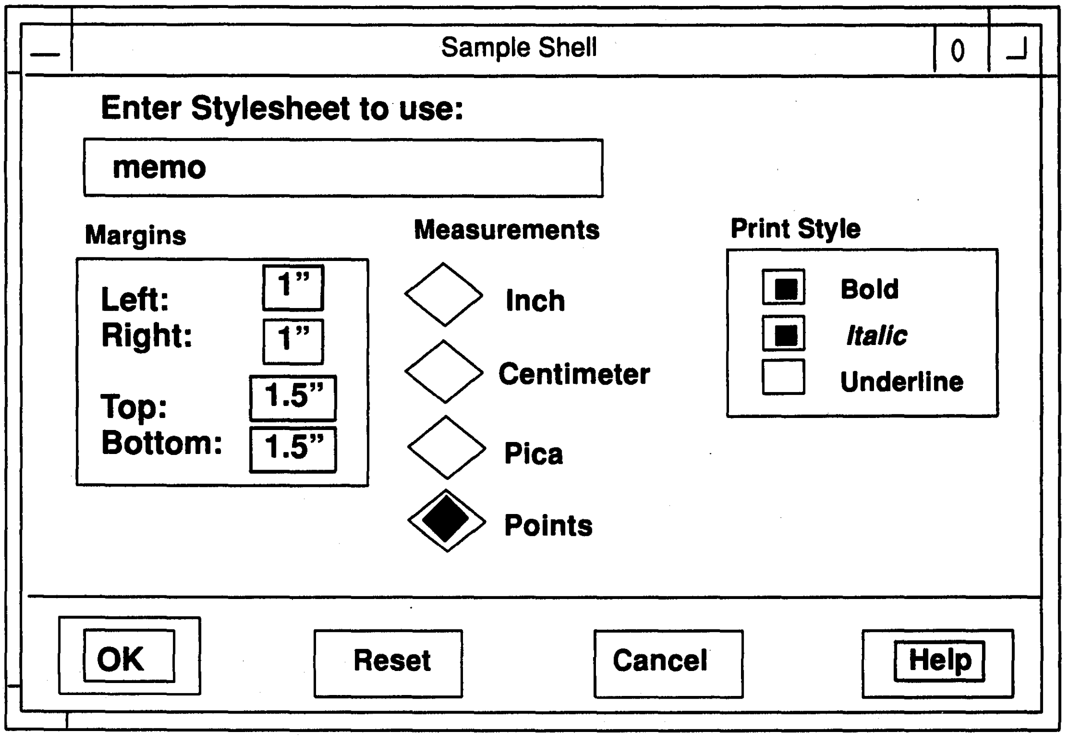
Pop–ups	Menus that pop up from nowhere rather than being related to a menu bar.
Cascading	Menus that cascade to the right and down from either pulldown or pop–up menus and provide a subsidiary level of selection.
Option	Menus that display from a dialog box. Those usually appear when a push button is pressed.

Dialog Boxes

Controls that are not frequently in use during the operation of your application can be included in a dialog box. A dialog box is a separate window from the application's main window, and contains controls that should be readily accessible but that do not need to be displayed permanently in the client area.

Grouping Similar Controls

Some controls perform similar functions or are logically related. For ease of use, as well as for proper visual design, these controls should be grouped into sets. This keeps the user interface of your application organized. Similar or related controls can be placed either in group boxes or in window subareas. Typically, group boxes occur in dialog boxes and are often surrounded by a simple frame. Window subareas often appear as part of an application's main window, and contain frequently–used controls that must remain readily available.



Grouped controls provide a visual cue that the controls are related to one another by isolating them from other controls. A group box or control panel usually has a title printed near it. Controls can be grouped in one row or column, or in multiple rows or columns.

Designing Grouped Controls with Push Buttons

Grouped controls containing push buttons should adhere to the following guidelines:

- Use push buttons sparingly
- Use push buttons only for frequently–used commands.

Unless you are trying to duplicate the physical appearance of an existing piece of equipment, place push buttons in dialog boxes in a row along the bottom of the box or in a column along the right side of the box.

Combining Controls

The AIXwindows environment does not restrict the combination of controls only to those combinations mentioned in this guide. The criteria for developing control combinations should always be consistent: will this combination, give control to the people who use it to work more efficiently and, as a result, become more productive.

Presenting Multiple Controls

The AIXwindows environment uses four basic ways of displaying multiple controls:

- Pull–down menus
- Pop–up menus
- Dialog boxes
- Control panels.

Pull–Down Menus

Pull–down menus usually contain push buttons, radio buttons, or check buttons as selection items. Selections can lead to dialog boxes or other controls. Menu items are always presented in a vertical column. To display a pulldown menu requires some degree of mouse movement or the use of an Alt key + mnemonic accelerator. While the selections of a pulldown menu do not appear until the menu is selected, the title of a pulldown menu is always displayed in the menu bar. pulldown menus combine a visual cue of their presence with an efficient use of space.

Pop–Up Menus

Pop–up menus, like pulldown menus, usually contain push buttons, radio buttons, or check buttons, and can have selections that lead to dialog boxes or other controls. Also like pulldown menus, pop–up menus are always presented as a vertical column. Pop–up menus are associated with a particular area of the screen. The advantage of pop–up menus is that they require no mouse travel; they simply pop–up at the current mouse location (provided that location has a menu associated with it). While pop–up menus take up no screen space until they are displayed, they provide no visual cue to their existence.

Dialog Boxes

Dialog boxes can contain all the button, box, and valuator controls. Usually, when the dialog box is displayed, all necessary controls are present. However, your application may require some extra display operations that should be in the dialog box but need not be displayed all the time. If so, you can include an option menu push button in your dialog box.

Dialog boxes allow a lot of flexibility in the arrangement of controls. Controls can be grouped in boxes, organized in rows or columns, and separated by white space for better visibility.

Message boxes are usually displayed by the application without any explicit action. Dialog boxes, on the other hand, are usually displayed as a result of some explicit action. Dialog boxes can be displayed directly using a keyboard accelerator. This saves the time and extra steps required when selecting them from a menu. Like pop–up menus, dialog boxes use space efficiently because they are not visible until displayed, however, this means they do not provide a visual cue to their existence. Dialog boxes are removed from the workspace when their primary window is minimized. Dialog boxes are returned to the workspace when their primary window is restored.

Control Panels

Control panels, like dialog boxes, can contain all the button, box, and valuator controls. They also offer the same flexibility of arrangement. Control panels, since they are permanently displayed, offer the potential of having frequently used controls always available. Being displayed, they offer a strong visual cue to their existence, but they do take up screen space.

Laying Out AIXwindows Application Client Area

The nature of your application may be such that you will need to design control panels or dialog boxes specific to your situation. When doing so, it is important for the AIXwindows–conformity of your application to use the following criteria:

- Arrange controls in natural scanning order
- Arrange controls in the sequence people use them
- Allow users to adjust the client area
- Choose the appropriate control for the job
- Know when to use a pop–up menu and push buttons

- Know when to use dialog boxes and menus
- Align columns of controls
- Use defaults consistently and for common settings

Arrange Controls in Natural Scanning Order

Design the layout of your application windows according to the natural scanning order of the people who will be using your application. In most cases, this order will be from left to right and from top to bottom.

Arrange Controls in the Sequence People Use Them

Intimately connected to the use of natural scanning order in control layout is the arrangement of controls in the sequence in which people will use them. The natural scanning order gives the position; the sequence of use gives the priority.

For example, suppose you have a dialog box that has push buttons that accept changes, test them, restore original values, and cancel the dialog box without making changes. Western conventions dictate that the push buttons be displayed from left to right. The sequence of use suggests that the OK button should be on the left (as the most frequently used button), followed by Apply, Reset, and Cancel.

Adjusting the Client Area

To increase the control users have over your application, your application should allow users to adjust the client area to fit their needs.

If your application uses window panes, it should allow users to adjust the size of the panes to suit their needs by repositioning the **sash**, the border separating the two panes. When one pane increases in size, the other pane decreases by the same amount. The overall sizes of the window frame and the client area do not change as the panes are resized.

Users should be able to adjust the sash using either a mouse or a keyboard operation. Moving the sash with the mouse is typically a button–press, drag, button–release operation like moving a window using the title area. Moving the sash with the keyboard typically requires a selection and the use of the navigation keys.

Choosing the Appropriate Control

Radio buttons, option menus, and list boxes can all be used to choose one option from a list of multiple options. Choosing the right control for the job depends on the number and nature of the options in the list.

For choosing a single option from among a small number of mutually exclusive options, a radio button is usually the easiest for users to operate. For more options, an option menu push button takes up a small amount of space and is relatively easy to use. For many options, the list box is the easiest for people to use; it also allows multiple items to be chosen at one time.

Deciding Between a Pop–Up Menu and Push Buttons

Pop–up menus provide users with quick access to application functions. So do control panels containing push buttons. Generally, pop–up menus are preferable when users are focused on their work areas. In these situations, moving the mouse between a control panel and the work area would be distracting.

Push buttons and a control panel are preferable when users make frequent selections, need to make several selections at the same time, or are already manipulating the mouse primarily in the control panel area.

Deciding Between Dialog Boxes and Menus

You should design your application so that it is consistent with other AIXwindows applications. To do so, you should understand when to use dialog boxes and when to use some other method of control. In particular, you should know the difference between dialog boxes and menus.

As you design your application, you will encounter many instances in which the same objective can be accomplished with either a dialog box or a menu. The menu selections act similar to the controls used in most dialog boxes. However, there are differences.

A menu is short–lived. It appears quickly, but exists only while a selection is being chosen. As soon as the selection is made, the menu disappears. A dialog box, on the other hand, can be displayed until told to go away, but usually takes up more workspace. While the dialog box is displayed, users can make several different selections.

Additionally, a menu is usually modal in nature. Until a menu goes away, users can't interact with any other part of the application. Dialog boxes, on the other hand, are frequently modeless. Users can still interact with other parts of the application while the dialog box is displayed.

Thus, if a modeless state were required, a modeless dialog box would be the appropriate solution. In the case of users browsing current settings or making a single selection, a menu would be faster. However, when several selections need to be made, a dialog box would be a better design choice.

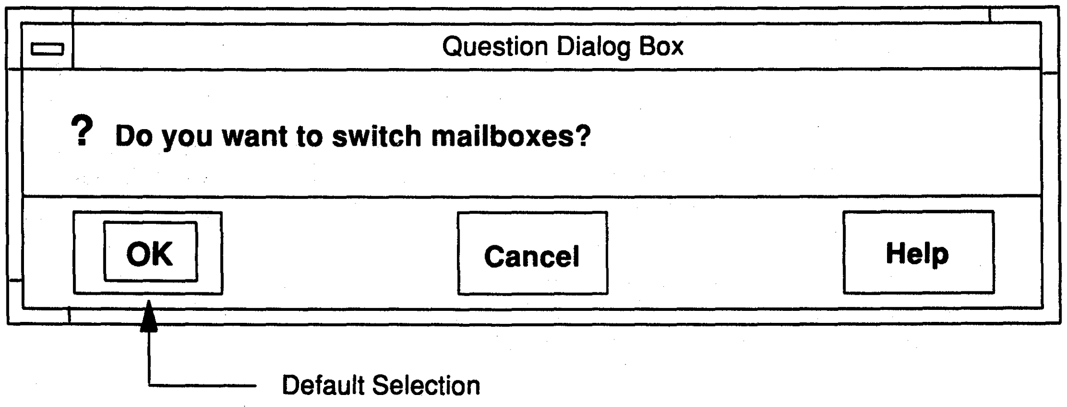
Aligning Columns of Controls

While push buttons are usually placed in a row along the bottom of the dialog box, check buttons and radio buttons are frequently placed in columns. When using columns, align the check buttons or radio buttons vertically so that the location cursor doesn't bounce around as users tab through the selections. Proper vertical alignment also enables users to slide the mouse pointer in a single direction rather than having to move all over the window to reach misaligned controls.

Using Defaults Consistently

Your application should use default values for common settings or obvious selections. A default selection should be easily distinguishable from other selections. Typically this is accomplished with an extra border around the default selection.

Default check buttons need only display with a check in them. The default selection in a set of radio buttons also displays selected. A default push button is typically indicated by a double border. Users activate the default push button by double–clicking the Select mouse button with the mouse pointer in the area containing the push button or by pressing the Enter key on the keyboard.



? Do you want to switch mailboxes?

Understanding AIXwindows Application Graphic Controls

Users control applications in the AIXwindows environment using a number of graphical controls. These controls are of the following three types:

Buttons	Like the control buttons in real life, users generally operate AIXwindows graphical control buttons by pressing them with the Select mouse button or the Select key.
Boxes	Like boxes in real life, graphical control boxes generally contain groups of related items
Valuators	Like some analog gauges in real life, valuators provide users with a way to specify or control incremental changes.

Types of Buttons

AIXwindows applications currently use three types of buttons: push buttons, radio buttons, and check buttons. Which button you use for your application depends on the situation you wish to control. Buttons should be of a size large enough so that users can easily position the mouse pointer on them.

Push Buttons

A **push button** consists of two parts:

- A graphical image that represents the button.
- A label or icon describing the action invoked by the button.

When users position the mouse pointer anywhere on a push button and click the Select mouse button, or position the location cursor on the push button and press the Select key, the action represented by the push button occurs.

The label of a push button should be short, usually a verb for action push buttons, such as Cancel or Apply. Response push buttons can have text such as OK or Yes, however, the question that prompts the response should be carefully worded to avoid ambiguity.

A push button can be used to display another dialog box. This is the case with an option menu in a dialog box. A push button used this way should provide a visual cue to its functionality by following its button label with an ellipsis (...).

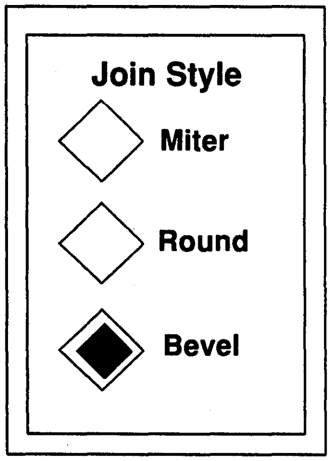
Radio Buttons

A **radio button** consists of two parts:

- The graphic image that visually represents the button.
- A label that describes the choice represented by the graphic image.

Each radio button represents a single–choice selection. Radio buttons are always in a fixed set of at least two buttons and always represent mutually exclusive choices.

Conceptually, radio buttons work like the buttons on a car radio (from which they derive their name). Users select or deselect a radio button by clicking the Select mouse button when the mouse pointer is over the radio button or by pressing the Select key when the location cursor is over the radio button. Like a car radio, when one radio button is selected, the previously selected button is deselected.



Radio buttons that refer to similar kinds of options should be grouped in sets. Sets of radio buttons that refer to different kinds of options should be grouped separately. Sets can be arranged in either rows or columns. Use white space to visually separate multiple sets of radio buttons into a control panel.

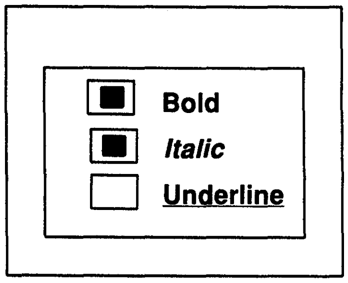
Radio buttons are usually circles in 2–dimensional environments and usually diamonds in 3–dimensional environments to distinguished them visually from check buttons which are usually square.

Check Buttons

A **check button** consists of the following two parts:

- A square box or button that is empty when deselected but that is filled in or contains some other visual cue when selected.
- A label that identifies the purpose of the check button. The label is at the same level as and to the right of the check button.

A Check button enable users to select choices that are not mutually exclusive. Users select or deselect a check button by clicking the Select mouse button when the mouse pointer is on the check button or by pressing the Select key when the location cursor is over the check button.



Types of Boxes

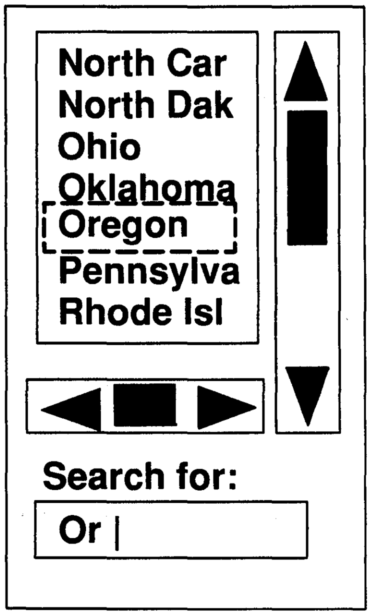
AIXwindows applications currently use two types of control boxes: list boxes and entry boxes. Like buttons, which box you use depends on the situation.

List Boxes

A **list box** typically consists of the following parts:

- A title that describes the purpose or contents of the list box. The title generally appears above the list box.
- A window containing the listings.
- Vertical and horizontal scroll bars, as needed. The scroll bars enable users to view the listings.

List boxes enable users to select from an existing list of items that is either long or variable in length.



Users move the slider on the scroll bar to change their current view of the list. To choose a selection, users position the mouse pointer on the selection and click the Select mouse button or position the location cursor on the selection and press the Select key. The standard methods for range and additional non–contiguous selections are also supported.

List boxes do not support mnemonics. Instead, they have a speed–search function that works as follows: When users type the first letter of an item in a list box, the box scrolls to the first occurrence of an item that begins with that letter. For example, if the list box contained an alphabetical listing of the states in the United States, a user would press the O key to view the states beginning with `Ohio`.

A list box can also be combined with an entry box to provide an incremental search function. For example, if the list box contained an alphabetical listing of the states in the United States, a user would press the O key to view the states beginning with `Ohio`. If the user then typed **Or**, the list would move to `Oregon`.

Double–clicking the Select mouse button in a list box chooses a selection and activates the default push button in the dialog box.

Entry Boxes

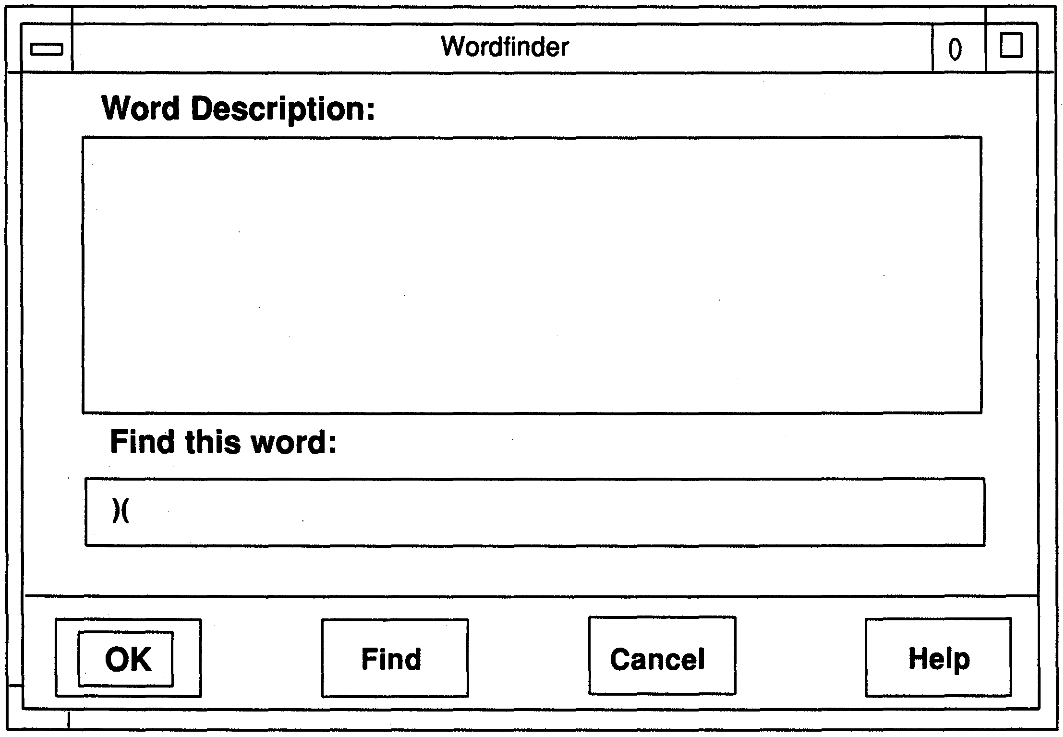
An **entry box** consists of the following parts:

- A title or label.

• The box in which text is entered.

Entry boxes enable users to enter text. The entry box may scroll horizontally if the text entered is longer than the box. The entry box may also be more than one line high, in which case it can have a vertical scroll bar like a list box.

The title describes what is to be entered in the entry box. Titles generally appear above or to the left of the box (in western countries).



When users move the mouse pointer into an entry box, the mouse pointer changes from the default shape to the shape of the text insertion cursor. Entry boxes follow the rules for basic text editing.

Pending Delete

Entry boxes include a function known as *pending delete.* Using this function, users can select a range of text to be overwritten in an entry box and then simply begin typing the replacement text. The selected text is deleted and the new text is inserted in its place as typed.

Text Cursor Shapes

A text insertion cursor shows where text will be inserted or overstruck. Insertion cursors should always provide users with a visual cue to the current text mode, insert or overstrike.

In insert mode, the cursor typically appears as a vertical bar or pipe (|) between two text characters. When users press a new character, the character appears to the left of the cursor. The cursor moves one character space to the right. In an inactive window or entry box the insertion cursor is de–emphasized.

In overstrike mode, the cursor typically appears as a block or underline located at the text character that will be replaced. When users press a new character, the cursor replaces the existing character with the new character and moves to the right to the next character. In an inactive window or entry box the block is not emphasized.

Typical text cursor shapes are shown in the figure.

	Insertion	Overstrike
Active	❙	█ or ▬
Inactive	| or ^	▒ or —

Your application should allow users to control whether insertion cursor blinks or not.

While several insertion cursors may appear on the workspace, only one cursor, the one in the window with the input focus, can be active; all other insertion cursors are inactive and have a de–emphasized shape.

An insertion cursor can change size and should be set to the height of the current font.

Pre–formatting Entry Areas

Where possible, the entry boxes of your application should be pre–formatted. Typical instances are entry boxes for supplying phone numbers or social security numbers. Preformatted entry boxes increase the uniformity of the data entry while easing the burden of remembering and correctly typing formats.

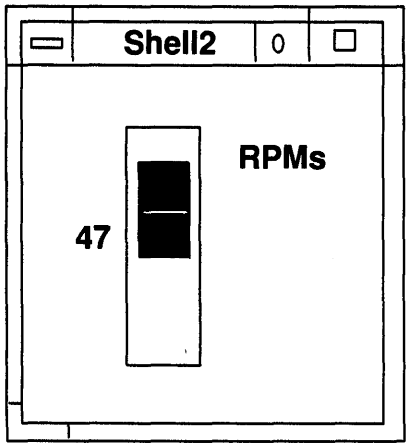
When the text entered in an entry box is all the same length (for example, phone numbers or social security numbers), you can implement **auto tabbing**. Auto tabbing speeds data entry in fixed–length fields by automatically moving the cursor to the next field as users finish making an entry in the current field. This saves moving the mouse or pressing the navigation keys.

Types of Valuators

The AIXwindows includes several types of valuators that enable you to provide users with analog–style controls.

Using a Scale

Your application could use a **scale** valuator. A scale enables users to enter a value from a range of values by adjusting in analog fashion a sliding arrow to a specific position along a line.



A scale valuator consists of the following components:

Scale bar	The scale bar may contain tick marks and represents the range of available values.
Slider	The slide arrow marks the currently chosen scale value.
Digital readout	The digital readout is an optional number directly opposite the slider that is the digital representation of the currently chosen analog scale value.

Using a Scroll Bar

People use scroll bars to scan rapidly through the contents of a window or to choose from a continuously variable set of values such as color intensity. The current location or setting of the scroll bar is shown by the position of the slider in the scroll bar for text windows or by example for variable choices such as color intensity.

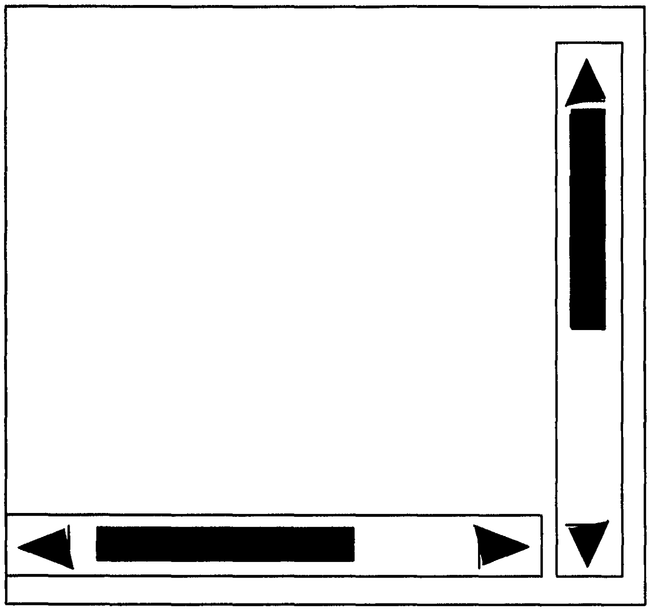
If users try to scroll beyond the end of the text, nothing should happen.

Scroll Bar Components

Scroll bars have the following components:

Scroll region	The scroll region is the "background" of the scroll bar and represents visually the length of the area that users can scroll.
Slider	The slider represents the window through which users look at the displayed data. Put another way, the position of the slider box on the scroll region provides a visual cue that marks the location of users' viewpoint in relation to the total scrollable area.
Stepper arrows	The stepper arrows enable users to scroll incrementally through the data and provide a visual cue to the direction of the scrolling movement.

Your application can use either horizontal or vertical scroll bars or both. The slider moves back and forth in the scroll region showing the position of the currently displayed section relative to the entire contents.



Operating Scroll Bars

Viewing text or graphical information through a window is like viewing the stars through binoculars. To change the view of the sky, users move the binoculars, not the stars. When the binoculars move up, the stars appear to move down; whichever direction the binoculars move, the stars appear to move in the opposite direction. Similarly, when people use a scroll bar to view a file, the file appears to move in the direction opposite to the movement of the slider. For example, in a text window, if the slider of a vertical scroll bar moves up, a text display seems to move down as previous lines in the file appear at the top of the window.

The following list discusses the different ways users can operate a scroll bar.

Action	**Description**
Performing a selection operation on a stepper arrow.	Highlights the stepper arrow and moves the window through the underlying file by a single unit, in the direction indicated by the arrow.
Performing a press selection operation on a stepper arrow.	Highlights the stepper arrow and causes a continuous scroll, in unit steps, in the direction indicated by the arrow.
Performing a selection operation on the scroll region.	Moves the window through the underlying file by one window length minus one unit for overlap.
Performing a press selection operation on the scroll region.	Continuously moves the window through the underlying file by one window length minus one unit for overlap.
Performing a drag selection operation on the slider.	Moves the slider and continuously moves the window to a location consistent with the new slider location.

Automatic Scrolling

When users drag the mouse pointer (with a mouse button pressed) beyond the top or bottom of the window, your application should continue the selection by scrolling in the direction of the mouse pointer. This automatic scrolling operates at the same speed as when users press on the directional arrows. The automatic scrolling ends when users move the mouse pointer back into the window or release the Select mouse button.

Slider Size

The slider itself may vary in size to represent the proportion of the entire contents currently covered by the window.

Scroll bars are usually located to the right or on the bottom of the area to be scrolled.

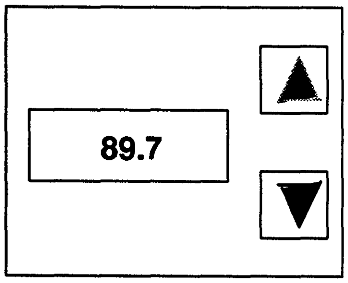
Application Extras

Besides the controls supplied by the AIXwindows toolkit, other dialog box controls can be supplied by the application as needed. One such application extra is the stepper button. A **stepper button** typically consists of the following parts:

- A title or label.
- A group box or panel separating the stepper button visually from the other parts of the dialog box.
- A text box displaying the current value of the stepper button.
- A scroll bar for stepping through the list of values to find a new value.

A stepper button enables users to select a value by scrolling through a circular list of possible values. The stepper button is similar in operation to the digital readout of a stereo tuner where pressing the button steps the read–out through the available radio stations.

The values that read out as people use the stepper button can be either letters or numbers, but they should be in consecutive order, either alphabetical or numerical, as opposed to random order. Users should be able to anticipate the appearance of values.



Combining Controls

Entry boxes and list boxes are often used in combination to provide users with particular capabilities. Some common examples are the following:

incremental searching	As users type each letter of an entry in the entry box, the list box moves to the part of the list that matches what has been typed so far. When the desired item appears in the list box, users click the Select mouse button to select the item. The list box can still be used in the normal way.
automatic entry	Users scan the items in the list box and click the Select mouse button to put the current item in the entry box. The entry box can still be used in the normal way.

List boxes used in incremental searches or in automatic entry must be single selection.

Understanding AIXwindows Menus

A **menu** typically consists of a title and a list of selections. Similar to restaurant menus, menus in the AIXwindows environment display a list of selections from which the user chooses an appropriate action. Menus provide the user with a simple means to quickly access the functions in your application. This article discusses the following aspects of menus:

- What types of menus there are
- What components make up menus
- How users operate menus
- What are the standard AIXwindows menus
- How to design AIXwindows menu extensions.

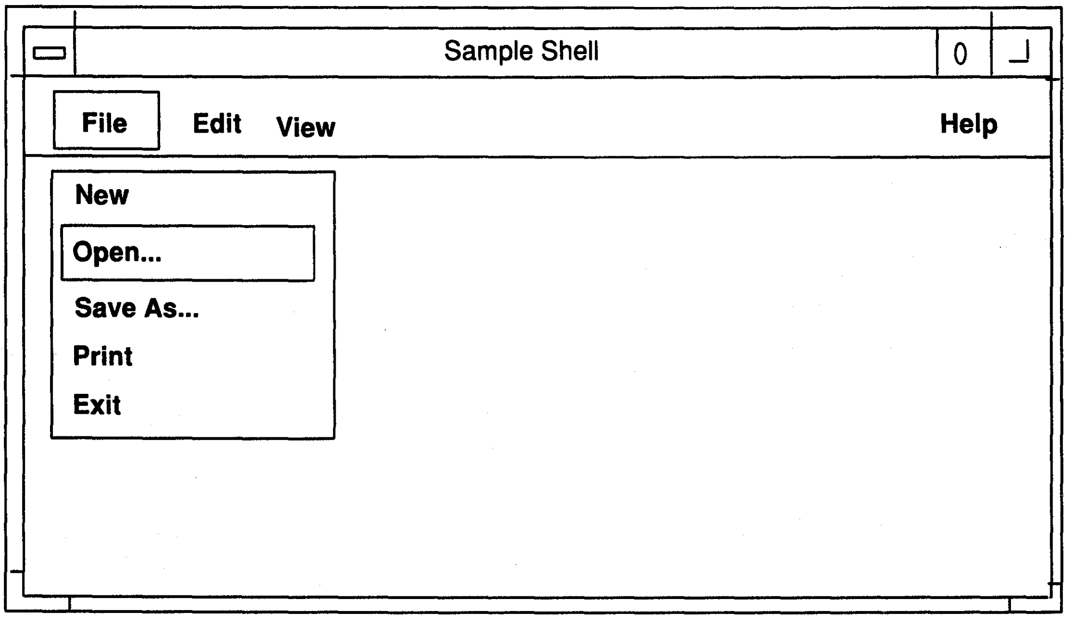
What Types of Menus There Are

The AIXwindows environment has the following four types of menus:

Pulldowns	Menus that pull down from a fixed location in the menu bar.
Option	Menus that display from a an option button in a window.
Popups	Menus that pop up at the current pointer location, wherever that may be.
Cascading	Menus that cascade to the right from another menu, providing more detailed selections related to the original menu selection.

Pulldown Menus

In most cases, pulldown menus provide an important part of the communication between users and application programs.

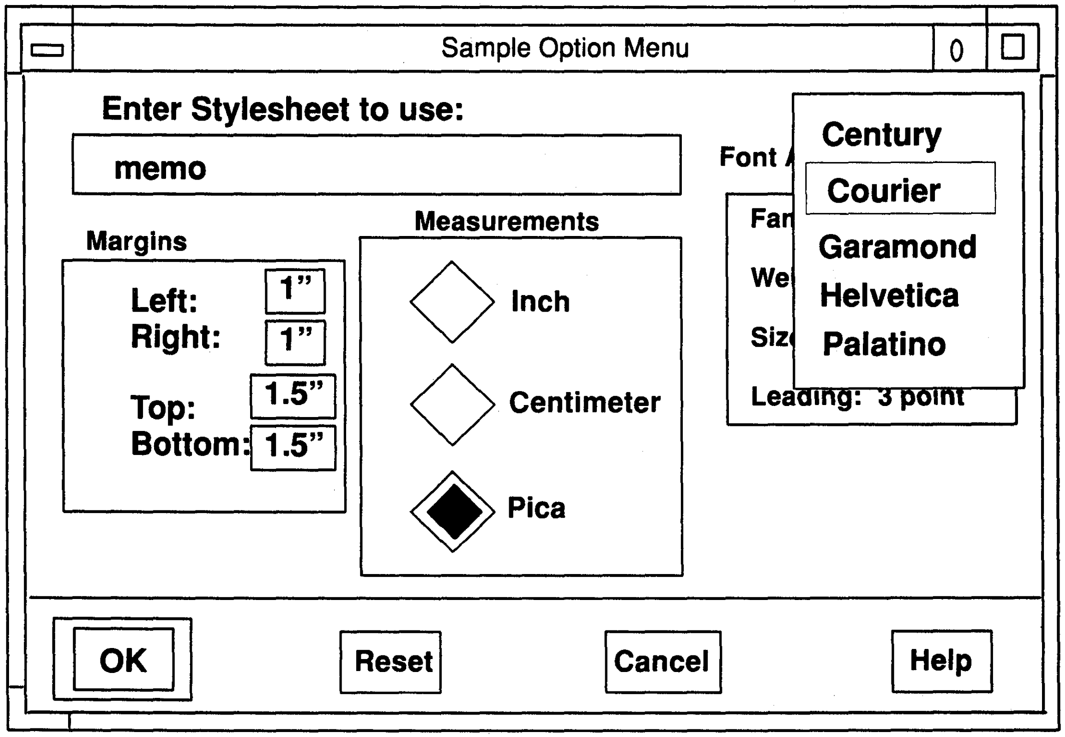


Pulldown menus, with the exception of the window menu, are always associated with an application's menu bar. The result is that, if you design your application to include a menu bar, the titles of available pulldown menus are always visible to the user running your application. Thus, the major portion of the functionality of your program is only a point–and–click (or point–and–drag) away from the user's fingertips.

Additionally, the user soon learns to drag the pointer across the titles in the menu bar, displaying each pulldown menu in turn, thus providing a handy table of contents to your application's functionality. This is another way of giving the user control as it enables the user to browse your program's functionality, refreshing their memory, rather than forcing the user to remember the arcane command–line syntax of a particular function.

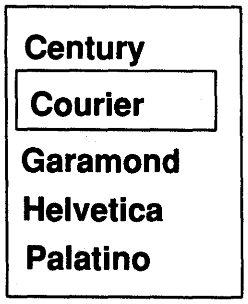
Option Menus

Windows which include selections from a list, but where space is at a premium are candidates for option menus. Only the option button is usually visible. The button shows the current value of the control. When users select the option button, the menu appears, showing the list of choices.



Popup Menus Save Space

Popup menus are also called context menus and have the advantage that they take up no permanent screen space. Not being associated with a menu bar, they simply pop up at the current pointer location. A workspace menu is an example of a popup menu.



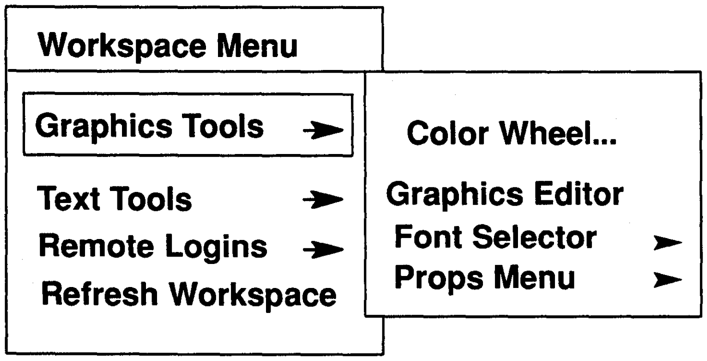
A second advantage of the popup menu is that it requires a minimum of mouse movement. To display a pulldown menu, the user must position the pointer on a title somewhere in the menu bar; to display a popup menu, the user need only press the Menu mouse button.

Popup menus are related to the context of the area in which they are selected. For example, a workspace menu that pops up when users position the pointer over the workspace and click the Menu mouse button may be associated with system–wide functions.

However, popup windows, by their very nature, do not provide a visual cue to their availability. The user must learn and remember that a popup menu is associated with a certain area. Therefore, your application design should not use popup menus too casually.

Cascading Menus Provide Further Selection Detail

Cascading menus add detail to pulldown and popup menus. You can think of them as submenus or child menus of other menus. Cascading menus provide you with a mechanism to organize menu selections in a tree structure, thus simplifying the presentation of complex selection lists. To maintain ease of use, the menu section tree should be no more than three levels deep. A cascading menu appears when users select or drag the pointer onto or across its title on the parent menu.



Cascading menus typically appear to the right of their parent menu selection. While cascading menus differ from the two other types of menus in the method of their appearance, cascading menus behave just like pulldown and popup menus as far as the choosing of a selection.

What Components Make Up Menus

All menus, regardless of type, have the same components.

Menus Have Titles

Menus have titles that name them. A menu's title should be unique to eliminate the possibility of confusion. The title should clearly indicate the purpose of the menu.

The title of a pulldown menu is on permanent display in the menu bar. The optional title of a popup menu displays at the top of the menu. The title of a cascading menu displays as a selection in the parent menu.

The titles of pulldown menus that appear on the menu bar employ single–character mnemonics as memory aids to increase the efficiency of the more experienced user. Mnemonics are explained later in this section.

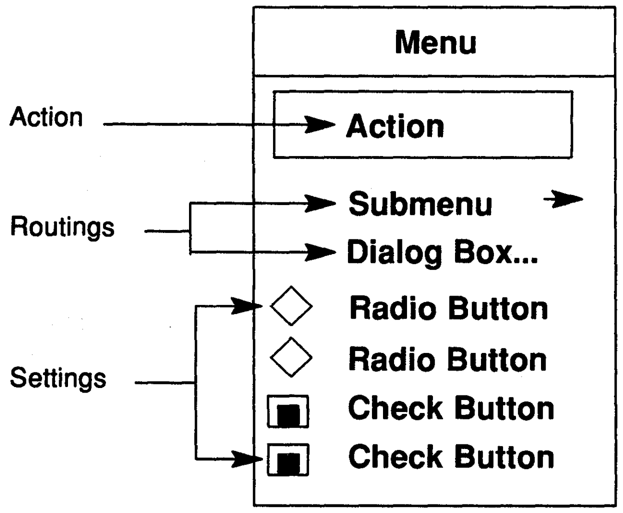
Menu titles are visually distinct (that is visually separated in some way) from the menu's selections. Typically, this is accomplished by placing a separator line below the title.

Menus Have Selections

Menu selections are listed below the menu title and, like the titles of pulldown menus, can also employ mnemonics. Additionally, a menu selection lists any keyboard accelerator associated with the selection. Selections can be text or graphics. Selections can also be grouped with a separator to provide a visual cue of similarity or related functionality.

AIXwindows menu selections can be one of three types. as shown in a sample menu containing the three selection types.

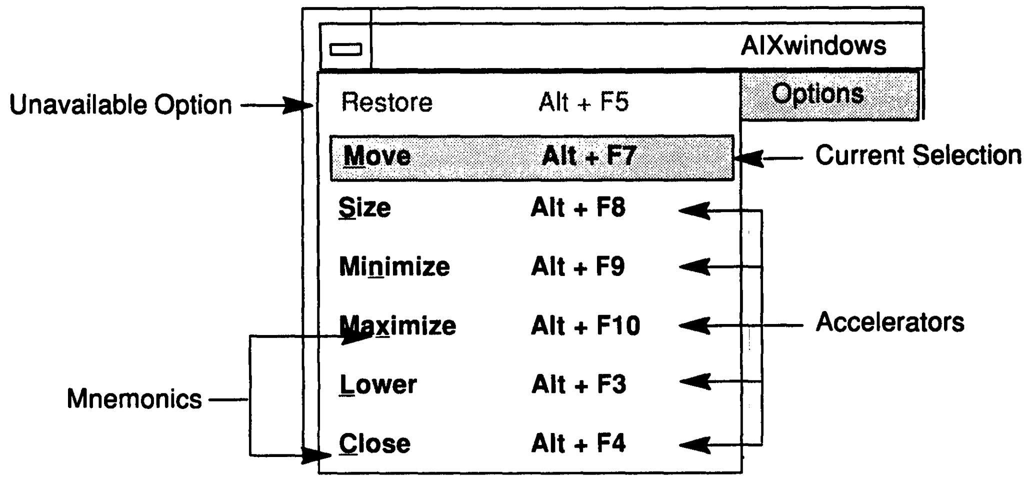
Type	**Description**
Actions	Issue a command or carry out an action.
Routings	Display a dialog box (dialog menu items are indicated by an ellipsis (. . .)) or a cascading menu (cascading menu selections are indicated by an arrow (–>)).
Settings	Set an application state using check buttons (multiple selections) or radio buttons (mutually exclusive selections).



Menu selections that are currently not available (disabled) are visually de–emphasized.

Menus Have Mnemonics

A **mnemonic** is a single character that provides a shortcut for making selection from the keyboard. Users press the mnemonic for a selection rather than using the navigation and select keys. In the AIXwindows environment, all menus (and the titles of pulldown menus on the menu bar) have mnemonics associated with their selections. AIXwindows menu mnemonics are a single letter, usually the initial letter of the selection. Typing the mnemonic for a selection chooses that selection, the same as if the selection were chosen with a mouse operation. If the title of a cascading menu is selected, the menu is displayed. To display a pulldown menu while the location cursor is outside the menu bar, press the Alt key and the mnemonic key, for example Alt+F4 to close a window.



Typically, an underline visually designates a letter as the mnemonic for a selection, as shown in the figure. When an initial letter cannot be used, as in the case where two selections begin with the same letter, some other letter in the selection name, preferably with some mnemonic value, should be chosen instead. If the mnemonic letter does not appear in the selection's text, it appears in parentheses after the text.

Mnemonics are only accessible when the menu containing them is displayed. The pulldown menus in a menu bar are always accessible (by pressing Alt+*mnemonic*) since the menu bar is always displayed.

Menus Have Keyboard Accelerators

Menu selections can also have **keyboard accelerators**, a key or key sequence that invokes a menu selection without displaying the menu. Keyboard accelerators do not have to be associated with each and every menu selection. In most cases, it is sufficient to provide accelerators only for frequently used functions. Providing accelerators should be a matter of utility, not design conformity.

If a keyboard accelerator exists for a menu selection, it appears right justified on the same line as and separated from the selection's text by enough space to make it visually distinct. Other than this, keyboard accelerators provide no visual cue to their existence and so users must memorize them. This is why frequently used functions make the best candidates for accelerators.

How Users Operate Menus

Users operate menus in two steps: first, they display the menu; second, they choose a selection. When a selection has been chosen (unless it leads to a cascading menu), the menu disappears.

Displaying Menus

In the case of pulldown menus, mouse users move the pointer to the title of the menu and press the Select mouse button. Keyboard users move the location cursor to the menu bar by pressing the F10 key, then press the pull down menu's mnemonic or use arrow keys to select the menu's title. An alternate method of displaying a pull down menu is to press the Alt modifier key and press the menu mnemonic.

In the case of popup menus, mouse users move the pointer over the popup area (for example, the workspace) and press the Menu mouse button. Keyboard users press the Menu key to display a popup menu.

In the case of cascading menus, mouse users move the pointer to the title of the cascading menu on the parent menu and click (or continue to press) the Select mouse button. Keyboard users use the navigation keys and the Select or Enter keys to achieve the same effect as the mouse user's actions.

Additionally, the user can access application menus using either of the following two methods:

Dragging	Drag the menu by pressing and holding down the mouse button to maintain a hold on the display. To make a choice, release the mouse button.
Clicking	Click the appropriate mouse button or press the appropriate key to display the menu. The menu remains displayed until either a selection is made or the display is canceled. A selection is made by moving the pointer to the selection and clicking the mouse button or by moving the location cursor to the selection and pressing the Select or Enter key.

Browsing the Menu Bar

The user can browse the menus listed on the menu bar by pressing the Select mouse button and dragging the pointer across the menu titles. As the pointer crosses each title, the menu associated with the title pulls down. To browse the menu bar from the keyboard, the user displays a menu and uses the left and right arrow keys to move laterally across the bar.

Choosing Menu Selections

To choose a menu selection, the user positions the pointer or location cursor on the selection and makes a selection action.

A person using the drag method drags the pointer onto the selection desired and releases the mouse button they pressed to initiate the drag process.

A mouse user using the click method slides the pointer onto the selection and clicks the appropriate mouse button.

A keyboard user using the click method uses the navigation keys to position the location cursor on the selection desired and presses the Select key.

A menu item that has been selected provides a visual cue of its selection. Typically, this cue is a change in color, either highlighting or reversed video. In a 3–D implementation, the current selection not only changes color, it also has a raised, 3–D appearance.

Releasing the mouse button or keyboard key selects the item under the mouse pointer or location cursor. If the release occurs on a command (action), check button, or radio button, the specified action takes place, and any menus displayed disappear. If the release occurs on the title of a submenu, the menu is displayed.

Avoiding Menu Selection

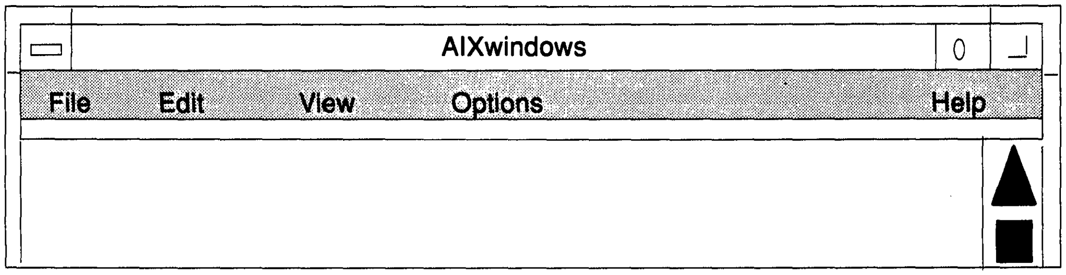
To avoid making a menu selection, a mouse user dragging the menu drags the pointer off the menu and releases the mouse button. A mouse user clicking the menu slides the pointer off the menu and clicks the Select button. A keyboard user presses the Esc key.

The Standard AIXwindows Menus

The following five pulldown menus provide general functions common to most applications. They are a part of most menu bars in the AIXwindows environment:

- File
- Edit
- View

- Options
- Help



The File and Edit menus have recommended contents. The View and Option menus are highly specific to each application, so only sample implementations are provided. This chapter also contains a sample Help menu. You can create additional menus for your application. The File and Edit menus (if used) should be the first two menus in your action bar. The Help menu is always the last.

While it is recommended that you include the standard menus in the menu bar of your application, your choice of menu titles and items depends on the nature of your application. Should your application require it, you should design more relevant titles and selections for your application's menu bar and menus, but do not change the meanings of words used in the standard menus.

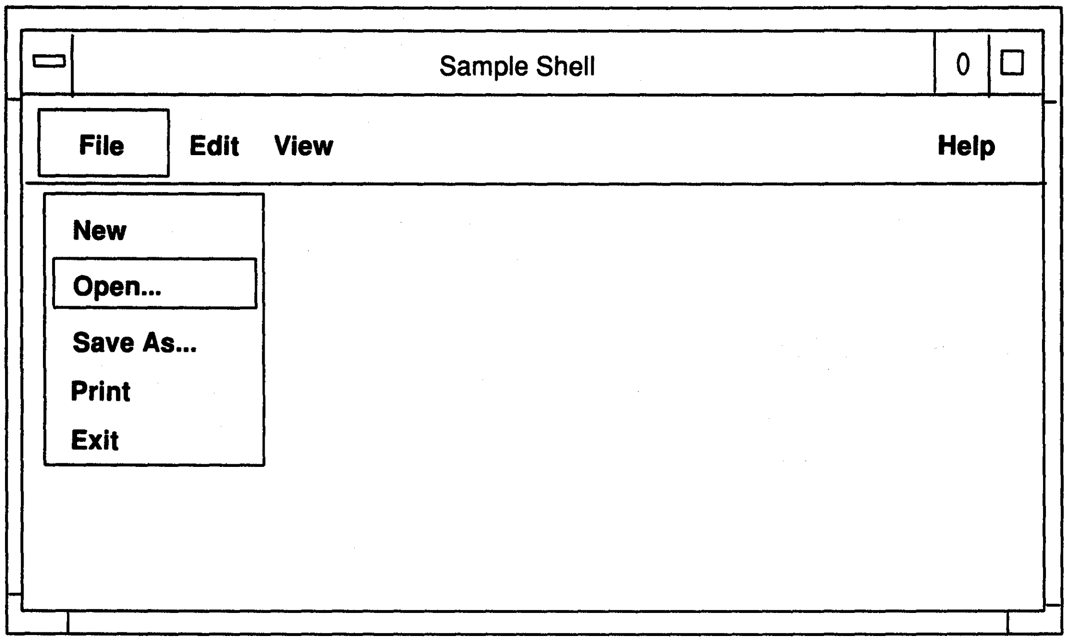
A Look at the File Menu

The File menu presents actions that deal with files in their entirety. All applications that deal with files should provide the File menu.

The selections in the menus are divided into four groups:

- Selecting actions that connect files to your application.
- Saving actions that transfer changes made to a storage medium.
- Output actions that send the changed file to an output device.
- Other actions.

These groups are important as you design extensions to the menu. Suppose you wanted to add an action called **Plot** to your application. You would place it in the File menu because it deals with the file as a whole. Within the File menu, you would place it with **Print** in the output group because **Plot** writes the entire file to an output device.



For consistent operation with other AIXwindows applications, if you include the File menu in the menu bar of your application, it should appear as the first title, placed on the far left, and with F as its mnemonic.

The **File** menu contains the following selections:

New	Creates a new file. The New operation clears existing data from the client area and replaces the current filename in the title bar with **Untitled** or some other application–generated name. If completion of the operation will obliterate current changes to the file, you must display a message box asking users whether they want to save their changes.
Open...	Opens an existing file. The Open operation prompts users for the name of the file by displaying a dialog box for the purpose. The title bar is updated with the name of the newly opened file. If completion of the operation will obliterate current changes to the file, you must display a message box asking users whether they want to save their changes.
Save	Saves the currently opened file to a storage device without removing the existing contents of the client area. If the currently opened file has no name, Save prompts users for a file name by displaying a dialog box for the purpose. Saved along with the file should be the current state of the file, including: window size, window location, and scrolling position within the file. This enables users to reopen the file and begin their work where they left off.
Save As ...	Saves the currently opened file under a new name. The Save As operation prompts users for the name of the file to be saved by displaying a dialog box for the purpose. If users try to save the new file under an existing name, the Save As operation displays a warning message alerting them of a possible loss of data.
Print	Schedules a file for printing. If your application requires specific printing information before printing, the Print... selection displays a dialog box in which to gather it.

Exit Ends the current application and closes all windows associated with it. This action is equivalent to closing all primary windows of the application. If completion of the operation will obliterate current changes to the file, you must display a message box asking users whether they want to save their changes.

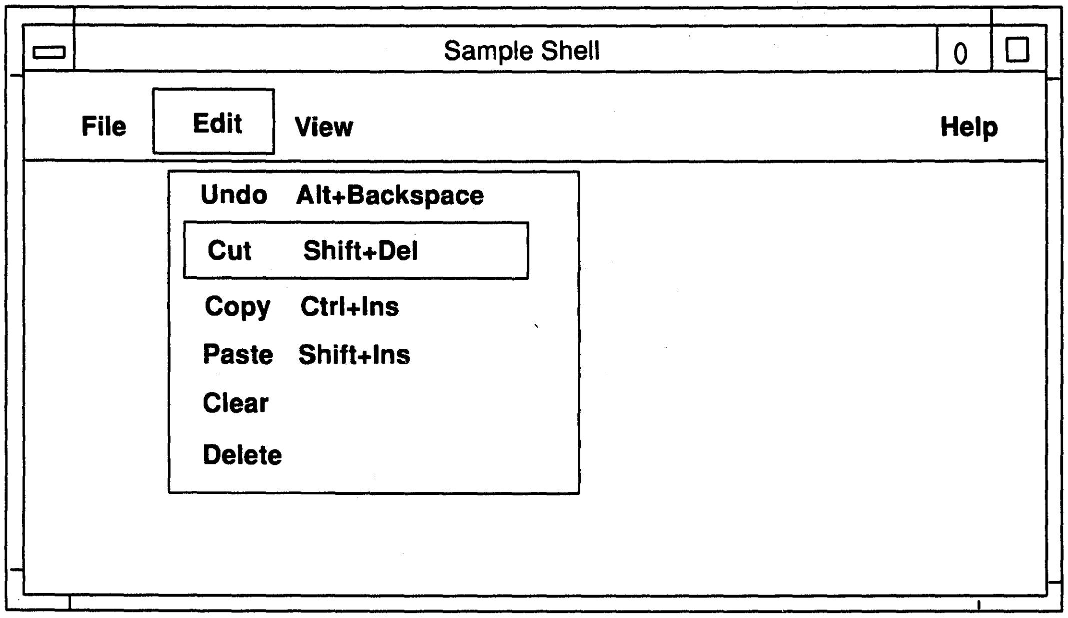
You are encouraged to include this action even though it duplicates much of the functionality of the Close action in the window menu. This assures users have a way to end the application even if they are not running the AIXwindows window manager. If your application does not have a File menu, put Exit at the end of the first pulldown menu.

A Look at the Edit Menu

The Edit pulldown menu contains actions that modify the contents of the data which which your application is currently dealing. Many of the actions relate to cut and paste functionality.

The standard edit selections are grouped as follows:

- Undo actions that reverse the effect of previous actions.
- Actions that relate to the system–wide clipboard.
- Other Actions.



For consistent operation with other AIXwindows applications, if you include the Edit menu in the menu bar of your application, it should appear as the second title from the left and with E as its mnemonic. The Edit menu contains the following selections:

Undo (Alt+Backspace)
Reverses the most recently executed action. To provide a visual cue to users, the Undo selection should be dynamically modified to indicate what is being undone. If the most recently executed action were a paste, the action name would be Undo paste. Your application should be able to undo all of the actions in the Edit pulldown. Text applications should also support Undo typing which restores text after it has been selected and replaced by newly typed text.

Cut (Shift+Del) Removes a selected portion of data from the client area to the clipboard buffer. Your application determines whether the area that used to be occupied by the removed data is left blank or whether the remaining data is compressed to fill in the space. Usually graphics applications leave the space blank while text application compress the remaining text to fill in the space.

Copy (Ctrl+Ins)
Copies a selected portion of data to the clipboard without removing the original data from the client area.

Paste (Shift+Ins)
Pastes the contents of the clipboard into a client area at the selected location. Your application determines whether the pasted data is reformatted to fit in the client area and whether existing data moves to create room for the pasted data. Text applications will typically reformat the pasted text to fit into the margins of the text field and they will move the existing text to make room for the new text. Graphics application might do neither. They might insert the graphics unmodified and overlay existing data.

Clear (Optional) Removes a selected portion of data from the client area without copying it to a clipboard buffer. Clear erases the data, but leaves the space formerly occupied by the data.

Delete (Optional) Removes a selected portion of data from the client area without copying it to a clipboard buffer. The remaining data is compressed to fill the space formerly occupied by the deleted data.

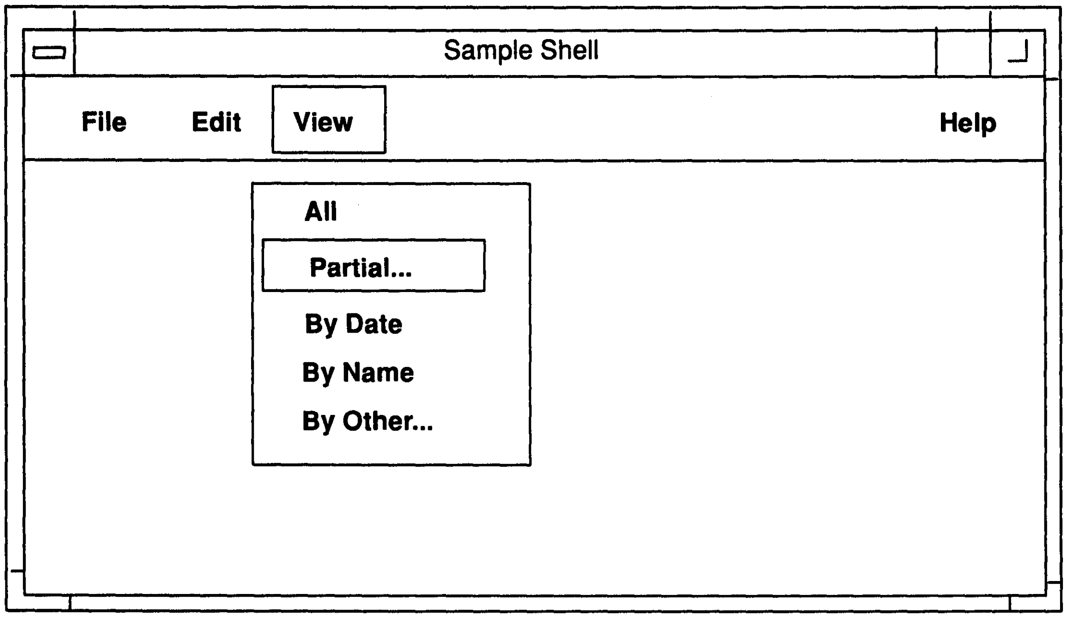
A Look at a View Menu

A **View** menu enables users to control how data is displayed. The data itself remains unchanged. Entries in the menu control such aspects of presentation as:

- Appearance of the data. Examples may be iconic or text–based.
- How much of the data is presented.
- In what order the data is sorted.
- To what level the data is summarized.

For consistent operation with other AIXwindows applications, if you include a **View** menu in the menu bar of your application, it should have the **V** as its mnemonic.

The content of **View** menus is very application specific.



A sample menu for a hypothetical file browser contains the following entries are:

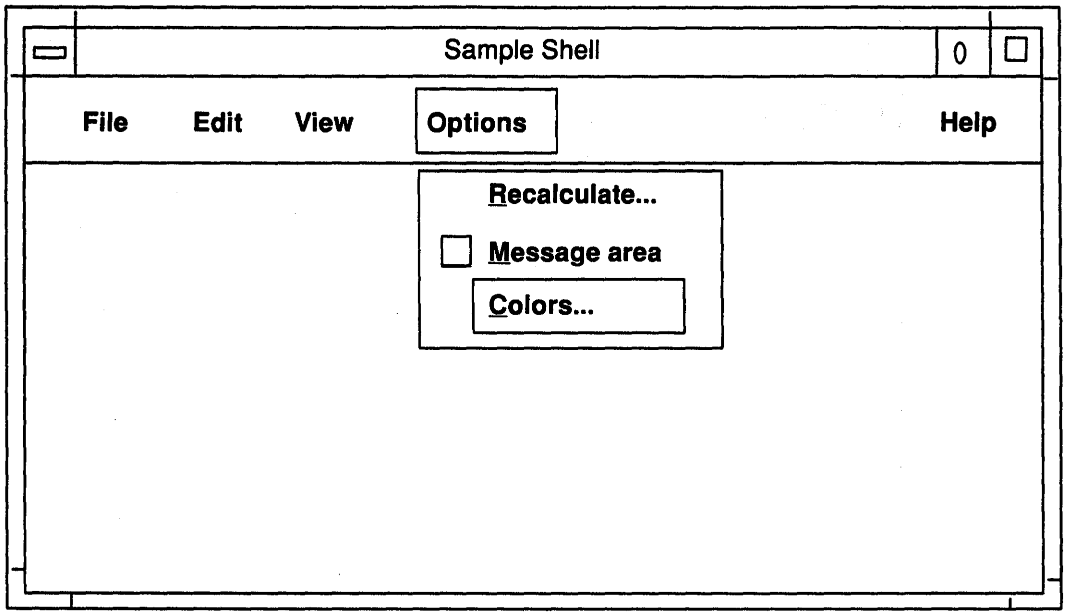
All	Views an entire list of items.
Partial	Views a partial list of items. The selection criteria are determined by a dialog box.
By Date	Views a list ordered by date of entry.
By Name	Views a list ordered by item name.
By Other	Views a list ordered by selection criteria determined by a dialog box.

Remember that the menu above is only an example. The needs of your application control what entries you place in your View menu.

A Look at an Options Menu

An Options menu enables users to customize various aspects of your application. Just like the View menu, the Options menu's content depends entirely on the needs of your application.

For consistent operation with other AIXwindows applications, if you include a Options menu in the menu bar of your application, it should have the O as its mnemonic.



The content of Options menus is very application specific. Entries in the menu for a hypothetical spreadsheet application might include such entries as:

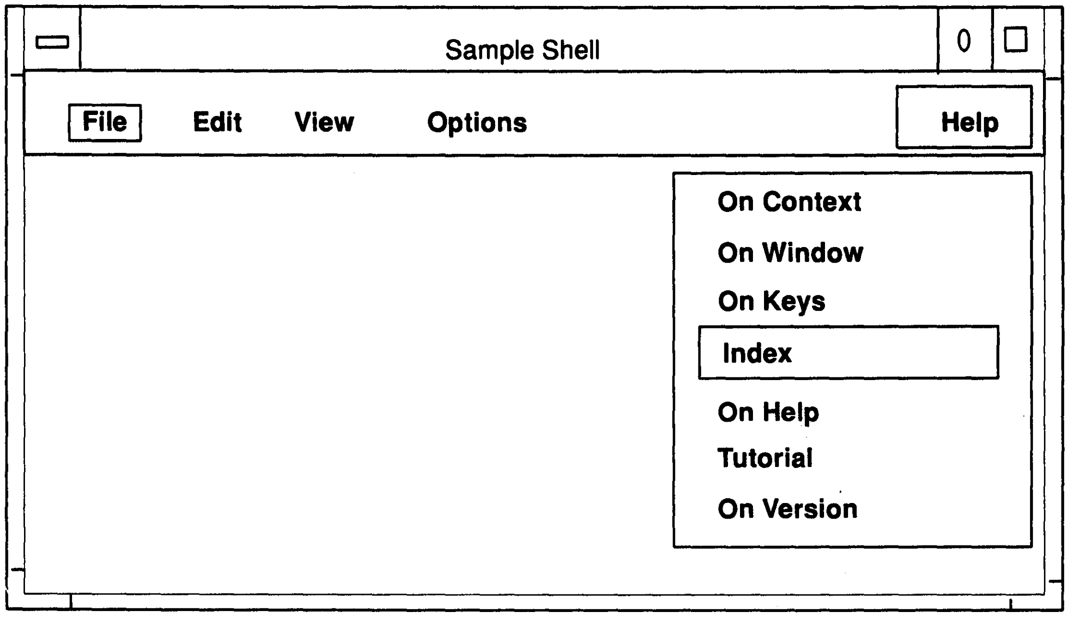
Recalculate	Displays a dialog box to control when cell formulas are recalculated.
Message area	A check box that controls whether a message area is displayed. The message area shows helpful explanations of commands for novice users.
Colors...	.Displays a dialog box used to select colors used by the application.

Remember that the menu above is only an example. Select entries appropriate to your application.

A Look at the Help Menu

Good applications provide help facilities for people to use when the need arises. No matter how intuitive you design your application to be, sometimes users get stuck.

For consistent operation with other AIXwindows applications, if you include a Help menu in the menu bar of your application, it should appear as the last title, placed on the far right of the menu bar, and with H as its mnemonic.



The sample menu contains the following selections:

On Context	Provides context–sensitive help about the specific situation that exists when the help was requested.
On Help	Provides information on how to use the application's help facility.
On Window	Provides general information about the operation of the window from which the help was requested.
On Keys	Provides information about the application's use of function keys, mnemonics, and keyboard accelerators.
Index	Provides an index, with a search capability, for all help information in the application.
Tutorial	Provides access to the application's tutorial if such a tutorial exists.
On Version	Provides the name, version, and date of the application.

Help windows your application displays generally are normal secondary windows whose parent window is the window from which users requested help. The title in the menu bar is the name of the application (or name of the object in object–oriented systems) followed by the word Help. Help menus may also have a title in the client area that describes the topic for which help is being provided.

How to Design AIXwindows Menus

You will most likely have to design menus specific to your application. The following sections describe some considerations in menu design.

Group Like Menu Selections Together

As a general rule, wherever possible, organize the selections on your menus into logical groups.

However, it's not a good idea to put a destructive command (such as Delete or Quit) next to other frequently chosen selections. This is because of a common problem of menus called the off by one error: Users mistakenly choose the selection next to the one they intended to choose. So be aware of the result if a user mistakenly chooses the menu selection above or below the intended selection.

Use a visual cue such as a separator line to divide menu selections into logical groups.

Sets of related items, such as radio buttons or check buttons, should be located together and separated from other menu items.

List Selections in Order of Frequency

Order menu selections according to the frequency of usage, positioning the most frequently used selections near the top of the menu.

As much as the logical grouping of the menu allows, order the selections of your menu in decreasing frequency of

usage. Within logical groups, the same principle applies: List menu selections according to the frequency of usage with the most frequently used selections at the top.

As you order menus, maintain a global perspective. Consistency across your entire application is generally more important than frequency of use in a particular menu. Evaluate frequency of use over the entire environment faced by people who use your application.

Keep Menu Structures Simple

If your application requires submenus, keep the menu structure simple. While it is possible to create menu structures that start from a single pulldown or popup menu and cascade down several levels of submenus, it is seldom necessary to do so. The awe of users who see submenu after submenu cascading down the screen quickly turns to consternation over such poor design.

Avoid using many submenus in your application. It is better to use a few more pulldown and popup menus, or to have more selections per menu, than to have multiple cascading submenus.

As a general guideline, use as few menu levels as possible with three levels as a maximum. A dialog box is a good alternative to menu complexity. So is redesigning your menu structure to eliminate complexity.

Provide Accelerators and Mnemonics

Keyboard accelerators and mnemonics enable users who have become familiar with your application to take short cuts, increasing their efficiency, while not affecting those users who are still learning the basics.

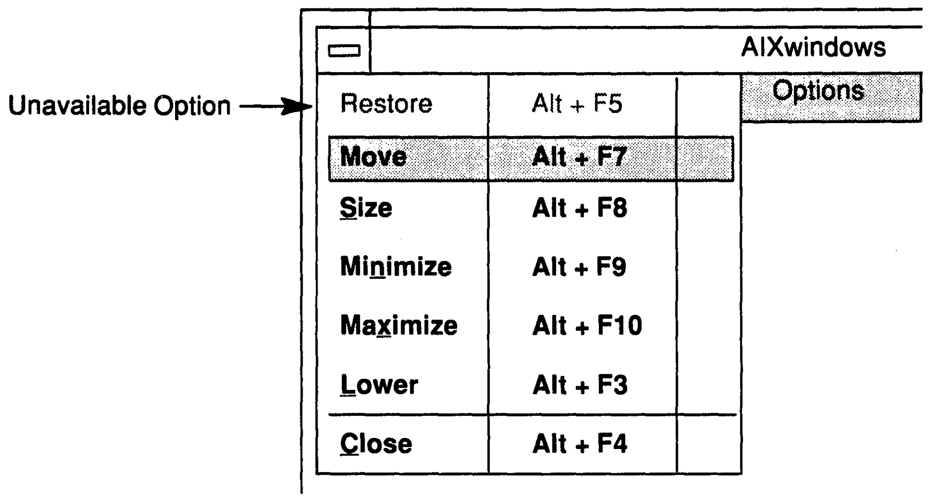
Mnemonics require the display of a menu but are preferred by some users because they allow those users to operate the mouse with one hand and make the selection with the other (as the mouse hand returns to the keyboard). Mnemonics can be made more memorable by carefully choosing the mnemonic letter.

Keyboard accelerator sequences don't require the display of a menu, hence they are active whenever your application's window is active. They are designed for menu entries that people use very frequently. To make the accelerators for your application more memorable, design your application so that accelerator sequences are consistent and progress logically.

Control Availability of Menu Selections

During use of your application, the application will enter a state where some menu selections will not make sense. In that case, make them unavailable while that state occurs. This avoids excessive message boxes noting a selection error.

Making menu selection unavailable provides a temporary constraint. Menu selections that are never applicable to your application should not be included in the menu.



Unavailable menu selections provide a visual cue, such as being dimmed, that they are not functional in the current context. But even if all selections in a menu are disabled, users should still be able to display the menu (but not choose any of the selections) and get help.

Consider Use of Graphic Images

An important option to consider is the use of graphic images (bitmaps) for selections. A good picture can be more readily understood and easily remembered than a line of text. Graphics also ease the task of localizing your application for other countries.

Keep Menu Selections Stable

You should generally keep menu selections the same for the duration of an application's invocation. Settings in menus may be set or unset and any selection may become unavailable, but the selections themselves should not change.

Do not replace menu selections during an application's use. Entries that are temporarily unavailable should be disabled and should visually appear as such.

You may want to reword some menu selections slightly in order to better reflect their meaning during the current state of the application. The Undo entry in the Edit menu does this.

Adding menu entries dynamically is discouraged. If your application does require this, however, add them at the end. Number the selections and use the number as the mnemonic. This is the only case where menu selections should be numbered.

The above discussion of dynamic changes in menus applies only to changes made by the application in response to changing conditions in the application. They do not apply to any changes due to user customization.

Allow Users to Customize Menus

Should you choose to, you can further empower the people using your application by allowing them to create their own menu titles and selections and associate them with their own choice of functions.

Understanding AIXwindows Dialog Boxes

A **dialog box** is a window that contains graphical controls that people use to converse with your application. While the AIXwindows toolkit supplies you with graphical controls for most occasions, you must select the appropriate controls and create the dialog boxes for your application.

The following aspects of dialog boxes are discussed in this overview:

- The characteristics of dialog boxes
- Dialog box actions
- The anatomy of a dialog box
- Message diaglog box types
- Common dialog boxes
- How the user converses with dialog boxes.

The Characteristics of Dialog Boxes

People use dialog boxes to control the operation of your application. From a technical point of view, a dialog box is any window that solicits or displays information or instructions. While dialog boxes are similar to menus, because they enable the user to control your application, they can be much more flexible than menus. It is important that you design dialog boxes with the needs of your application and your user in mind.

The Purpose of Dialog Boxes

Dialog boxes have a variety of purposes. Some display information. These message boxes may be rather plain and relatively simple. Other dialog boxes solicit data from the user. These may include elaborate combinations of text and graphics and a variety of controls (entry boxes, list boxes, radio buttons, etc.) through which the user conveys information to your application.

You are not restricted to these purposes. In fact, many of your dialog boxes may serve a combination of purposes. However, you will probably notice that many of your dialog boxes require the same or similar actions. Dialog Box Actions describes standard dialog actions. Your application should take advantage of these when appropriate.

Ending a Dialog

Many dialog boxes disappear after the user acknowledges the information, provides the information requested, or selects an action. This is another area in which dialog boxes are similar to menus. Unlike menus however, you may want a particular dialog box to remain visible after a selection. This enables the user to continue the dialog without having to redisplay the dialog box. The **Find String** dialog box of some editor programs is an example. It remains visible until the user explicitly closes it.

Making Controls Unavailable

As people use your application, the operational context that develops may make the use of certain controls inappropriate. For example, the use of the Minimize selection in a window menu is inappropriate when the window is already minimized. In such cases, you should make the inappropriate controls unavailable. This is also called disabling the controls. Unavailable controls are visually de–emphasized. The unavailable controls can be in any window of your application. While unavailable controls do not operate, the user should still be able to get help on them.

If a control can never be used, that control should not appear at all. Applications that have various authorization levels, for instance, should only show those controls the user is authorized to use.

Some dialog boxes require the user to complete the interaction with them before the application continues. In essence, these dialog boxes temporarily disable all other controls in the application. This type of dialog box is sometimes called **application modal**. Others take over the entire display and make all other windows on the screen unavailable. These windows are called **system–modal**. Message boxes that require the user to perform some immediate action before processing can continue are examples of modal windows. Use the disabling feature sparingly, since it compromises the basic premise of the AIXwindows interface: empowering the user.

Dialog Box Actions

Most often dialog boxes present information or solicit data. Typically, they also provide several actions from which the user selects the action desired.

While on occasion your application may require special dialog box actions, most of your dialog boxes will share common actions. The AIXwindows user interface has identified actions that are likely to occur in many dialog boxes and has given them common names and definitions. No dialog box will contain all of the common actions listed below. Select the ones appropriate to your application or determine your own actions using the guidelines in this section. The list is ordered in the approximate sequence they appear in dialog boxes.

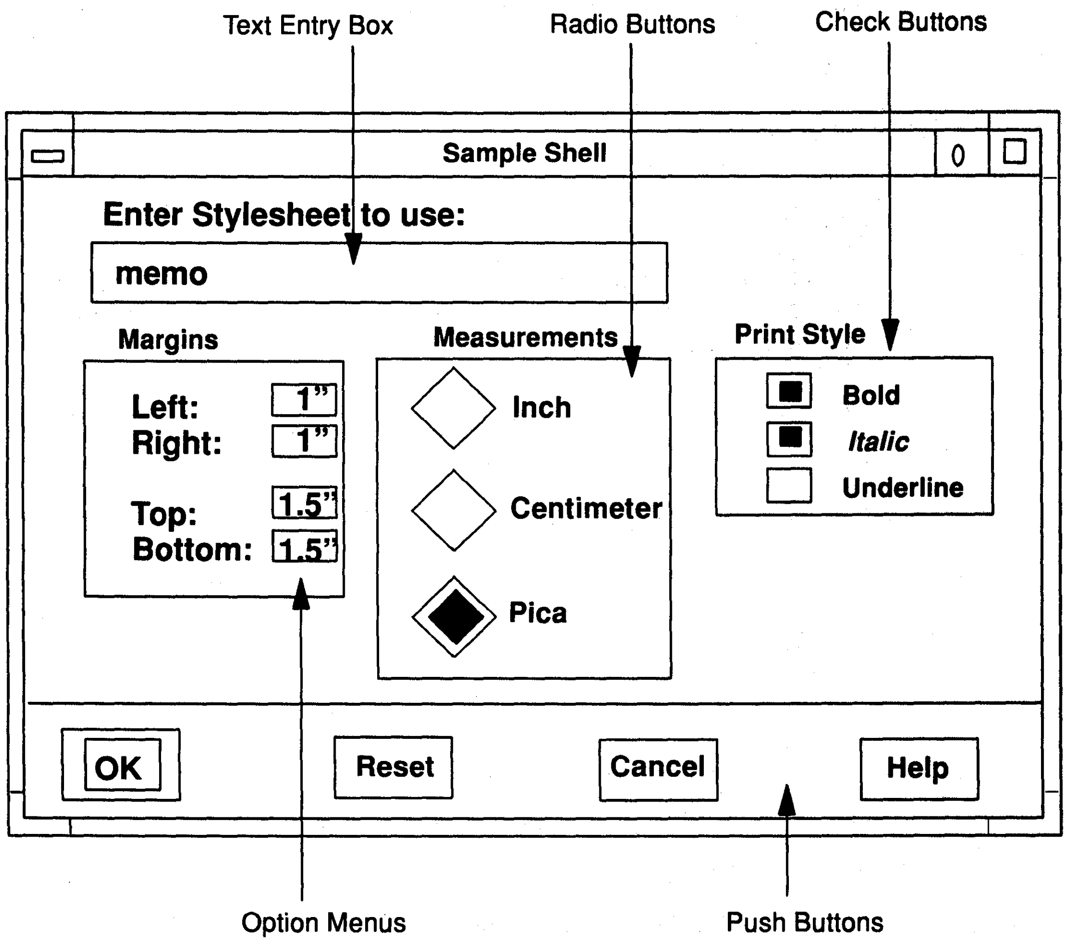
Action	Performs the action specified by the label. You specify actions that are appropriate to your application.
Yes	Indicates an affirmative response to a question posed in the dialog box and closes the window. See note below.
No	Indicates a negative response to a question posed in the dialog box and closes the window. See **Note:** at the end of this list.
OK	Combines **Apply** and **Close** actions (both are described below) in one convenient push button.
Apply	Applies any changes made to controls in the dialog box.
Retry	Causes the task in progress to be attempted again. This action is commonly found in message boxes reporting some sort of hardware error.
Stop	Ends the task in progress at the next possible breaking point. This action is commonly found in progress message boxes.
Text...	Opens another dialog box that extends the current dialog.
Reset	Resets any changes to controls in the dialog box to the values they had when the dialog box was opened.
Cancel	Combines **Reset** and **Close** actions in one convenient pushbutton.
Close	Removes the dialog box. The user explicitly closes dialog boxes that remain open after each use. This function duplicates the **Close** action in the window menu of the dialog box. You are encouraged to include it in your dialog box in case the user runs your AIXwindows application without the AIXwindows window manager.
Help	Enters the help subsystem.

Note: While **Yes** and **No** are not actions, they imply do and do not and are used in dialog boxes in that context. However, you should be careful to avoid ambiguity. Only if a question is very simple, phrased without negatives, and results in an action that is not damaging should you use **Yes** and **No**. Otherwise, use an action.

The Anatomy of a Dialog Box

Dialog boxes are composed of combinations of the controls described in AIXwindows Application Graphic Controls Overview.

Dialog boxes differ from one another, not because their controls differ. Controls do not differ in behavior. A push button, for example, is always pushed; a check button is always checked. This is fundamental to common behavior. Rather, dialog boxes differ because your choice of controls and combinations of controls differs depending on the needs of your application.



While the AIXwindows Toolkit provides a variety of standard controls, you can create additional controls should the need arise. A stepper button is an example of an application supplied control.

Arranging Push Buttons in Dialog Boxes

You should arrange push buttons in your dialog box in in an order that supports the user's natural progress through the controls in the dialog box. This guideline usually results in pushbuttons either in a row at the bottom of the window or in a column at the right .

Of the two positions, in a row along the bottom of the dialog box is preferable because push buttons are frequently used to end the dialog and thus are the last thing the user encounters as the contents of the dialog box are scanned.

Push buttons are commonly found in the following combinations and sequences:

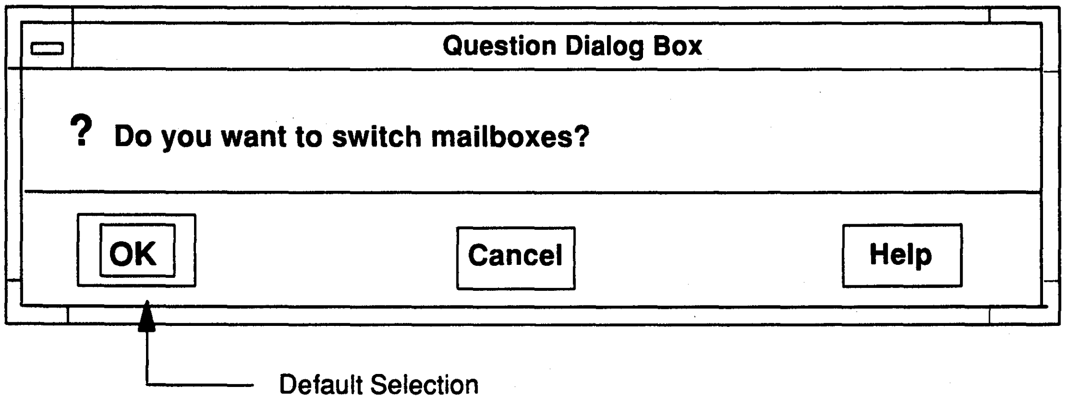
- Close
- OK Cancel Help
- OK Reset Cancel Help
- Apply Reset Close Help
- Yes No Help
- Retry Stop Help

In this order, the positive (confirming) selections are toward the left, the selections negating change are toward the right, and Help is consistently placed as the right most push button. If you choose to arrange push buttons in a column, the positive selections should be toward the top with Help on the bottom.

Default Push Buttons

A **default push button** enables the user to easily select the most likely response to a dialog box query. The default push button is always visually distinguishable from other push button selections. The OK push button is frequently the default push button in dialog boxes.

If possible, the action performed by the default push button should be reversible. If an action is potentially destructive (for example, if loss of data could occur), its button should not be made the default push button.



The user can start the default push button action in either of two ways:

- By double–clicking when making a selection in the dialog box.
- By pressing the Enter key after making a selection in the dialog box.

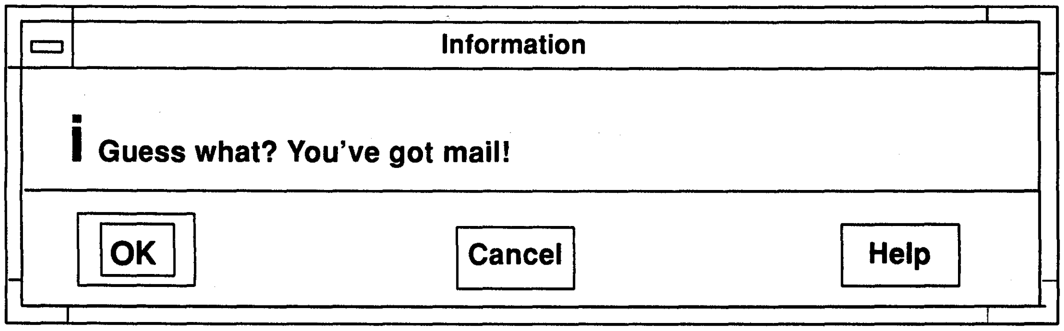
When people use the keyboard to navigate through the push buttons, the button with the location cursor always becomes the default push button. This ensures that pressing the Enter key over a push button invokes that push button. When the location cursor leaves the push buttons, the original default button once again becomes the default.

Message Dialog Box Types

Some dialog boxes provide the user with a message and ask for an acknowledgement or response or some sort. These dialog boxes, called message boxes, are not requested by the user, but are displayed by an application as a result of some event. The dialog boxes can be divided into five general types depending on the nature of their message: information, progress, question, warning, and action.

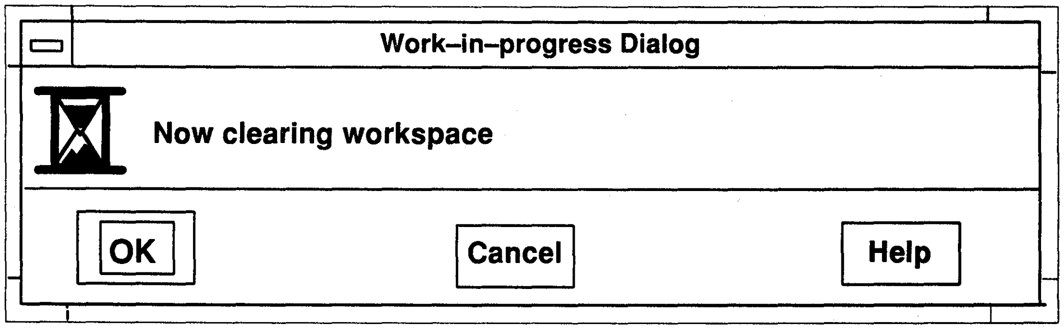
Information

Some dialog boxes simply convey information. They inform the user. The information dialog box does not interrupt any tasks. For example, they may display the fact that a user has newly arrived electronic mail. When the user acknowledges the information, the information box goes away. A typical information dialog box that conveys information is illustrated.



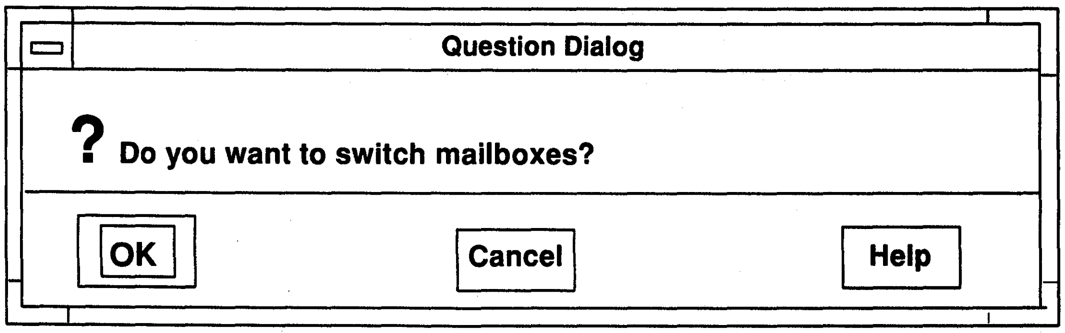
Progress

Other dialog boxes convey current progress. The message displayed by a work–in–progress dialog box tells the user of the progress of an operation and allows the user to choose between continuing the operation or canceling it. A typical work–in–progress dialog box that conveys the status of work in progress is illustrated. You may want to show in your application's progress box how much of the total work has been completed.



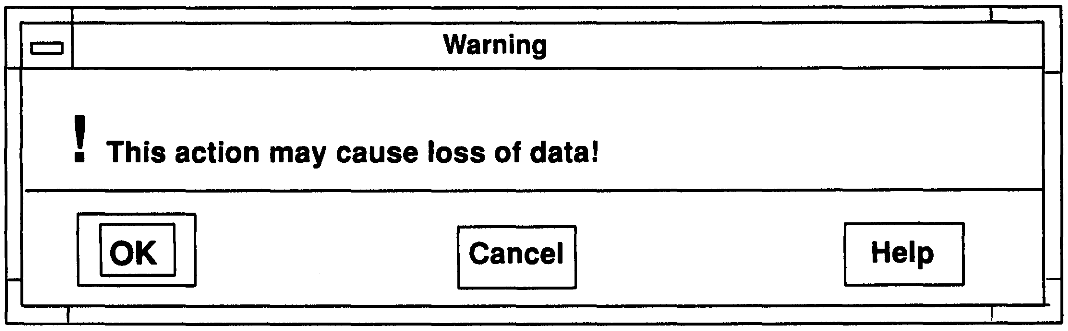
Question

Some dialog boxes clarify a previous response by asking a question. The work does not proceed until the question has been answered. The question briefly explains the situation and is phrased so that the reply is a choice between mutually exclusive alternatives. A typical question dialog box that asks a question is illustrated.



Warning

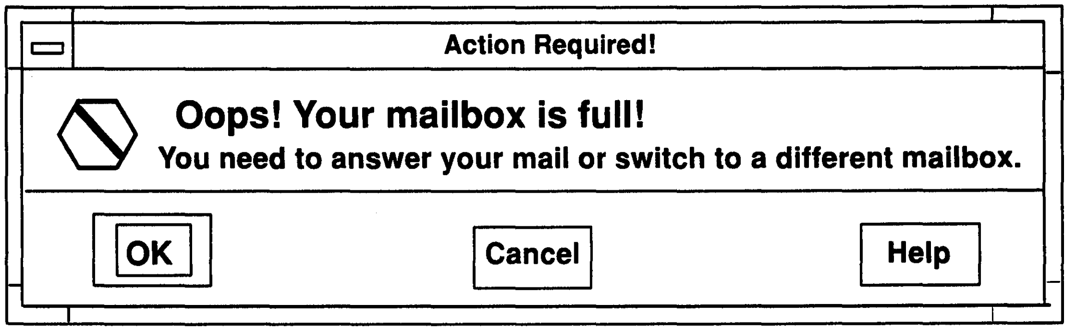
Other dialog boxes convey a warning. The warning dialog box alert the user to some eminent danger (for example, the potential loss of data) and allow the user to choose between ignoring the warning and continuing the operation or heeding the warning and canceling the operation. A typical warning dialog box that warns of possible danger is illustrated.



This action may cause loss of data!

Action

Still other dialog boxes convey a message that requires immediate action. They alert the user that some action is required (or that some condition exists that requires action) by constraining the operation of the application until the action has been completed. A typical action dialog box that conveys an action message is illustrated.



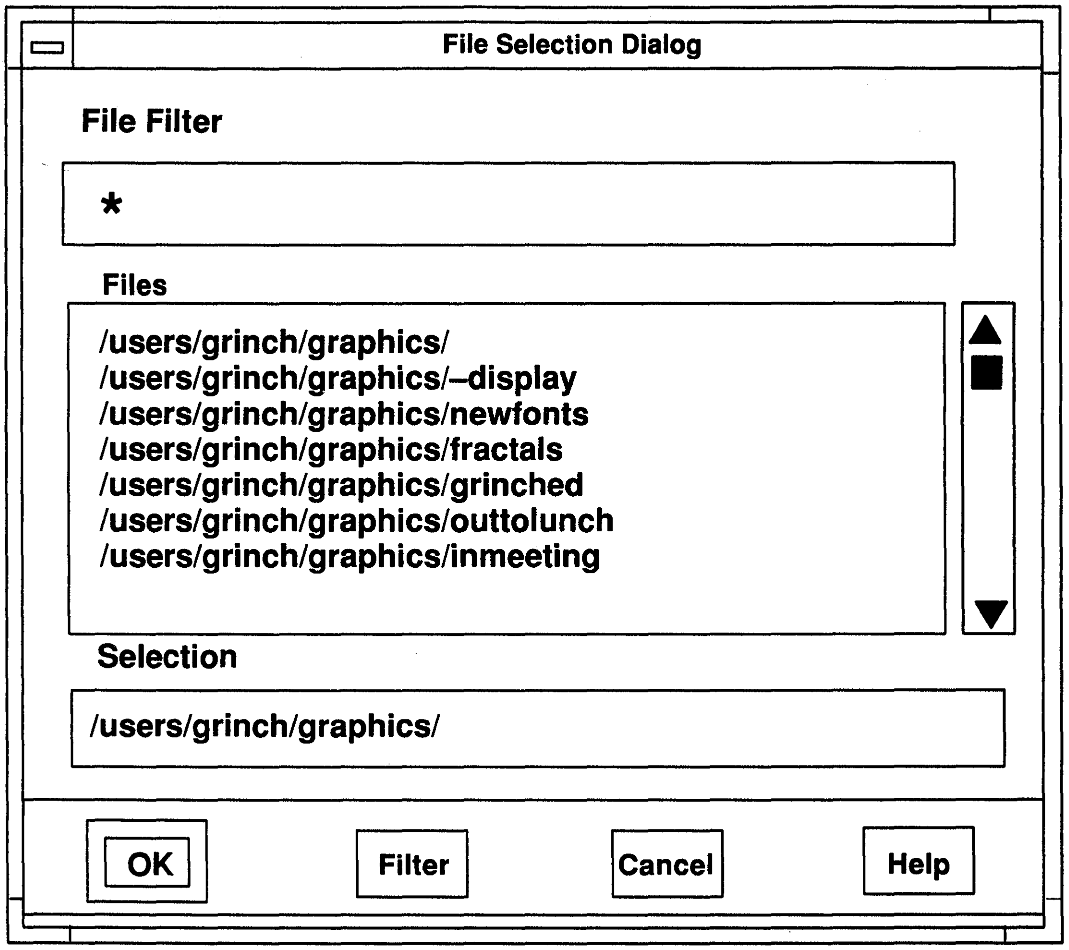
Oops! Your mailbox is full!
You need to answer your mail or switch to a different mailbox.

Common Dialog Boxes

Some dialog boxes perform an operation based on supplied input. Usually, these dialog boxes are application specific, but some operations are common to a wide variety of applications. Examples include operations to save a file, enter a command, and make a selection.

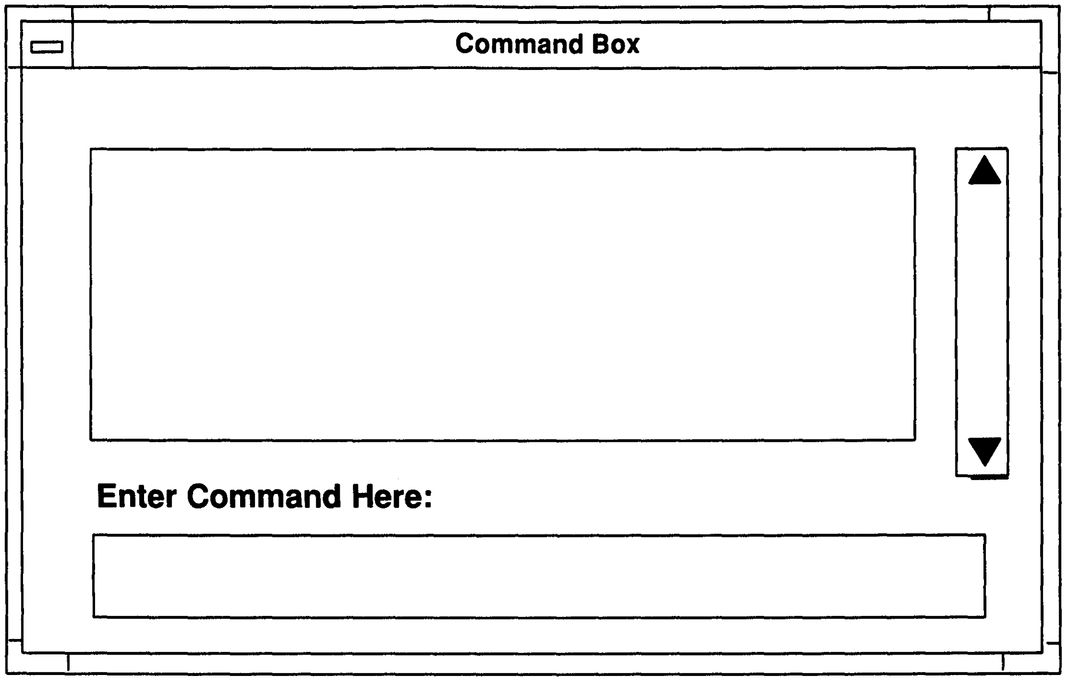
File Selection

The File Selection dialog box is used by the user to enter the name of a file to save. A sample File Selection dialog box is illustrated. The box is most commonly used in conjunction with the common File menu described in Understanding AIXwindows Menus.


File Selection Dialog
File Filter
*
Files
/users/grinch/graphics/
/users/grinch/graphics/–display
/users/grinch/graphics/newfonts
/users/grinch/graphics/fractals
/users/grinch/graphics/grinched
/users/grinch/graphics/outtolunch
/users/grinch/graphics/inmeeting
Selection
/users/grinch/graphics/
OK
Filter
Cancel
Help

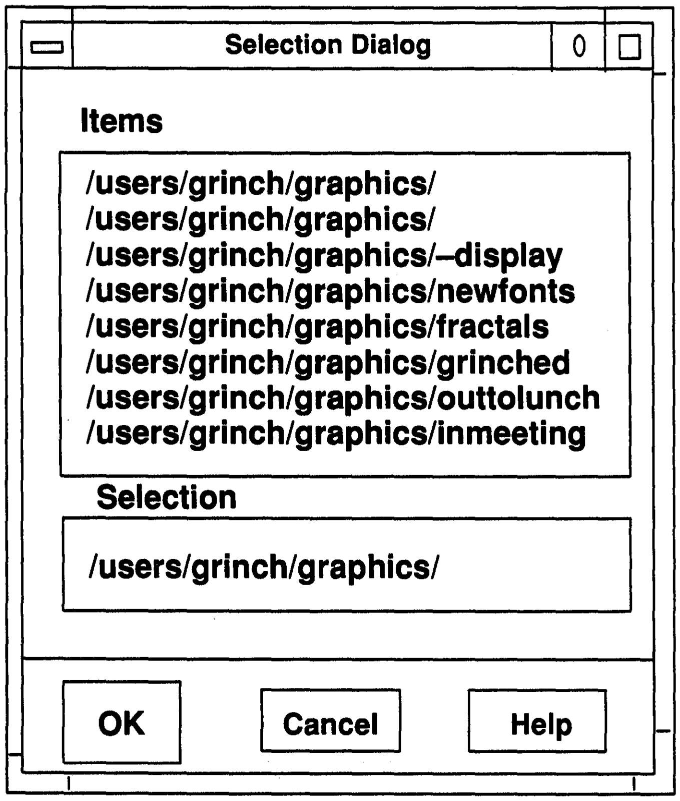
Command

The Command dialog box is used by people to enter a command either to the application or to the computer operating system. The Command dialog box provides the same function as a command line prompt and provides a history list from which previous commands can be reselected. Your application should use either the command line or the command dialog box. A typical command dialog box is illustrated.



Selection

The Selection dialog box is used by people to choose from a list of several possible selections. A typical Selection dialog box is illustrated.


Selection Dialog
Items
/users/grinch/graphics/
/users/grinch/graphics/
/users/grinch/graphics/–display
/users/grinch/graphics/newfonts
/users/grinch/graphics/fractals
/users/grinch/graphics/grinched
/users/grinch/graphics/outtolunch
/users/grinch/graphics/inmeeting
Selection
/users/grinch/graphics/
OK
Cancel
Help

Starting a Dialog

In general, dialog boxes are started by the user choosing selections from controls such as menus. Menu selections that lead to dialog boxes follow the selection name with ellipsis (...) as a visual cue.

In the case of message boxes, the opposite is true. The application starts the dialog box based on some event. In the case of information and progress boxes, work progresses unimpeded by the presence of the dialog box. For the others, work on a task cannot proceed until a question has been answered, a warning of potential damage acknowledged, or some specific action taken.

Navigating Through a Dialog Box

In many cases, the user does not need to have a long–winded conversation with a dialog box to produce results. Rather, the user calls the dialog box from the menu, makes a few changes to some settings, and sends the dialog box away. Availability of an appropriate default action is key to ease of use.

Your application should provide the user with the ability to carry on a selective conversation with a dialog box and to easily navigate through the box. The mouse user slides the pointer directly to a control. The keyboard user presses the Tab and Arrow keys to step through a dialog box.

Determining Dialog Box Location and Size

Your application determines the size and location of its dialog boxes.

Location

Whenever possible, do not place a dialog box so that it obscures important information in the underlying window. A dialog box should be horizontally offset from the underlying window. This enables the user to see the chosen selections.

When a dialog box relates to an item in an underlying window, position the dialog box next to the item. As much as possible, avoid covering information in the underlying window that the user might refer to in conversing with the dialog box.

If it is necessary to display two dialog boxes, offset the top dialog box (the one that was called last) to the right and below the title of the underlying dialog box (the one that was called first).

Obviously, there is only a limited amount of screen real estate in which to place a dialog box. While the above suggestions seem simple enough, they can not always be followed completely. Therefore, dialog boxes, once displayed, should be movable so that the user can relocate them as needed to see information in underlying windows.

Size

The initial size of a dialog box should be large enough to contain the dialog controls without crowding or visual confusion, but otherwise as small as possible.

Understanding AIXwindows User Help Design

Good applications provide help facilities for the people to use when the need arises. No matter how intuitive you design your application to be, there will be times when some users get stuck. Paper documentation is important, but on– line help is often more readily available (manuals can easily get lost) and more appropriate. This is particularly true of context–sensitive help.

Users can call for help using the **Help** menu from the menu bar as described in Chapter 6. Since dialog boxes usually don't have menu bars, they have a **Help** push button which opens a help window. Help windows in turn have an action bar which contains the **Help** menu.

Types of Help

AIXwindows provides for several types of help information. Each corresponds to an entry in the **Help** menu. If your application does not support all forms of help, be sure that your **Help** menus only include the forms you do support.

Help On Context

Context–sensitive help may be the most important type of help information for people using your application. Context–sensitive help describes the nature of a specific control and how people use that control.

The help message is unique to the field. Context–sensitive help for a field specifying printer baud rate may be:

```
Baud Rate is the speed at which your printer can accept data.
Acceptable speeds are 1200, 2400, and 9600. Enter the speed in bits
per second.
```

Users request context–sensitive help from the keyboard by moving the selection cursor to the desired field. Pressing the **Help** key will then display help for that field.

Using the mouse, people can use the **On Context** entry in the **Help** menu. The pointer shape changes to the help pointer shape. Selecting a field with the help pointer displays help on that field. After the selection, the help pointer returns to its previous shape. To cancel the context–sensitive help mode, users press the **Esc** key.

Typing the Help key while the Select button is pressed on a field will also display context help for that field.

Help On Window

This help provides information about the current window. It describes the function of the window and the tasks that can be performed using that window.

Users request help on the window via the **Help** menu.

Help On Keys

Applications frequently have special uses for keys, particularly function keys. Help on keys shows the meanings assigned by this application to special keys.

Users request help on keys via the **Help** menu.

Help Index

Users often have a certain topic in mind when they request help. This option lists alphabetically a series of topics on which help is available. Users select the topics of interest and help on those topics is then displayed in the window. Applications may provide an entry field to allow users to type in a topic of interest.

Help on Help

This option provides information on how to use the help facility.

Help Tutorial

Complex applications may provide on–line help tutorials. This option provides access to such a capability.

Help On Version

This option provides information about the specific version of the application. Information typically includes the name, version, and date of the application as well as copyright data.

Help Windows

Help windows are standard application secondary windows. They contain an action with at least the **Help** menu. Their title is the title of the window from which they were requested followed by the word **Help**. They may also have a title in the client area to specify which field in the window the help window describes. Place help windows so that they do not obscure the objects they are describing.

Understanding International Implementation for AIXwindows Application Design

Designing software for the international market requires producing applications that are easily translatable. In a typical scenario, when an application is translated into another language, some portion of it remains constant, while other portions change.

The constant portion forms the **base** of the product. The base contains all parts that are internationally invariable. The portion that changes from country to country is the **locali–zation** portion. Identifying and separating the base from the localization portion is important in creating applications that are efficient to translate.

This chapter discusses the following localization issues:

- Collating sequences.
- Country–specific data formats.
- Icons and pointer shapes.
- Scanning direction.
- Text translation.
- Messages.

Collating Sequences

To produce an alphanumeric list, printable characters are sorted according to a **collating sequence**. Printable characters include letters, numbers, punctuation characters, and other symbols such as asterisks (*) and ampersands (&). The collating sequence defines the value and position of a character relative to the other characters.

Many applications make frequent use of collating sequences to produce alphanumeric lists. Examples of alphanumeric lists include the following:

- A directory listing of file names.
- The output from a sorting utility.
- An index produced by a text–processing application.
- The lists produced by a database application, such as lists of names or addresses.

If your application contains sorting functions, it needs to be flexible enough to accommodate individual countryspecific collating sequences. Collating sequences must not be hard–coded into your application. At run–time, your application should refer to a table holding the collating sequences.

Country–Specific Data Formats

The formats for many types of data vary from country to country. When designing data formats, use the same format for input and display. If the format changes, it should change for both. Do not make formats dependent upon other features of your application or dependent upon each other.

Country–specific data includes the following:

- Thousands separators.
- Decimal separators.
- Positive and negative values.
- Currency.
- Dates.
- Time formats.
- Time zones.
- Telephone numbers.

Thousands Separators

For separating units of thousands, the comma, period, space, and apostrophe are considered to be valid separators. Your application should allow for arbitrary separators or no separators at all. The thousands separator should be definable by people who use your application or at the time of localization.

Decimal Separators

For separating decimal fractions, the period, comma, and the center dot are considered to be valid separators. Do not use the same character for both the thousands separator and the decimal separator. The decimal separator should be

definable by people who use your application or at the time of localization.

Positive and Negative Values

The plus (+) and minus (–) signs are valid either before or after a number. In applications such as a spreadsheet, allow negative numbers to be enclosed in parentheses.

Currency

For currency, the comma, period, and colon are valid separators. They should be modified independently of the separators used for other data formats.

The currency symbol is a valid separator. It can be placed before or after the numerical value, or be used as the decimal separator. Allow for one or no space between the currency symbol and the amount. The currency symbol can be changed.

Dates

If your application displays or prints the date, the date should be in the format and language of the people using your application. Properly formatting the date requires the following:

- Alphanumeric date formats should allow sufficient storage and display space to accommodate date names in other languages.
- The position of each component of an alphanumeric date should be able to be varied or omitted.
- The hyphen, comma, period, space, and slash are all valid separators for the day, month, and year.

- Numeric date formats should allow the month and day fields to be reversed. For example, the 4th of August, 1990, can be displayed as either 4/8/90 or 8/4/90.
- The format for entering the date should match the display format.

Time Formats

For time formats, hour, minute, and second components always appear in that order. Hours and seconds are never used without minutes.

The colon, period, and space are valid separators for hours, minutes, and seconds. In 24–hour notation, the four–digit format may use a separator.

Time Zones

For time zone displays, allow up to three characters. Also, allow for the time offsets to be fractions, because time offsets are not always an integer number of hours from Greenwich Mean Time (GMT).

Telephone Numbers

Telephone numbers differ in the number of digits and the format used. The space, hyphen, period, comma, and square brackets are all valid separators.

For international numbers, the plus sign is frequently used in Europe to indicate that a number is a country code. For national numbers, it is common (but not universal) to put the city code or area code in parentheses.

Icons and Pointer Shapes

It may not always be possible to design an icon, pointer shape, or other graphical symbol that adequately represents the same object or function in different countries. Culture is inherent even in seemingly universal symbols. For example, sending and receiving mail is a commonly understood function, but representing that function with an icon of a mail box may be inappropriate because the appearance of mail boxes varies widely among countries. An envelope may therefore be a more appropriate icon.

When used correctly, graphical symbols offer the following advantages:

- They are language independent and do not need to be translated.
- They can be used instead of computer terms that have no national–language equivalent.
- They may have more impact when used with text as warnings than the text alone.

Scanning Direction

Western languages scan left to right across the page (or display screen) and from top to bottom. In Eastern languages, however, this is not the case. The scanning direction of the country of localization may have an impact on the location of controls in dialog boxes, the order of selections in menus, and other areas.

Translating Screen Text

Well–written screen text makes an application easier for users to understand. It also makes translation easier.

In particular, relationships between nouns become very explicit in other languages. You should avoid stringing three or more nouns together. Use prepositions to clarify the relationship of the nouns.

Also, text translated from English is likely to expand 30% to 50%. If text space is limited, for example an error message restricted to one screen line, shorten the original version to allow for the translation.

While common words are easy to understand and translate, jargon, when translated into other languages, remains jargon and serves only to confuse and intimidate people who use your application. Avoid using jargon whenever the jargon is not a part of the working vocabulary of the people using your application.

Use the following guidelines to write screen text for translation:

- Brief and simple sentences are easy to understand and aid in translating.
- Affirmative statements are easier to understand than negative statements.
- Active voice is easier for both application users and application translators to understand.

Messages

Languages have different grammatical structures that must be accommodated when a product is adapted for a local environment.

Do not build messages dynamically; that is, do not store separate subsections of a message and assemble them for output. This will avoid embarrassing juxtapositions of words and phrases. Store each message as a complete string of variable length in separate modules or files. Do not embed messages in your code.

AIXwindows Application Default Keyboard and Mouse Button Bindings

Different keyboards may have different numbers of keys or different key labels. Similarly, different mice have different numbers of buttons. Nevertheless, the AIXwindows environment is able to employ a standard of consistent behavior because it prescribes that the same function always be invoked in the same way.

In particular, common and frequently performed functions are associated with (bound to) particular sequences of keyboard key presses and mouse button operations. These default keyboard and mouse actions, and the functions they invoke, are presented in this appendix in tabular form.

Default Keyboard Assignments

Modifier Keys

The typical keyboard also has the following **modifier** keys:

- Shift
- Alt or Meta
- Ctrl

Users use these keys to modify the meanings of other keys or mouse button operations. Modifier keys are used in conjunction with other keys or mouse actions. Accelerators are an example of modifier usage.

Keyboard Function Assignments

The following table includes navigation, function, and special–purpose keys and their associated functions.

Key	No Modifier	Shift	Control	Alt
Left Arrow	Left	Range selection	Word left	
Right Arrow	Right	Range selection	Word right	
Down Arrow	Down	Range selection		Lower window to bottom of stack.*
Up Arrow	Up	Range selection		Raise window to top of stack.*
Delete	Delete	Cut	Erase to end of line	
End	End of line		End of data	
Esc	Cancel	Window menu		Next Application
F1	Help			
F2				
F3				Lower*
F4	Prompt/Popup menu**			Close window*
F5				Restore window*
F6	Switch window panes			Switch windows*
F9				Minimize window*
F10	Switch to menu bar			Maximize window*
Home	Beginning of line		Beginning of data	
Page Down	Page down		Page right	
Page Up	Page up		Page left	
Spacebar	Select**			Window Menu***
Tab				Next application***

Notes:

* An optional assignment and should remain available for user convenience.

** Used in substitutions when special purpose key is not available.

*** Provides compatibility with MS–Windows.

Default Keyboard Assignments for Window Management Functions

The following table contains the default key assignments used by the AIXwindows window manager for the window menu mnemonics and accelerators:

Menu Selection and Mnemonic	**Accelerator**
Restore R	Alt + F5
Move M	Alt + F7
Size S	Alt + F8
Minimize n	Alt + F9
Maximize x	Alt + F10
Lower L	Alt + F3
Close C	Alt + F4

Note: Alt+F5 means to hold down the Alt key and press F5.

Default Key Assignments for Text Keys

The following table contains the default key assignments used by text entry boxes and other editing windows:

Key	Function			
	No Modifier	Shift	Control	Alt
Backspace	Delete to left			Undo last action
Ins	Insert/Replace	Paste	Copy	
Del	Delete	Cut		

Default Key Assignments for Cursor Navigation

The following table contains the default key assignments used by the AIXwindows window manager for cursor navigation:

Key	**Description**
Next Field	Moves the cursor forward to the next field going from left to right and top to bottom. At the last field, the cursor wraps to the first field.
Previous Field	Moves the cursor back to the previous field going from right to left and bottom to top. At the first field, the cursor wraps to the last field.
Home	Moves the cursor to the beginning of the current line.
Ctrl+Home	Moves the cursor to the beginning of the data.
End	Moves the cursor to the end of the current line.
Ctrl+End	Moves the cursor to the end of the data
Up arrow	Moves the cursor to the previous choice in the client area. At the first choice, the cursor wraps to the last choice.
Down arrow	Moves the cursor to the next choice in the client area. At the last choice, the cursor wraps to the first choice.
Right arrow	Moves the cursor to the next choice. At the last choice, the cursor wraps to the first choice on the line below. At the bottom last choice, the cursor wraps to the top firs choice.

Left arrow	Moves the cursor to the previous choice. At the first choice, the cursor wraps to the last choice on the line above. At the top first choice, the cursor wraps to the bottom last choice.
Page Up	Scrolls a window, displaying the previous page of information.
Page Down	Scrolls a window, displaying the next page of information.
Ctrl+Page Up	Scrolls a window, displaying information previously out of view to the left.
Ctrl+Page Down	Scrolls a window, displaying information previously out of view to the right.
F6	Moves the cursor from one pane to another in a split window. Precedence is left to right, top to bottom.
F10	Moves the cursor to the menu bar.
Ctrl+Left Arrow	Moves the cursor to the previous word in a field.
Ctrl+Right Arrow	Moves the cursor to the next word in a field.

Substitutions for Special Purpose Keys

Not every keyboard contains all the special purpose keys mentioned in this style guide. Where a special–purpose key is available, it should be used as described. Thus keyboards with a Select key use it for selection. Some keyboards may not include all special–purpose keys. The following table presents some suggested substitutions.

Key	Substitution
Select	Spacebar or Enter
Menu	F4
Help	F1
Alt	Extend
Esc	F12
Enter	Return
Next Field	Tab or Ctrl+Tab
Previous Field	Shift Tab or Ctr+Shift+Tab

If neither a special–purpose key nor a recommended substitution is available, function key equivalents should be used. Applications that do not need a function key for one of the standard functions may use that function key for application–specific purposes.

Operating Scroll Bars with a Mouse

Clicking the Select button with the pointer on stepping arrows.

Highlights the stepper arrow and moves the windowthrough the underlying file by a single unit, in thedirection indicated by the arrow. You must determine the appropriate unit to be scrolled in your application. For example, a unit in a spreadsheet may be a single row or column.

Pressing the Select button with the pointer on stepper arrows.

Highlights the stepper arrow and causes a continuous scroll, in unit steps, in the direction indicated by the arrow. The default step speed is approximately 30 milliseconds and is timed by the completion of the scroll operation.

Clicking the Select button with the pointer in the scroll region.

Moves the window through the underlying file by one window length minus one unit for overlap. Clicking below (or to the right of) the slider advances to the next window's worth of information. Clicking above (or to the left of) the slider moves back to the previous window's worth of information.

Pressing the Select button with the pointer in the scroll region.

Continuously moves the window through the underlying file by one window length minus one unit for overlap until the user releases the select button or until the slider moves under the pointer.

Dragging theslider with the Select button pressed.

Moves an outline of the slider. Releasing the select button repositions the window to a location consistent with the new slider location. The user can release the select button when the pointer is anywhere in the window, not just within the scroll region. This action can be canceled by clicking any of the other buttons before releasing the select button.

Object Selection

Selection Task	**Mouse Selection Operation**	**Keyboard Selection Operation**
Change selection focus.	Move the mouse to position the pointer	Press the navigation keys to position the selection cursor.
Select a single object.	Click Select	Press Select
Select a range of objects.	Drag Select	Press Select+navigation keys
Select all objects between current location and previous location.	Click Shift+Select	Press Shift+Select
Select an additional object.	Click Ctrl+selection operation	Press Ctrl+selection operation.
Deselect an object.	Ctrl+selection operation on a selected object.	Ctrl+selection operation on a selected object.

Incremental Text Selection

In addition to text selection using the press–and–drag method, your application can use an incremental selection method. In this method, the user can perform multiple clicks of the select button to select successively larger (or smaller) portions of text.

Number of Select Button Clicks	**Text Block Selected**
1 click	Null selection
2 clicks	Word
3 clicks	Line
4 clicks	Paragraph
5 clicks	Module

Chapter 5. AIXwindows User Interface Language (UIL) and Resource Manager (MRM) Overview for Programming

The AIXwindows User Interface Language (UIL) is a compiled specification language for describing the initial state of a user interface.

The AIXwindows Resource Manager (MRM) consists of library subroutines that access UIL at run time and create a user interface.

UIL and its associated MRM are described in greater detail in the following articles:

- Understanding UIL and its MRM
  - Using UIL and its MRM as Interface Programming Tools
  - Understanding the UIL Compiler
  - Benefits of Using UIL and its MRM
  - Productivity Features Supported by UIL

Understanding UIL and its MRM

UIL specifies the AIXwindows widgets, gadgets, and compound objects (objects created from widgets and gadgets) that make up the interface. It also identifies the subroutines to be called whenever the interface changes state as a result of user interaction. The AIXwindows Resource Manager (MRM) creates widgets based on definitions it obtains from one or more AIXwindows User Interface Definition (UID) files generated by the UIL Compiler.

Using UIL and its MRM as Interface Programming Tools

Creating a user interface with the AIXwindows UIL and its MRM involves the following programming tasks:

- Specifying the user interface in a UIL Module
- Compiling a UIL Specification File
- Writing MRM run-time subroutines into your application

Specifying the User Interface in a UIL Module

You use UIL to create a UIL Module that specifies the following aspects of the user interface:

- Objects

The widgets, gadgets, and compound objects from which your interface is built

- Resources

The graphical and behavioral attributes that affect the collective look and feel of the objects in your interface

- Callbacks

The subroutines called when the interface changes state due to user interaction

- Widget Tree

The specific object hierarchy for your application

– Literal Values

The literal values assigned to various object parameters. These values are fetched by the application at run time, often from specific AIXwindows resource files in previously-identified directories.

Note: The literal values stored in AIXwindows resource files can be changed without recompilation of the application or its interface. This means that a user can customize the appearance and behavior of the objects within a specific AIXwindows interface in many important ways without recompiling any code.

Compiling the UIL Specification File

When a UIL Specification File is compiled, it generates a User Interface Definition (UID) file. The compiled UID file is a separate file from the compiled executable code of the application. This separation of files permits work to proceed independently on the application and its interface.

AIXwindows User Interface Language (UIL) Syntax Overview for Programming contains additional information about the syntax rules concerning the high-level constructs and low-level constructs of the language.

How to Create a User Interface with AIXwindows User Interface Language (UIL) and Resource Manager (MRM) contains an overview of the process of creating an AIXwindows interface with UIL and MRM.

Using Recommended Coding Techniques in AIXwindows User Interface Language (UIL) contains additional information about UIL coding techniques and their importance.

How to Create UIL Specification Files contains detailed information about UIL Specification Files.

How to Compile an AIXwindows User Interface Language (UIL) Specification File to Generate a User Interface Definition (UID) File contains additional information about the UIL Compiler.

Understanding the Structure of a User Interface Language (UIL) Module contains additional information about the structure of a UIL Module.

Using Icons in the AIXwindows User Interface Language (UIL) contains additional information about the use of icons in UIL.

Using the AIXwindows User Interface Language (UIL) Callable Interface Subroutine and the Symbol Table Dump Subroutine contains detailed information about invoking these two related subroutines from within an application.

Writing MRM Run-time Subroutines Into Your Application

You must write MRM run-time subroutines in your application to permit the MRM to open the UID file and access the interface definitions it finds there. MRM conveniently builds parameter lists and calls widget creation subroutines for you, saving programming effort and reducing coding errors.

Understanding the UIL Compiler

The UIL Compiler accepts input lines up to 132 characters in length. It has built-in tables containing information about every class of object in the AIXwindows Toolkit. This information includes the following data:

- All widgets and gadgets that are valid children of each object
- All valid widget parameters associated with each object

- All valid callback reasons associated with each object

The UIL Compiler uses this information to check the validity of your interface specification at compilation time, a process that reduces run-time errors.

Benefits of Using UIL and its MRM

Using UIL and MRM to create an AIXwindows interface for an application provides the following benefits:

- Reduced Coding Time
- Earlier Error Detection
- Separation of Form and Function
- Rapid Prototype Development
- Interface Customization without Recompliation

Reduced Coding Time

You can specify an interface more quickly in UIL than in C because UIL does not require the use of specific widget creation subroutines or information about the format of their parameter lists. Only those object parameters subject to change must be included in the UIL source code, and they can usually be specified in any order.

UIL is a specification language that describes the characteristics of an interface; it has no need for control flow. You can define objects in UIL in approximately the same order the objects are presented in the widget tree for your interface. This positional flexibility makes it easier for those reading your UIL specification to interpret the design of the interface.

At run time, when interface objects are created, MRM simplifies your programming tasks by performing some AIXwindows Toolkit subroutine calls.

Earlier Error Detection

The UIL Compiler does type checking that is not available through AIXwindows or Enhanced X-Windows. This means that interfaces specified with UIL typically have fewer errors to debug.

The UIL Compiler displays an appropriate diagnostic message if it detects any of the following errors in your UIL specification:

- The wrong type of value for a parameter
- An object parameter that is not supported by that object
- A reason for an object that the object does not support
- A child of a parent object that the parent object does not support

Separation of Form and Function

UIL allows you to define an interface in a separate UIL Module instead of writing AIXwindows creation subroutines directly into the application source code. This approach separates the form of an interface from the function performed by its associated application.

Separating form and function permits you to do the following things:

- Change the appearance of an interface—for example, by repositioning widgets or changing their borders or colors—without recompiling its associated application.

- Design multiple interfaces that share a common set of functions. For example, you could build a single international application with several different interfaces—one interface for each group of users who speak a different language.

Rapid Prototype Development

UIL helps develop prototypes of user interfaces more quickly than they could be developed with the AIXwindows Toolkit alone. This means that you can create several different interface prototypes in a relatively short time. Designers can then work with end users to evaluate the general "look and feel" of each prototype while development work continues on the application code. Because interface evaluation can proceed independently, the completed UIL interface and its application can be delivered in less time than if the interface code were contained within the application source code in the traditional manner.

Interface Customization

You can customize an interface by putting in place a hierarchy of UID files, each of which contains a set of literal values assigned to various object parameters. This hierarchy is called a UID hierarchy, and at run time the MRM searches it (in any sequence you specify) to build appropriate parameter lists for the AIXwindows widget creation subroutines.

This feature provides a convenient way of providing an interface that supports, for example, several different IBM-supported languages. The text displayed on title bars, menus, and other objects in the interface can be displayed in any of these languages the user chooses without alteration of the application code. In this case, each file in the UID hierarchy represents an alternative interface.

Another use of the UID hierarchy is to isolate individual, department, and division-level customizations of a given interface. In this case, the definitions in the first file listed in the hierarchical array provided to the MRM **MrmOpenHierarchy** subroutine takes precedence over all others, providing definitions for use by the interface at run-time.

Productivity Features Supported by UIL

The AIXwindows UIL supports the following productivity features:

- Named Values
- Compile-Time Expressions
- Identifiers
- Support for Compound Strings
- Include Files for Useful Constants
- Named Lists

Named Values

Rather than specifying object resource values directly in your source code, UIL supports the specification of named values. You declare and use a named value much the same way you declare and use a new variable in a programming language: you assign a literal value (such as an integer or string) a name, then use the name in place of the value throughout the remainder of the code.

Just as variables make application source code easier for others to understand and maintain, using named values makes your UIL specification easier for others to decipher. Using named values also isolates changes and makes it easier to use MRM subroutines to fetch object-related information from the UID file for use at run time.

Compile-Time Expressions

You can use expressions to specify values in UIL. A valid UIL expression can contain any of the following values:

- Integers
- Strings
- Floating-point numbers
- Boolean values
- Named values
- Operators

Expressions can make values more descriptive ("bulletin_board_width/2" is easier to understand than a simple integer) and can help you avoid recomputing values that must be changed (for example, if you needed to change the width of an **XmBulletinBoard** widget).

Identifiers

Identifiers provide a mechanism for referencing values in UIL that are provided by the application at run time. In the application program, you use an MRM subroutine to associate a value with the identifier name. Unlike a named value, an identifier does not have an associated data type. You can use an identifier as a resource value or callback procedure tag regardless of the data type specified in the object or procedure declaration. Identifiers are useful for specifying the position of an object based on the type of terminal on which the interface is displayed, or for passing a data structure (as opposed to a constant) to a private callback subroutine.

Support for Compound Strings

Most AIXwindows Toolkit objects require that all strings in the user interface (labels, menu items, and so on) must be compound strings. UIL fully supports the use of compound strings, including left-to-right and right-to-left writing direction and font selection.

Include Files for Useful Constants

The AIXwindows Toolkit provides a UIL include file that contains useful constants for coding a user interface. For example, some widgets have an **XmNorientation** resource parameter; you can use the constant **XmHORIZONTAL** or **XmVERTICAL** to specify this parameter.

Named Lists

UIL allows you to create named lists of the following resources, children widgets that share the same parent, and callback procedures that you can later refer to by name. This feature allows you to easily reuse common definitions by simply referencing these definitions by name.

Suggested Reading

Related Information

Using AIXwindows Resource Manager (MRM) Subroutines contains detailed information about using MRM subroutines.

How to Work on Large Projects Using the AIXwindows User Interface Language (UIL)

If several programmers are working together to specify the interface for an application, the AIXwindows User Interface Language (UIL) Module can be broken up into several small files, each containing a segment of the total interface specification, to allow several people to work in parallel on the UIL interface.

Prerequisite Tasks or Conditions

1. You must be familiar with the AIXwindows User Interface Language (UIL).
2. You must be acquainted with the C language data structures and subroutines in the AIXwindows Toolkit.

Procedure

To enable several programmers to work on the same UIL Module at the same time, complete the following steps in sequence:

1. Construct a main UIL file. Once you create this main UIL file, it rarely needs to be changed. The main UIL file should contain the following information:

- Comments describing copyright information, Module history, project information, and other relevant information
- Global declarations, such as case sensitivity, objects clause, and procedure declarations
- A series of include directives. Each include directive points to a UIL specification file containing some portion of the interface specification.

2. Divide the UIL Specification for your application interface into the following files, giving each file a unique name that reflects its contents:

- The **Shared Literals** file

Defines all literals shared between the UIL Module and the application source code. (These are the constants used as tags to the callback procedures.)

- The **Main Window** file

Defines the main window for the application. This could include a menu bar with associated cascade buttons, the work region, and other relevant pieces.

- The **Bulletin Board Dialogs** file

Defines all the bulletin board dialogs used in the application.

- The **Other Interface Objects** file

Defines all remaining objects that do not fit into the first three categories. This file might include display windows with their menu bars and work regions, pop–up menus, and the command dialog box.

Note: It is a matter of style whether the included files themselves contain include directives. You might prefer to work with a single main UIL file that names all of the remaining files needed to complete the interface specification. This can be helpful if you are translating the user interface into another national language because all files can easily be accounted for throughout the translation process.

Suggested Reading

Related Information

How to Create a User Interface at Run Time Using the AIXwindows Resource Manager (MRM) contains detailed information about the AIXwindows Resource Manager.

AIXwindows User Interface Language (UIL) Syntax Overview for Programming

UIL is a free–form language. This means that high–level constructs such as objects and value declarations, and low–level constructs such as keywords and punctuation characters do not need to begin in any particular column.

High–level constructs can span any number of lines. Low–level constructs, on the other hand, cannot span lines (except for string literals and comments).

The UIL Compiler accepts input lines up to 132 characters in length. These input lines can include any legal combination of the following programming elements:

- Legal Character Set
  - Letters
  - Digits
  - Formatting and Punctuation Characters
  - Other Characters
- Names
  - Case–sensitivity
  - Keywords
- Literals
  - String Literals

Concatenated String Literals

Compound String Literals

Character Sets for String Literals

Parsing Rules for Character Sets

Data Storage Consumption for String Literals

  - Integer Literals
  - Boolean Literals
  - Floating–Point Literals
- Value–Generating Subroutines
  - **CHARACTER_SET** Subroutine
  - **COLOR** Subroutine
  - **COLOR_TABLE** Subroutine
  - **ICON** Subroutine
  - **XBITMAPFILE** Subroutine
  - **FONT** Subroutine
  - **FONT_TABLE** Subroutine
  - **COMPOUND_STRING** Subroutine

- **COMPOUND_STRING_TABLE** Subroutine
- **ASCIZ_STRING_TABLE** Subroutine
- **INTEGER_TABLE** Subroutine
- **ARGUMENT** Subroutine
- **REASON** Subroutine
- **TRANSLATION_TABLE** Subroutine

- Include File for Useful Constants
- Compile–Time Value Expressions
- Suggested Reading

Legal Character Set

Use the following alphabetic, numeric, and formatting and punctuation character sets to construct the elements of UIL:

Note: Control characters, except for the form–feed character, are not permitted in a UIL Module. To construct a string literal containing a control character, use UIL escape sequences.

Letters

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Digits

```
0 1 2 3 4 5 6 7 8 9
```

Formatting and Punctuation Characters

<Space>, <tab>, <form–feed>, `_ ' ( ) * , + – . / ; : = ! $ \ { }`

Other Characters

Remaining 8–bit character codes in the range 160 to 255 decimal, including:

`& [ ] | < > % " # ? ~ ' ^`

All other characters are valid only in comments and string literals.

Punctuation in a UIL Module resembles that used in C programs. For example, statements end in a semicolon, braces are used to delimit definitions, and comments can be delimited by the slash–asterisk (/*) and asterisk–slash (*/) character sequences.

The following character sequences cannot be used in UIL names; they are reserved for punctuating UIL Modules:

( )	*Left parenthesis, right parenthesis.*
{ }	*Left brace, right brace.*
\ !	*Backslash, exclamation point.*
/**/	*Slash-asterisk, asterisk-slash.*
; :	*Semicolon, colon.*
' ,	*Apostrophe, comma.*
=	*equal sign.*

Spaces, tabs, and comments are special elements in the language. They are a means of delimiting other elements, such as two names. One or more of these elements can appear before or after any other element in the language. However, spaces, tabs, and comments that appear in string literals are treated as character sequences rather than delimiters.

Comments can take one of two forms, as follows:

- The comment is introduced with the sequence (`/*`) followed by the text of the comment and terminated with the sequence (`*/`). This form of comment can span multiple source lines.
- The comment is introduced with an exclamation point (`!`), followed by the text of the comment and terminated by the end of the source line.

Note: Neither of these two forms of comment can be nested.

The form–feed character is a control character; it cannot appear directly in a UIL specification file. Use the escape sequence `\12\` instead. The one exception to this rule is a form–feed that appears in column 1 of a line of source code due to the common practice of separating parts of a program with a form–feed. In this case, The form–feed causes a page break in the module listing.

Names

Each value, procedure, and object in UIL is identified by a name. This name can also be used to reference the entity elsewhere in UIL Module. Names have a maximum length of 31 characters and cannot begin with a digit. Names can consist of any of the following characters:

- **A–Z**
- **a–z**
- **0–9**
- dollar sign (**$**)
- underscore (**_**)

Case–sensitivity

UIL gives you a choice of either case–sensitive or case–insensitive names through a clause in the module header file. For example, if names are case sensitive, the names "sample" and "Sample" are distinct from each other. If names are case insensitive, these names could be used interchangeably. By default, names are case insensitive in UIL. In case–insensitive mode, the UIL Compiler outputs all names in the UID file in uppercase form. In case–sensitive mode, names appear in the UIL file exactly as they appear in the source.

You must define any referenced name exactly once in a UIL Module; if you omit a definition or define the same name more than once, the UIL Compiler issues an error at compile time.

Keywords

Keywords are names that have special meaning in UIL. The following types of keywords are used in UIL:

- Reserved keywords

You cannot use any of these reserved keywords to name an entity.

Reserved Keywords	Description
ARGUMENTS	Identifies an parameters list.
CALLBACKS	Identifies a callbacks list.

CONTROLS	Identifies a controls list.
END	Signifies the end of the UIL Module.
EXPORTED	Specifies that this object or value can be referenced by other UIL Modules.
FALSE	Represents the Boolean value zero; synonym for OFF.
GADGET	Specifies an object as the gadget variant (For objects having a widget and a gadget variant).
IDENTIFIER	Indicates an identifier declaration.
INCLUDE	Used with the nonreserved keyword FILE to specify an include file.
LIST	Identifies a list declaration.
MODULE	Signifies the start of a UIL Module.
OFF	Represents the Boolean value zero; synonym for FALSE.
ON	Represents the Boolean value 1; synonym for **TRUE**.
OBJECT	Identifies an object declaration.
PRIVATE	Specifies that this object or value cannot be referenced by other UIL Modules.
PROCEDURE	Identifies a procedure declaration.
PROCEDURES	Identifies a procedures list.
TRUE	Represents the Boolean value 1; synonym for ON.
VALUE	Identifies a value (literal) declaration.
WIDGET	For objects having a widget and a gadget variant, specifies this object as the widget variant.

- Nonreserved keywords

You can use a nonreserved keyword to name an entity in UIL without generating an error message. However, if you use a nonreserved keyword as a name, you cannot also use the UIL–supplied definition of that keyword. For example, if you use the name of a resource parameter (such as **XmNx**) to name a specific value, you cannot also specify the x parameter in any object definitions.

Nonreserved Keyword	**Description**
Built–in parameter name (**XmNx, XmNheight**)	Identifies an object parameter (widget–specific attribute).
Built–in reason names (**XmNactivate, XmNhelp**)	Identifies a callback reason.
Character set names (ISO_HEBREW_LR)	Identifies a character set and (implicit) writing direction.
Object types (**XmPushButton**)	Identifies an AIXwindows Toolkit object.
ANY	Suppresses data type checking.
ARGUMENTS	Identifies the built–in **ARGUMENTS** subroutine.
ASCIZ_STRING_TABLE	Specifies a value as UIL data type asciz_table.

ASCIZ_TABLE	Specifies a value as UIL data type asciz_table.
BACKGROUND	In a color table, specifies monochrome mapping to the background color.
BOOLEAN	Specifies a literal as UIL data type boolean.
CASE_INSENSITIVE	Used with nonreserved keyword NAMES to specify that names and keywords in the UIL Module are case insensitive.
CASE_SENSITIVE	Used with nonreserved keyword NAMES to specify that names and keywords in the UIL Module are case sensitive.
CHARACTER_SET	Identifies default character set clause; identifies the built–in **CHARACTER_SET** subroutine.
COLOR	Specifies a value as UIL data type color.
COLOR_TABLE	Specifies a value as the UIL data type color_table.
COMPOUND_STRING	Specifies a value as UIL data type compound_string; identifies the built–in **COMPOUND_STRING** data conversion subroutine.
COMPOUND_STRING_TABLE	Specifies a value as UIL data type string_table.
FILE	Used with reserved keyword INCLUDE to specify an include file.
FLOAT	Specifies a literal as UIL data type float; identifies the **FLOAT** data conversion subroutine.
FONT	Specifies a value as UIL data type font.
FONT_TABLE	Specifies a value as UIL data type font_table.
FOREGROUND	In a color table, specifies monochrome mapping to the foreground color.
ICON	Specifies a value as UIL data type pixmap; identifies the built–in **ICON** subroutine.
IMPORTED	Specifies that this literal takes its value from a corresponding literal in another UIL Module.
INTEGER	Specifies a literal as UIL data type integer; identifies the **INTEGER** data conversion subroutine.
INTEGER_TABLE	Specifies a value as UIL data type integer_table.
MANAGED	Specifies that a child is managed by its parent.
NAMES	Identifies the case sensitivity clause.
OBJECTS	Identifies the default object variant clause.
REASON	Specifies a value as UIL data type reason.
RIGHT_TO_LEFT	Specifies the writing direction in the **COMPOUND_STRING** subroutine.
STRING	Specifies a literal as UIL data type string.
STRING_TABLE	Specifies a value as UIL data type string_table.

TRANSLATION_TABLE	Specifies a value as UIL data type translation_table.
UNMANAGED	Specifies that a child is unmanaged by its parent.
VERSION	Identifies the version clause.
XBITMAPFILE	Specifies a value as UIL data type pixmap; identifies the **XBITMAPFILE** subroutine.

If you specify case–insensitive mode, you can type UIL keywords in uppercase, lowercase, or mixed case. If you specify case–sensitive mode, you must type UIL keywords in lowercase. You cannot abbreviate keywords by truncating characters from the end.

Note: all examples assume case–insensitive mode. Keywords are shown in uppercase letters to distinguish them from user–specified names, which are shown in lowercase letters. However, the use of uppercase letters is not required in case–insensitive mode.

In the following example `ARGUMENTS` is a keyword and `circle_radius` is the user–specified value of a resource:

```
•
•
•
{ ARGUMENTS
    { circle_radius = 1000 };
•
•
•
};
```

You could type the keyword `ARGUMENTS` in lowercase, uppercase, or mixed–case letters as long as you specified that names are case insensitive. If you specify that names are case sensitive, you must type the keyword in lowercase letters.

Literals

Literals are one means of specifying a value. UIL provides literals for several of the value types it supports. Some of the value types are not supported as literals (for example, pixmaps and string tables). Specify values for these types by using subroutines provided by UIL.

UIL directly supports the following literal types:

- String literals
- Integer literals
- Boolean literals
- Floating–point literals

Note: You can designate UIL values as exported, imported, or private. An exported object or value can be referenced in another UIL Module that uses the same name for the object or value and indicates that the object or value is to be imported. By default, top–level objects are exported, and all other objects and values are private (not accessible by other UIL Modules).

String Literals

A string literal is a sequence of zero or more 8–bit or 16–bit characters or a combination delimited by single quotation marks (') or double quotation marks ("). A string literal cannot exceed approximately 2000 characters.

A single–quoted string literal can span multiple source lines. To continue a single–quoted string literal, terminate the continued line with a backslash (\). The literal continues with the first character on the next line.

Double–quoted string literals cannot span multiple source lines because double–quoted strings can contain escape sequences and other special characters that prevent you from using the backslash character to designate continuation of the string. To build a string value that spans multiple source lines, use the concatenation operator.

The syntax of a string literal fits one of the following patterns:

- `'[ char... ]'`
- `[ #char–set ]"[ char... ]"`

Both string forms associate a character set with a string value. UIL uses the following rules to determine the character set and storage format for string literals:

- A string declared as `'string'` is equivalent to `#ISO_LATIN1"string"`. (The `ISO_LATIN1` character set matches the default font for the AIXwindows Toolkit.)
- A string declared as `"string"` is equivalent to `#char–set"string"` if you specified `char–set` as the default character set for the Module. Otherwise, `"string"` is equivalent to `#ISO_LATIN1"string"`.
- A string of the form `"string"` or `#char–set"string"` is stored as a null–terminated string.

The following examples contain valid and invalid string literal syntax:

Note: The **COMPOUND_STRING** subroutine forces the UIL Compiler to generate a compound string.

Form	**Storage Format**
`'string'`	Null–terminated string. Character set is ISO_LATIN1.
`#char–set'string'`	Invalid syntax. Will not compile.
`COMPOUND_STRING('string')`	Compound string. Character set is ISO_LATIN1.
`"string"`	Null–terminated string. If specified, the string has the default character set for the module. Otherwise, the character set is ISO_LATIN1.
`#ISO_GREEK"string"`	Null–terminated string. Character set is ISO_GREEK.
`COMPOUND_STRING("string")`	Compound string. If specified, the string has the default character set for the module. Otherwise, the character set is ISO_LATIN1.
`COMPOUND_STRING(#ISO_ARABIC"string")`	Compound string. Character set is ISO_ARABIC.
`'string'&"string"`	If the character sets and writing directions of the operands match, the resulting string is null–terminated. Otherwise, the result is a

	multiple–segment compound string. The string has the character set or sets specified for the individual segments.
`"string"&#ISO_HEBREW"string"`	If the implicit character set and writing direction for the left operand matches the explicit character set (ISO_HEBREW) and writing direction (right to left) for the right operand, the resulting string is a null–terminated string. Otherwise, the result is a multiple–segment compound string.

String literals can contain characters with the eighth (high–order) bit set. You cannot type control characters (00..1F, 7F, and 80..9F) directly in a single–quoted string literal. However, you can represent these characters with escape sequences. The following characters cannot be directly included in a UIL string literal, but can be represented by the indicated escape sequences:

Meaning of Control Character	UIL Escape Sequence
Backspace	`\b`
Form–feed	`\f`
Newline	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Vertical tab	`\v`
Apostrophe	`\'`
Quotation mark	`\"`
Backslash	`\`
Character whose internal representation is given by integer (in the range 0 to 255 decimal)	`\integer\`

Many AIXwindows Toolkit routines use the null character to determine the end of a string. The UID file also holds the string length so that it is possible to deal correctly with strings that have embedded nulls.

Concatenated String Literals

The ampersand (&) is the UIL concatenation operator. It takes two strings as operands and creates a new string made up of the left operand followed immediately by the right operand as indicated in the following example:

'abcd' & 'xyz' *becomes* **'abcdxyz'**

The operands of the concatenation operator can be null–terminated strings, compound strings, or any combination of these two types of strings. The operands can hold string values of the same or different character sets.

The string resulting from the concatenation is a null–terminated string unless one or more of the following conditions exists, in which case the resulting string is a compound string:

- One of the operands is a compound string

- The operands have different character set properties
- The operands have different writing directions

You cannot use imported or exported values as operands of the concatenation operator.

Compound String Literals

A compound string consists of a string of 8–bit or 16–bit characters, a named character set, and a writing direction. Its UIL data type is compound_string.

The writing direction of a compound string is implied by the character set specified for the string. You can explicitly set the writing direction for a compound string by using the **COMPOUND_STRING** subroutine.

A compound string can consist of a sequence of concatenated compound strings, null–terminated strings, or any combination of these two types of strings, each of which can have a different character set property and writing direction. Use the concatenation operator (&) to create a sequence of compound strings as in the following example:

```
#ISO_HEBREW"txet werbeh"&#ISO_LATIN8"latin text"
```

Each string in the sequence is stored in the UID file along with its character set information and writing direction information. You can manipulate a compound string with the AIXwindows Toolkit routines for compound strings.

Generally, a string literal is stored in the UID file as a compound string when the literal consists of concatenated strings having different character sets or writing directions, or when you use the string to specify a value for a parameter that requires a compound string value. If you want to guarantee that a string literal is stored as a compound string, you must use the **COMPOUND_STRING** subroutine."

Character Sets for String Literals

The following character sets for string literals are supported by the UIL Compiler:

Notes:

1. several UIL names map to the same character set. In some cases the UIL name influences how string literals are read. For instance, strings identified by a UIL character set name ending in _LR are read left–to–right.
2. Names that end in a different number reflect different fonts (for example, ISO_LATIN1 or ISO_LATIN6)
3. All character sets for UIL string literals are represented by 8 bits.

UIL Name	**Description**
ISO_LATIN1	GL: ASCII, GR: Latin–1 Supplement.
ISO_LATIN2	GL: ASCII, GR: Latin–2 Supplement.
ISO_ARABIC	GL: ASCII, GR: Latin–Arabic Supplement.
ISO_LATIN6	GL: ASCII, GR: Latin–Arabic Supplement.
ISO_GREEK	GL: ASCII, GR: Latin–Greek Supplement.
ISO_LATIN7	GL: ASCII, GR: Latin–Greek Supplement.
ISO_HEBREW	GL: ASCII, GR: Latin–Hebrew Supplement.
ISO_LATIN8	GL: ASCII, GR: Latin–Hebrew Supplement.
ISO_HEBREW_LR	GL: ASCII, GR: Latin–Hebrew Supplement.

ISO_LATIN8_LR	GL: ASCII, GR: Latin–Hebrew Supplement.
JIS_KATAKANA	GL: JIS Roman, GR: JIS Katakana.

Parsing Rules for Character Sets

The following parsing rules are observed for each character set:

Character Set	**Parsing Rules**
All character sets	Character codes in the range 00..1f, 7f, and 80..9f are control characters including both bytes of 16–bit characters. The UIL Compiler flags these as illegal characters.
ISO_LATIN1, ISO_LATIN2 ISO_ARABIC, ISO_LATIN3 ISO_GREEK, ISO_LATIN4	These sets are parsed from left to right. The escape sequences for null–terminated strings are also supported by these character sets.
ISO_HEBREW, ISO_LATIN8	These sets are parsed from right to left. The string `#ISO_HEBREW "012345"` generates the primitive string `"543210"` with character set `ISO_HEBREW`. A DDIS descriptor for such a string has this segment marked as right_to_left. The escape sequences for null–terminated strings are also supported by these character sets and the characters that compose the escape sequences are in left–to–right order. For example, you enter `\n`, not `n\`.
ISO_HEBREW_LR, ISO_LATIN8_LR	These sets are parsed from left to right. The string `#ISO_HEBREW_LR "012345"`, for example, generates the primitive string `"012345"` with character set `ISO_HEBREW`. A DDIS descriptor for such a string marks this segment as being left_to_right. The escape sequences for null–terminated strings are also supported by these character sets.
JIS_KATAKANA	This set is parsed from left to right. The escape sequences for null–terminated strings are also supported by this character set. The backslash (\) may be displayed as a yen symbol.

In addition to designating parsing rules for strings, character set information remains a resource of a compound string. If the string is included in a string consisting of several concatenated segments, the character set information is included with that string segment. This gives the AIXwindows Toolkit enough information to decipher the compound string and choose an appropriate display font.

For an application interface displayed only in English, UIL lets you ignore the distinctions between the two uses of strings. The UIL Compiler recognizes by context when a string must be passed as a null–terminated string or as a compound string.

The UIL Compiler recognizes enough about the various character sets to correctly parse string literals. The UIL Compiler also issues error messages if you use a compound string in a context that supports only null–terminated strings.

Note: The character set names are keywords. If case–sensitive names are in force, put names in lowercase. If names are case insensitive, character set names can be either uppercase, lowercase, or mixed case.

In addition to the built–in character sets recognized by UIL, you can define your own character sets with the **CHARACTER_SET** subroutine. You can use the **CHARACTER_SET** subroutine anywhere a character set can be specified.

Data Storage Consumption for String Literals

The way a string literal is stored in the UID file depends on how you declare and use the string. The UIL Compiler automatically converts a null–terminated string to a compound string if you use the string to specify the value of a parameter that requires a compound string. However, this conversion uses up critical amounts of storage space.

Private, exported, and imported string literals require storage for a single allocation when the literal is declared. Thereafter, storage is required for each reference to the literal. Literals declared inline require storage for both an allocation and a reference.

The storage requirement for an allocation consists of a fixed portion and a variable portion. The fixed portion of an allocation is roughly the same as the storage requirement for a reference (a few bytes). The storage consumed by the variable portion depends on the size of the literal value (that is, the length of the string). To conserve storage space, avoid making string literal declarations that result in an allocation per use.

The following summary describes the data storage consumed by various string literals:

Declaration	**Data Type**	**Used As**	**Storage Requirements Per Use**
In line	Null–terminated	Null–terminated	Allocation and reference (within the module).
Private	Null–terminated	Null–terminated	Reference (within the module).
Exported	Null–terminated	Null–terminated	Reference (within the UID hierarchy).
Imported	Null–terminated	Null–terminated	Reference (within the UID hierarchy).
In line	Null–terminated	Compound	Allocation and reference (within the module).
Private	Null–terminated	Compound	Allocation and reference (within the module).
Exported	Null–terminated	Compound	Reference (within the UID hierarchy).
Imported	Null–terminated	Compound	Reference (within the UID hierarchy).
In line	Compound	Compound	Allocation and reference (within the module).
Private	Compound	Compound	Reference (within the module).
Exported	Compound	Compound	Reference (within the UID hierarchy).
Imported	Compound	Compound	Reference (within the UID hierarchy).

Integer Literals

An integer literal represents the value of a whole number. Integer literals have the form of an optional sign followed by one or more decimal digits. An integer literal must not contain embedded spaces or commas.

Integer literals are stored in the UID file as longwords. Exported and imported integer literals require a single allocation when the literal is declared; thereafter, a few bytes of storage are required for each reference to the literal. Private integer literals and those declared in line require allocation and reference storage per use. To conserve storage space, avoid making integer literal declarations that result in an allocation per use.

The following summary describes data storage consumption for integer literals:

Declaration	Storage Requirements Per Use
In line	An allocation and a reference (within the module).
Private	An allocation and a reference (within the module).
Exported	A reference (within the UID hierarchy).
Imported	A reference (within the UID hierarchy).

Boolean Literals

A Boolean literal represents the value true (reserved keyword **TRUE** or **ON**) or false (reserved keyword **FALSE** or **OFF**). These keywords are subject to case–sensitivity rules.

In a UID file, **TRUE** is represented by the integer value 1 and **FALSE** is represented by the integer value 0.

Data storage consumption for Boolean literals is the same as that for integer literals.

Floating–Point Literals

A floating–point literal represents the value of a real (or float) number. Floating–point literals are expressed in one of the following forms:

- [+ | –] digit...
- [digit...] [{ E | e } [+ | –] digit...
- [+ | –] digit... [{ E | e } [+ | –] digit...]

A floating–point literal can represent values in the range .29E–38 to 1.7E+38 with up to 16 significant digits. A floating–point literal must not contain embedded spaces or commas.

Floating–point literals are stored in the UID file as double–precision floating–point numbers. The following examples of valid and invalid floating–point notation indicate the manner in which the UIL Compiler handles floating–point literals.

Valid Floating–Point Literals	Invalid Floating–Point Literals
1.0	1e1 (no decimal point).
.1	E–1 (no decimal point or digits).
3.1415E–2 (which equals .031415)	2.87 e6 (embedded blanks).
–6.29e7 (which equals –62900000)	2.0e100 (out of range).

Note: Data storage consumption for floating–point literals is the same as that for integer literals.

Value–Generating Subroutines

UIL provides subroutines to generate the following types of values:

- Character sets
- Colors
- Pixmaps
- Fonts
- Font tables

- Compound strings
- Compound string tables
- ASCIZ string tables
- Integer tables
- Parameters
- Reasons
- Translation tables

Note: The examples in the following sections assume case–insensitive mode. Keywords are shown in uppercase letters to distinguish them from user–specified names, which are shown in lowercase letters. This use of uppercase letters is not required in case–insensitive mode. In case–sensitive mode, keywords must be in lowercase letters.

CHARACTER_SET Subroutine

You can define your own character sets with the **CHARACTER_SET** subroutine. You can use the **CHARACTER_SET** subroutine anywhere a character set can be specified, providing you use the following syntax:

```
CHARACTER_SET (string–expression [,property]...);
```

The result of the **CHARACTER_SET** subroutine is a character set named **string–expression** with the properties you specify. The **String–expression** character set name must be a null–terminated string. At your option you can include one or both of the following clauses to specify properties for the resulting character set:

- RIGHT_TO_LEFT = boolean–expression

Sets the default writing direction of the string to from right-to-left if the boolean–expression is true, and left–to–right otherwise.

- SIXTEEN_BIT = boolean–expression

Allows the strings associated with this character set to be interpreted as 16–bit characters if boolean–expression is true. They are interpreted as 8–bit characters otherwise.

COLOR Subroutine

The **COLOR** subroutine supports the definition of colors. You designate a value to specify a color. You then use that value for parameters requiring a color value.

The **COLOR** subroutine has the following syntax:

```
COLOR (string–expression [,FOREGROUND|,BACKGROUND])
```

The string expression names the color you want to define. The optional keywords FOREGROUND and BACKGROUND identify how the color is to be displayed on a monochrome device when the color is used in the definition of a color table.

The following example shows how to use the **COLOR** subroutine:

```
VALUE red: COLOR( 'Red' );
VALUE green: VALUE blue: COLOR( 'Blue' );
OBJECT primary_window:
    XmMainWindow
    { ARGUMENTS
        { XmNforeground = green;
          XmNbackground = COLOR( 'Black' ); };
    };
```

In this example, the **COLOR** subroutine is used with the VALUE declaration to define three colors and give each a name. One of these colors, green, is then used to specify the foreground color of the main window.

A second use of the **COLOR** subroutine is to define the main window background color as the color associated with the string 'Black.'

The UIL Compiler does not have built–in color names. Colors are server–dependent resources of an object. Colors are defined on each server and may have different RGB values on each server. The string you specify as the color parameter must be recognized by the server on which your application runs.

In a UID file, UIL represents a color as a character string. MRM calls X translation subroutines that convert a color string to the device–specific pixel value. If your application is running on a monochrome server, all colors translate to black or white. If it is running on a color server and all of the following conditions are met, the color names translate to their proper colors: providing that the following two conditions are met:

- The color is defined
- The color map is not yet full

If the color map is full, even valid colors translate to black or white (foreground or background).

Interfaces do not, in general, specify colors for widgets, so that the selection of colors can be controlled by the user through the **.Xdefaults** file and the preferred editor.

To write an application that runs on both monochrome devices and color devices, specify which colors in a color table (defined with the **COLOR_TABLE** subroutine) map to the background and which colors map to the foreground. UIL lets you use the **COLOR** subroutine to designate this mapping in the definition of the color. The following example shows how to use the **COLOR** subroutine to map the color red to the background color on a monochrome device:

```
VALUE c: COLOR ( 'red',BACKGROUND );
```

The mapping comes into play only when the MRM is given a color and the application is displayed on a monochrome device. In this case, each color falls into one of the following three categories:

- The color is mapped to the background color on the monochrome device
- The color is mapped to the foreground color on the monochrome device
- Monochrome mapping is undefined for this color

If the color is mapped to the foreground color, MRM substitutes the foreground color. If the color is mapped to the background color, MRM substitutes the background color. If you do not specify the monochrome mapping for a color, MRM passes the color string to the AIXwindows Toolkit for mapping to the foreground or background color.

Subroutines for Specifying Pixmaps

Pixmap values are designed to let you specify labels that are graphic images rather than text. Pixmap values are not directly supported by UIL. Instead, UIL supports icons, which are a simplified form of pixmap. You use a character to describe each pixel in the icon.

Pixmap support in UIL Compiler is provided by the following subroutines:

- **COLOR_TABLE**
- **ICON**
- **XBITMAPFILE**

In a UIL Module, any parameter of type pixmap must have an icon or an Xbitmap file specified as its value.

COLOR_TABLE Subroutine

The **COLOR_TABLE** subroutine has the following syntax:

```
COLOR_TABLE ( { color–expression = character },... )
```

Parameter	**Description**
color expression	Can be any previously defined color, a color defined in line with the **COLOR** subroutine, or the phrase BACKGROUND COLOR or FOREGROUND COLOR.
character	Can be any valid UIL character.

The following example shows how to specify a color table:

```
VALUE
    rgb : COLOR_TABLE ( red = 'r', green = 'g', blue = 'b' );
    bitmap_colors : COLOR_TABLE ( BACKGROUND COLOR = '0',
                                  FOREGROUND COLOR = '1' );
```

The **COLOR_TABLE** subroutine provides a device–independent way to specify a set of colors. It accepts either previously defined UIL color names or inline color definitions (using the **COLOR** subroutine). A color table must be private because its contents must be made available to the UIL Compiler to construct an icon. The colors within a color table, however, can be imported, exported, or private.

The single letter associated with each color is the character you use to represent that color when creating an icon. Each letter used to represent a color must be unique within the color table.

ICON Subroutine

The **ICON** subroutine has the following syntax:

ICON ([COLOR_TABLE=*color–table–name* ,] *row*,...)

Parameter	**Description**
color–table–name	Refer to a previously defined color table.
row	A character expression giving one row of the icon.

The following example shows how to define a pixmap using the COLOR_TABLE and **ICON** subroutines:

```
VALUE
    rgb : COLOR_TABLE ( red = '=', green = '.', blue = ' ' );
    x_icon : ICON( COLOR_TABLE=rgb,     '=========',
                                        '==.   .==',
                                        '== . . ==',
                                        '==  .  ==',
                                        '== . . ==',
                                        '==.   .==',
                                        '=========');
```

The **ICON** subroutine describes a rectangular icon that is x pixels wide and y pixels high. The strings surrounded by single quotation marks describe the icon. Each string represents a row in the icon; each character in the string represents a pixel.

The first row in an icon definition determines the width of the icon. All rows must have the same number of characters as the first row. The height of the icon is dictated by the number of rows. For example, the X icon defined above is 9 pixels wide and 7 pixels high.

The first parameter of the **ICON** subroutine (the color table specification) is optional and identifies the colors that are available in this icon. By using the single letter associated with each color, you can specify the color of each pixel in the icon. In the example, an equal sign (=) represents the color red, a period (.) is green, and a space is blue. The icon must be constructed of characters defined in the specified color table. In the above example, the color table named rgb specifies colors for the equal sign (=), period (.), and space. The X icon is constructed with these three characters.

A default color table is used if you omit the parameter specifying the color table. To make use of the default color table, the rows of your icon must contain only spaces and asterisks.

The default color table is defined as follows:

```
COLOR_TABLE( BACKGROUND COLOR = ' ', FOREGROUND COLOR = '*' )
```

You can define other characters to represent the background color and foreground color by replacing the space and asterisk in the BACKGROUND COLOR and FOREGROUND COLOR clauses shown in the previous statement. You can specify icons as private, imported, or exported. Use the MRM **MrmFetchIconLiteral** subroutine to retrieve an exported icon at run time.

XBITMAPFILE Subroutine

The **XBITMAPFILE** subroutine is similar to the **ICON** subroutine in that both describe a rectangular icon that is x pixels wide and y pixels high. However, the **XBITMAPFILE** subroutine allows you to specify an external file containing the definition of an Xbitmap, while all **ICON** subroutine definitions must be coded directly within UIL. Xbitmap files can be generated by many different X applications. UIL reads these files through the **XBITMAPFILE** subroutine, but does not support creation of these files. At run time, MRM reads the Xbitmap file specified as the parameter to the **XBITMAPFILE** subroutine.

The **XBITMAPFILE** subroutine returns a value of type pixmap and can be used anywhere a pixmap data type is expected. The **XBITMAPFILE** subroutine has the following syntax:

```
XBITMAPFILE( string—expression );
```

The following example shows how to use the **XBITMAPFILE** subroutine:

```
VALUE
XmNbackgroundPixmap=XBITMAPFILE('myfile_button.xbm');
```

In this example, the Xbitmap specified in `myfile_button.xbm` is used to create a pixmap, which can be referenced by the value `background_pixmap`.

FONT Subroutine

The UIL Compiler has no built–in fonts. You must define all fonts using the **FONT** subroutine. To do so, designate a value to specify a font and then use that value for parameters that require a font value.

Each font makes sense only in the context of a character set. The **FONT** subroutine has an additional parameter to let you specify the character set for the font. This parameter is optional; if you omit it, the default character set is ISO_LATIN1. The **FONT** subroutine has the following syntax:

```
FONT( string–expression [, CHARACTER_SET = char–set ] )
```

Parameter	**Description**
string–expression	Specifies the name of the font. Cannot be a compound string.
char–set	Specifies the character set for the font.

The following example shows how to use the **FONT** subroutine:

```
VALUE big:
  FONT('–ADOBE–Times–Medium–R–Normal––*–140–*–*–P–*–ISO8859–1');
VALUE bold:
  FONT('–ADOBE–Helvetica–Bold–R–Normal––*–100–*–*–P–*–ISO8859–1');
OBJECT danger_window:
    XmWarningDialog
    { ARGUMENTS
        { XmNdialogTitle =  'You are about to lose all changes';
          XmNlabelFontList = bold;
        };
    };
```

In this example, the **FONT** subroutine is used with the VALUE declaration to define two fonts and give them names. The UIL Compiler automatically converts one of these fonts, bold, to a font table (the parameter `XmNlabelFontList` requires a font table). This table specifies the text font of the warning dialog.

Use the pattern matching character (*) to specify fonts in a device–independent manner.

Whenever possible, allow end users to select fonts through the **.Xdefaults** file. This approach allows them to control font selection rather than having fonts specified in your interface source code.

FONT_TABLE Subroutine

A font table is a sequence of pairs of fonts and character sets. At run time, when an object must display a string, the object scans the font table for the character set that matches the character set of the string to be displayed.

UIL provides the **FONT_TABLE** subroutine to provide you with a means of supplying such a parameter. The syntax of the **FONT_TABLE** subroutine is as follows:

```
FONT_TABLE( font expression,... )
```

The *font expression* parameter is created with the **FONT** subroutine.

If you specify a single font value to specify an parameter requiring a font table, the UIL Compiler automatically converts a font value to a font table.

COMPOUND_STRING Subroutine

Use the **COMPOUND_STRING** subroutine to set the following properties of a null–terminated string and to convert it into a compound string:

- the character set

- writing direction
- separator

The **COMPOUND_STRING** subroutine has the following syntax:

```
COMPOUND_STRING( string–expression [, property ]...);
```

The **COMPOUND_STRING** subroutine returns a compound string with the string expression as its value. You can optionally include one or more of the following clauses to specify properties for the resulting compound string:

- CHARACTER_SET = char–set
- RIGHT_TO_LEFT = boolean–expression
- SEPARATE = boolean–expression

Clause	**Description**
CHARACTER_SET	Defines the character set.

Note: If you omit a character set clause, the resulting string has the same character set as the string expression.

RIGHT_TO_LEFT	Sets the writing direction of the string from right–to–left if boolean–expression is **TRUE** and left–to–right if boolean–expression is **FALSE**.

Note: Specifying this parameter does not cause the value of the string expression to change. If you omit the RIGHT_TO_LEFT parameter, the resulting string has the same writing direction as string–expression.

SEPARATE	Appends a separator to the end of the compound string if boolean–expression is **TRUE**.

Note: If you omit the SEPARATE clause, the resulting string does not have a separator.

Note: You cannot use imported or exported values as the operands of the **COMPOUND_STRING** subroutine.

COMPOUND_STRING_TABLE Subroutine

A compound string table is an array of compound strings. Objects requiring a list of string values, such as the XmNitems and XmNselectedItems parameters for the list widget, use string table values. The **COMPOUND_STRING_TABLE** subroutine builds the values for these two parameters of the list widget. The **COMPOUND_STRING_TABLE** subroutine generates a value of type string_table.

The **COMPOUND_STRING_TABLE** subroutine has the following syntax:

```
COMPOUND_STRING_TABLE(string–expression,...)
```

The following example shows how to specify a string table:

```
OBJECT file_privileges: XmList
    { ARGUMENTS
        { XmNitems = COMPOUND_STRING_TABLE("owner read",
                                           "owner write",
                                           "owner delete",
                                           "system read",
                                           "system write",
                                           "system delete",
                                           "group read",
                                           "group write",
                                           "group delete") ;
          XmNselectedItems = COMPOUND_STRING_TABLE
                                          ("owner read",
                                           "owner write",
                                           "system read:,
                                           "system write" );
        };
    };
```

This example creates a list box with nine menu choices. Four of the choices are initially displayed as having been selected.

The strings inside the string table can be simple strings, which the UIL Compiler automatically converts to compound strings if the strings are declared as null–terminated strings.

ASCIZ_STRING_TABLE Subroutine

An ASCIZ string table is an array of ASCIZ (null–terminated) string values separated by commas. This subroutine allows you to pass more than one ASCIZ string as a callback tag value. The **ASCIZ_STRING_TABLE** subroutine returns a value of type asciz_table.

The **ASCIZ_STRING_TABLE** subroutine has the following syntax:

```
ASCIZ_STRING_TABLE(string–expression,...);
```

The following example shows how to specify an ASCIZ string table:

```
VALUE
    v1 = "my_value_1";
    v2 = "my_value_2";
    v3 = "my_value_3";
OBJECT press_my: XmPushButton {
    CALLBACKS {
        XmNactivateCallback =
                 PROCEDURE my_callback(asciz_table (v1, v2, v3));
        };
    };
```

INTEGER_TABLE Subroutine

An integer table is an array of integer values separated by commas. This subroutine allows you to pass more than one integer per callback tag value.

The **INTEGER_TABLE** subroutine has the following syntax:

```
INTEGER_TABLE( integer–expression,...);
```

The following example shows the **INTEGER_TABLE** subroutine defining an array of integers to be passed as a callback tag to the procedure my_callback:

```
VALUE
v1 = 1;
v2 = 2;
v3 = 3;
OBJECT press_my: XmPushButton {
    ARGUMENTS {
        XmNheight = 30;
        XmNwidth = 10;
    };
    CALLBACKS {
        XmNactivateCallback =
                PROCEDURE my_callback(integer_table(v1, v2, v3));
    };
};
```

ARGUMENT Subroutine

The **ARGUMENT** subroutine defines the parameters for a user–defined widget.

Each object that can be described by the UIL permits a set of parameters. For example, XmNheight is a parameter to most objects and has integer data type. To specify height for a user–defined widget, use the built–in parameter name XmNheight and specify an integer value when you declare the user–defined widget. You do not use the **ARGUMENT** subroutine to specify parameters that are built into the UIL Compiler.

The **ARGUMENT** subroutine has the following syntax:

ARGUMENT(*string–expression* [, *parameter_type*])

Parameter	Description
string–expression	Specifies the name the UIL Compiler uses for the parameter in the UID file.
parameter_type	Specifies the the type of value that can be associated with the parameter.

Note: If you omit the second parameter, the default type is ANY and no value type checking occurs.

Use one of the following keywords to specify the parameter type:

- ANY
- ASCIZ_TABLE
- BOOLEAN
- COLOR
- COLOR_TABLE
- COMPOUND_STRING
- FLOAT
- FONT
- FONT_TABLE
- ICON
- INTEGER
- INTEGER_TABLE

- REASON
- STRING
- STRING_TABLE
- TRANSLATION_TABLE

The following example shows how to use the **ARGUMENT** subroutine to define the parameters for a user–defined widget that draws a circle and takes the following four parameters:

- my_radius
- my_color
- XmNx
- XmNy

Note: The **ARGUMENT** subroutine is not used to specify XmNy and XmNy because these are built–in parameter names. The data type of XmNx and XmNy is integer.

When you declare the circle widget, you must use the **ARGUMENT** subroutine to define the name and data type of the parameters that are not built–ins (my_radius and my_color). Parameters are specified in an arguments list, identified by the keyword **ARGUMENTS** in the following example:

```
VALUE circle_radius: ARGUMENT( 'my_radius', INTEGER );
VALUE circle_color: ARGUMENT( 'my_color', COLOR );
OBJECT circle:
    USER_DEFINED PROCEDURE CircleCreate
        { ARGUMENTS
            { circle_radius = 1000;
              circle_color = color_blue
              XmNx = 1050;
              XmNy = 1050;
            };
        };
```

In this example, the **ARGUMENT** subroutine is used in a value declaration to define the two parameters that are not UIL built–ins: circle–radius (which takes an integer value) and circle–color (which takes a color value). When these parameters are referenced in the arguments list of the circle widget, the UIL Compiler verifies that the value you specify for each of these parameters is of the type specified in the **ARGUMENT** subroutine. The arguments list placed in the UID file (and supplied to the creation subroutine for the circle widget at run time) includes the following:

Parameter	**Parameter Value**
my_radius	1000.
my_color	Value associated with UIL name color_blue.

You can use the **ARGUMENT** subroutine to allow the UIL Compiler to recognize extensions to the AIXwindows Toolkit. For example, an existing widget may accept a new parameter. Using the **ARGUMENT** subroutine, you can make this new parameter available to the UIL Compiler before the updated version of the UIL Compiler is released.

REASON subroutine

The **REASON** subroutine is useful for defining new reasons for user–defined widgets.

Each of the objects in the AIXwindows Toolkit defines a set of conditions under which it calls a user–defined subroutine. These conditions are known as *callback reasons* (or *reasons*). The user–defined subroutines are termed callback procedures. In a UIL Module, you use a callbacks list to specify which user–defined subroutines are to be called for which reasons.

When you declare a user–defined widget, you can define callback reasons for that widget using the **REASON** subroutine. The **REASON** subroutine has the following syntax:

```
REASON(string–expression)
```

The string expression specifies the parameter name stored in the UID file for the reason. This reason name is supplied to the widget creation routine at run time.

For example, if you build a new widget that implements a password system to prevent a set of windows from being displayed unless the correct password is entered, the widget might define the following callbacks:

```
VALUE passed: REASON( 'AccessGrantedCallback' );
VALUE failed: REASON( 'AccessDeniedCallback' );
OBJECT guard_post:
    USER_DEFINED PROCEDURE guard_post_create
{ CALLBACKS
    { passed = PROCEDURE display_next_level();
      failed = PROCEDURE logout(); };
    };
```

Note: The **REASON** subroutine is used in a value declaration to define two new reasons, passed and failed. The callback list of the widget named guard_post specifies the procedures to be called when these reasons occur.

A widget specifies its callbacks by defining a parameter for each reason that it supports. The parameter to the **REASON** subroutine gives the name of the parameter that supports this reason. Therefore, the arguments list placed in the UID file for the guard_post widget includes the following parameters:

Parameter	**Parameter Value**
AccessGrantedCallback	Callback structure for procedure display_next_level.
AccessDeniedCallback	Callback structure for procedure logout.

TRANSLATION_TABLE Subroutine

Each of the AIXwindows Toolkit widgets has a translation table that maps X events (for example, mouse button 1 being pressed) to a sequence of actions. Through widget parameters, such as the common translations parameter, you can specify an alternate set of events or actions for a particular widget.

The **TRANSLATION_TABLE** subroutine creates a translation table that can be used as the value of a parameter that is of the data type translation_table. The **TRANSLATION_TABLE** subroutine has the following syntax:

```
TRANSLATION_TABLE( string–expression,... )
```

Each of the string expressions specifies the run–time binding of an X event to a sequence of actions, as shown in the following example:

```
LIST new_translations:
    ARGUMENTS
    { XmNtranslations =
      TRANSLATION_TABLE
      (
        '<Btn1Down>: XMPBFILLHIGHLIGHT() XMPBARM() XMPBUNGRAB()',
        '<Btn1Up>: XMPBFILLUNHIGHLIGHT() XMPBACTIVATE()
                   XmSelectionDialog(self_destruct XMPBDISARM()',
        '<Btn3Up>: XMPBHELP()',
        'Any(<LeaveWindow>: XMPBFILLUNHIGHLIGHT() XMPBUNGRAB()
                            XMPBDISARM()'
      );
    };
OBJECT self_destruct:
    XmPushButton
    { ARGUMENTS new_translations; };
```

This example defines an parameter list named new_translations. The first translation specifies that pressing the left button, <Btn1Down>, should result in the following sequence of procedures:

- XMPBFILLHIGHLIGHT()
- XMPBARM()
- XMPBUNGRAB()

The self_destruct push button widget defines new_translations as its translation table.

ANY Data Type

The purpose of the ANY data type is to shut off the data–type checking feature of the UIL Compiler. You can use the ANY data type to specify the following parameter types:

- callback procedure tags
- user–defined parameters

Use the ANY data type when you need to use a type not supported by the UIL Compiler or when you want the data type restrictions imposed by the UIL Compiler to be relaxed. For example, you can use the ANY data type to define a widget with a parameter that can accept different types of values depending on run–time circumstances.

Note: If you specify a parameter that takes an ANY value, the UIL Compiler does not check the type of the value specified for that parameter. If you specify a value for an parameter of type ANY, you could get unexpected results at run time if you pass a value having a data type that the widget does not support for that parameter.

Include File for Useful Constants

The file **XmAppl.uil** contains useful constants for specifying objects in UIL. Include this file in your UIL Module to have access to the constants.

Compile–Time Value Expressions

UIL provides literal values for a diverse set of types (integer, string, real, Boolean) and a set of AIXwindows Toolkit–specific types (colors and fonts). These values provide the value of AIXwindows Toolkit parameters.

UIL includes compile–time value expressions. These expressions can contain references to other UIL values but cannot be forward–referenced.

Compile–time value expressions are useful for implementing relative positioning of children widgets without using **XmForm** parent widgets. For example, to create an **XmMesssageBox** widget child inside an **XmBulletinBoard** parent widget using compile–time expressions (the child being half as wide and half as tall as the parent and centered within it), you can specify the coordinates of the **XmMessageBox** widget child by referring to the values you already defined for the parent widget. If you do not use compile–time value expressions in this case, you would have to compute the x and y location and the height and width of the widget child, then recompute these values whenever the parent widget changes size or location. In addition, the computed values would be absolute numbers rather than descriptive expressions like "bulletin_board_width/2".

Notes:

1. The AIXwindows Toolkit provides direct support for resolution independence with the millimeter, inch, and point unit types; the preceding example could be implemented using this feature.
2. The concatenation of strings is also a form of compile–time expression. Use the concatenation operator to join strings and convert the result to a compound string.

The following table lists set of operators in UIL that allow you to create integer, real, and Boolean values based on other values defined with UIL Module. A precedence (**P**) of 1 is the highest.

Op.	**Operator Type**	**Operand Types**	**Meaning**	**P**
~	Unary	Boolean	NOT	1
		integer	complement of 1	
–	Unary	float	Negate	1
		integer	Negate	
+	Unary	float,integer	NOP	1
		integer	NOP	1
*	Binary	float,float	Multiply	2
		integer,integer	Multiply	2
/	Binary	float,float	Divide	2
		integer,integer	Divide	2
+	Binary	float,float	Add	3
		integer,integer	Add	3
–	Binary	float,float	Subtract	3
		integer,integer	Subtract	3
>>	Binary	integer,integer	Shift right	4
<<	Binary	integer,integer	Shift left	4
&	Binary	Boolean,Boolean	AND	5
		integer,integer	Bitwise AND	5
		string,string	Concatentate	5
|	Binary	Boolean,Boolean	OR	6
		integer,integer	Bitwise OR	6
^	Binary	Boolean,Boolean	XOR	6
		integer,integer	Bitwise XOR	6

Notes:

1. A string can be either a single compound string or a sequence of compound strings. If the two concatenated strings have different properties (such as writing direction or character set), the result of the concatenation is a sequence of compound strings.
2. The result of each operator has the same type as its operands. You cannot mix types in an expression without using conversion routines.
3. You can use parentheses to override the normal precedence of operators.
4. In a sequence of unary operators, the operations are performed in right–to–left order. For example, **– + –A** is equivalent to **–(+(–A))**.
5. In a sequence of binary operators of the same precedence, the operations are performed in left–to–right order. For example, **A*B/D*D** is equivalent to **((A*B)/C)*D**.
6. A value declaration gives a value a name. You cannot redefine the value of that name in a subsequent value declaration.
7. You can use a value containing operators and subroutines anywhere you can use a value in a UIL Module.
8. You cannot use exported or imported values as operands in expressions.
9. Several of the binary operators are defined for multiple data types. For example, the operator for multiplication (*) is defined for both floating–point and integer operands. For the UIL Compiler to perform these binary operations, both operands must be of the same type. If you supply operands of different data types, the UIL Compiler automatically converts one of the operands to the type of the other according to the following conversions rules:

Data Type	**Data Type**	**Conversion Rule for Operands**
Boolean	Integer	Operand 1 converted to integer
Integer	Boolean	Operand 2 converted to integer
Integer	Float	Operand 1 converted to floating–point
Float	Integer	Operand 2 converted to integer

10. You can also explicitly convert the data type of a value by using one of the following subroutines:

Result	**Subroutine**	**Comments**
Integer	**INTEGER**(*Boolean*)	TRUE–>1, FALSE–>0
Integer	**INTEGER**(*integer*)	
Integer	**INTEGER**(*float*)	Integer part of the floating–point number (truncate toward zero). This can result in overflow.
Float	**FLOAT**(*Boolean*)	
Float	**FLOAT**(*integer*)	Floating–point representation of an integer. There should not be any loss of precision.
Float	**FLOAT**(float)	

The following example shows a value section from a UIL Module containing compile–time expressions and data conversion subroutines:

```
VALUE
value outer_box_width: 200;
value outer_box_height: 250;
value box_size_ratio: 0.5;
value inner_box_width: integer(outer_box_width * box_size_ratio); va
lue inner_box_height: integer(outer_box_height
                                  * box_size_ratio);
value inner_box_x: (outer_box_width - inner_box_width) >> 1;
value inner_box_y: (outer_box_height - inner_box_height) / 2;
value type_field: 0;
value class_field: 16;
value type1: 1;
value type2: 2;
value type3: 3;
value class1: 1;
value class2: 2;
value class3: 3;
value combo1: (type1 << type_field) | (class3 << class_field);
```

Suggested Reading

Related Information

Using AIXwindows Resource Manager (MRM) Subroutines contains additional information about the MRM subroutines.

How to Create a User Interface with AIXwindows User Interface Language (UIL) and Resource Manager (MRM)

You can use the AIXwindows User Interface Language (UIL) and its Resource Manager (MRM) to create interactive user interfaces for your applications without the necessity of hard–coding each interface into your application source code. The following sections are associated with this process:

- Understanding the Structure of a User Interface Language (UIL) Module
- How to Create Specification Files Containing UIL Modules
- Using AIXwindows Resource Manager (MRM) Subroutines
- How to Create a User Interface at Run Time Using the AIXwindows Resource Manager (MRM)
- Using Icons in the AIXwindows User Interface Language (UIL)
- How to Compile an AIXwindows User Interface Language (UIL) Specification File to Generate a User Interface Definition (UID) File
- Using Recommended Coding Techniques in AIXwindows User Interface Language (UIL)
- How to Customize an AIXwindows Interface Using UIL and MRM
- Using the AIXwindows User Interface Language (UIL) Callable Interface Subroutine and the Symbol Table Dump Subroutine
- How to Create Your Own Widgets and Gadgets with the AIXwindows User Interface Language (UIL)

Notes:

1. All of the sample source code in this section is based on a demo application called **AIXburger**. Only relevant portions of the UIL Module for this application are shown.
2. The **AIXburger** demo application is designed to show as many different widgets, gadgets, and UIL coding techniques as possible, but it does not use every feature of AIXwindows UIL.

Understanding the Structure of a User Interface Language (UIL) Module

A User Interface Language (UIL) Module is a file that contains all of the UIL language elements required to create an interactive AIXwindows user interface for your application at run time. Each UIL Module is designed to be compiled separately before being combined with the application it supports. This modularity permits separate, yet parallel, development of interfaces and applications, from the initial prototyping stage through the final debugging.

Parallel development promotes coding efficiency and reduces the overall time required to accommodate code changes in the application and its interface.

To create a UIL Module you must first familiarize yourself with the standard values and objects defined in the AIXwindows Toolkit and UIL.

A UIL Module contains definitions of objects to be stored in a User Interface Definition (UID) file, which represents the compiled output of the UIL Compiler. A UIL Module consists of a module block, which contains the following sections:

- value
- procedure
- list
- object
- identifier

Any number of these sections can be included within a UIL Module. A UIL Module can also contain include directives, which can be placed anywhere in the module, except within a value, procedure, list, identifier or object section.

Note: As you read through the related examples, keep in mind that the case–insensitive mode is always in effect and that keywords are shown in uppercase letters to distinguish them from user–specified names shown in lowercase letters. This use of uppercase letters is not required in case–insensitive mode. In the case–sensitive mode, keywords must always be in lowercase letters.

UIL Module Name

The module name is the name by which this UIL Module is known in the UID file. This name is stored in the User Interface Description (UID) file for later use during the retrieval of resources by the MRM. This name is always stored in uppercase in the UID file.

UIL Module Structure

The structure of a UIL Module is as follows:

```
uil_module ::=
    MODULE module_name
        [ version–clause ]
        [ case–sensitivity–clause ]
        [ default–character–set–clause ]
        [ default–object–variant–clause ]
        { value–section
        | procedure–section
        | list–section
        | object–section
        | identifier–section
        | include–directive }...
    END MODULE ";"
version–clause ::=
    VERSION "=" character–expression
case–sensitivity–clause ::=
    NAMES "=" { CASE_SENSITIVE
                CASE_INSENSITIVE }
default–character–set–clause ::=
    CHARACTER_SET "=" char–set
default–object–variant–clause ::=
    OBJECTS "=" "{" object–type "=" WIDGET | GADGET ";" ... "}"
```

A Sample UIL Module

- Version Clause
- Case Sensitivity Clause
- Default Character Set Clause
- Default Object Variant Clause

The following example of a UIL Module indicates the contents of each element of the structure:

```
!+
! Sample UIL Module
!-
MODULE example        ! module name
VERSION = 'V1.0'      ! version
NAMES = CASE_INSENSITIVE     ! keywords and names are
                             ! not case sensitive
CHARACTER_SET = ISO_LATIN6   ! character set for compilation
                             ! is ISO_LATIN6
OBJECTS = { XmPushButton = GADGET; }     ! push buttons are
                                         ! gadgets by default
!+
! Declare the VALUES, PROCEDURES, LISTS,
! IDENTIFIERS, and OBJECTS here...
!-
END MODULE;
```

Version Clause

The version clause specifies the version number of the UIL Module. It is provided so that the programmer can verify that the correct version of a UIL Module is being accessed by MRM. The character expression that specifies the version can be up to 31 characters in length.

Case Sensitivity Clause

The case sensitivity clause indicates whether names are to be treated as case sensitive or case insensitive. The default is case insensitive.

The case sensitivity clause is the first clause in the module header. It precedes every statement that contains a name.

If names are case sensitive in a UIL Module, UIL keywords in that module must be in lowercase. Each name is stored in the UID file in the same case that it appears in the UIL Module. If names are case insensitive, keywords can be in uppercase, lowercase, or mixed case, and the uppercase equivalent of each name is stored in the UID file.

The following summary clarifies these basic rules:

Case Sensitivity	**Keyword Treatment in UIL Module**	**Name Treatment in UID File**
Case sensitive	Must be lowercase	Stored in UID file in same case as they appear in UIL Module
Case insensitive	Can be entered in lowercase, uppercase, or mixed case	Stored in uppercase in UIL module and UID file

Default Character Set Clause

The default character set clause specifies the default character set for string literals in the module. The use of the default character set clause is optional. If specified, the character set clause designates the character set used to interpret an extended string literal if a character set has not been specified for that literal. If you do not include the character set clause in the module header, the default character set for the compilation is ISO_LATIN1.

Default Object Variant Clause

A gadget is a windowless, somewhat simplified version of a widget that enhances application performance because it consumes fewer system resources. However, widget and gadget variants are interchangeable in many circumstances. The primary reason for the existence of both variants is the fact that, due to their relative simplicity, gadgets support only limited customization. The following types of user interface objects have both a widget and a gadget variant:

- Separators
- Push Buttons
- Toggle Buttons
- Cascade Buttons
- Labels

The UIL Compiler assumes you intend to use widgets rather than gadgets. To use gadgets in an application, explicitly specify to the UIL Compiler that you want that particular variant. Otherwise, the UIL Compiler defaults to its *widgets only* mode.

There are two ways to explicitly specify a gadget instead of a widget:

- Add the default object variant clause to the module header
- Add the keyword GADGET to each appropriate object declaration

By using the default object variant clause in the UIL Module header, you can declare all cascade buttons, labels, push buttons, separators, and toggle buttons, or any combination of these types, to be gadgets. The sample module shows only push buttons declared as gadgets.

To change these objects from one variant to the other, change the default object variant clause. For example, use a default object variant clause to declare that all push buttons should be made from gadgets. To change all push button objects from gadgets to widgets, remove the type **XmPushButton** gadget from the clause.

Because you do not specify the variant for imported objects (discussed in the next section), the variant (whether a widget or gadget) of imported objects is unknown until run time.

Value Section of a UIL Module

- Scope of Reference to Values and Objects
- Structure of a Value Section

Scope of References to Values and Objects

UIL values can have one of three levels of privacy as indicated by the following keywords:

- IMPORTED

Declared as a named resource in a UID file. Can be referenced by name in other UID files. MRM resolves this declaration with the corresponding exported declaration at application run time.

- EXPORTED

Declared as a named resource in a UID file. Can can be referenced by name in other UID files. When you define a value as exported, MRM looks outside the module in which the exported value is declared to get its value at run time

- PRIVATE

Not stored as a distinct resource in a UID file. Referenced only in the UIL Module containing the value declaration. Private values or objects are directly incorporated into every element in the UIL Module that references the declaration. Values and objects are private by default.

Structure of a Value Section

A value section consists of the keyword VALUE followed by a sequence of value declarations. It has the following syntax:

```
value–section ::=
    VALUE value–declaration...
value–declaration ::=
    value–name ":"
        { EXPORTED value–expression
        | PRIVATE value–expression
        | value–expression
        | IMPORTED value–type } ";"
```

A value declaration provides a way to name a value expression. The value name can be referred to by declarations that occur later in the UIL Module in any context where a value can be used. Values cannot be forward referenced; a value name must be declared before it can be referenced.

Value sections can include a keyword defining the scope of references to the value. The following value types are supported in UIL:

Value Type	**Description**
ANY	Prevents the UIL Compiler from checking the type of a parameter value.
ARGUMENT	Defines a value as a user–defined parameter.
BOOLEAN	Defines a value as TRUE (ON) or FALSE (OFF).
COLOR	Defines a value as a color.
COLOR_TABLE	Provides a device–independent way to define a set of colors (usually for a pixmap).
COMPOUND_STRING	Defines a value as a compound string.
FLOAT	Defines a value as a floating point literal.
FONT	Defines a value as a font.
FONT_TABLE	Defines a value as a sequence of font and character set pairs.
ICON	Describes a rectangular pixmap using a character to represent each pixel.
INTEGER_TABLE	Defines a value as an array of integers.
REASON	Defines a condition under which a widget is to call an application subroutine.
STRING	Defines a value as a null–terminated (ASCIZ) string.
STRING_TABLE	Defines a value as an array of compound strings.

TRANSLATION_TABLE	Defines an alternative set of events or actions for a widget.

The following declarations show how values are declared in UIL:

```
VALUE
  k_main              : EXPORTED 1;
  k_main_menu         : EXPORTED 2;
  k_main_command      : EXPORTED 3;
VALUE
  white               : IMPORTED COLOR;
  blue                : IMPORTED COLOR;
  arg_name            : PRIVATE 'new_argument_name';
VALUE
  main_prompt         : 'next command'; ! PRIVATE by default
  main_font           : IMPORTED FONT;
```

Note: Because the values k_main, k_main_menu, and k_main_command are defined as exported, you can use these values in another UIL Module as follows:

```
VALUE
  k_main          : IMPORTED integer;
```

Procedure Section of a UIL Module

- Structure of a Procedure Section
- Using a Procedure Section

Structure of a Procedure Section

A procedure section consists of the keyword PROCEDURE followed by a sequence of procedure declarations. It has the following syntax:

```
procedure–section ::=
    PROCEDURE procedure–declaration...
procedure–declaration ::=
    procedure–name
        [ formal–parameter–spec ]";"
formal–parameter–spec ::=
        "(" [ value–type ] ")" ";"
```

Using a Procedure Section

Use a procedure declaration to declare the following:

- A routine that can be used as a callback routine for a widget
- The creation subroutine for a user–defined widget

Reference a procedure name in declarations that occur later in the UIL Module in any appropriate context. Procedures cannot be forward referenced; you must declare a procedure name before you reference it. You cannot use a name you used in another context as a procedure name.

A procedure declaration provides a means of specifying that a parameter is to be passed to the corresponding callback routine at run time. This parameter is called the callback tag. You can specify the data type of the callback tag by putting the data type in parentheses following the procedure name. When you compile the module, the UIL Compiler checks that the parameter you specify in references to the procedure is of this type. The data type of the callback tag must be one of the following valid UIL data types and it must be declared in the indicated manner:

Data Type	Declaration Rules
No arguments	The UIL Compiler provides no parameter type or parameter count checking. You can supply any number of parameters in the procedure reference.
( )	Checks that the parameter count is zero.
(ANY)	Checks that the parameter count is 1. Does not check the parameter type. Use the ANY type to prevent type checking on procedure tags.
(type)	Checks for one parameter of the specified type. In the procedure declaration `PROCEDURE toggle_proc (integer);` the callback procedure named `toggle_proc` is passed an integer tag at run time. The UIL Compiler checks that the parameter specified in any reference to procedure `toggle_proc` is an integer.

Note: While it is possible to use any UIL data type to specify the type of tag in a procedure declaration, you must be able to represent that data type in the programming language you are using. Some data types (such as integer, boolean, and string) are common data types recognized by most programming languages. Other UIL data types (such as string tables) are more complicated and may require an appropriate corresponding data structure in the application in order to pass a tag of that type to a callback routine.

A procedure declaration can also specify the creation subroutine for a user–defined widget. In this case, no formal parameters are specified. The procedure is invoked with the standard three parameters passed to all AIXwindows widget creation subroutines.

The following declaration shows how to declare a procedure:

```
PROCEDURE
    app_help (INTEGER);
    app_destroy (INTEGER);
```

List Section of a UIL Module

- Structure of a List Section
- Arguments List
- Callbacks List
- Controls List
- Procedures List

Structure of a List Section

A list section consists of the keyword LIST followed by a sequence of list declarations. It has the following syntax:

```
list-section ::=
    LIST list-declaration...
list-declaration ::=
    list-name ":" list-definition ";"
list-definition ::=
    list-type list-spec
list-type ::=
    { ARGUMENTS
    | CONTROLS
    | CALLBACKS
    | PROCEDURES }
list-spec ::=
    { list-name
    | "{" list-entry... "}" }
list-entry ::=
    { list-definition
    | argument-list-entry
    | control-list-entry
    | callback-list-entry
    | procedure-list-entry } ";"
```

You use list sections to group together a set of parameters, controls (children), or callbacks for later use in the UIL Module. Lists can contain other lists, so that you can set up a hierarchy to clearly show which parameters, controls, and callbacks are common to which widgets. You cannot mix different types of lists; a list of a particular type cannot contain entries of a different list type or reference the name of a different list type.

A list name is always private to the UIL Module in which you declare the list. A list name cannot be stored as a named resource in a UID file.

There are four types of lists in UIL:

- Arguments list (the list type is ARGUMENTS)
- Callbacks list (the list type is CALLBACKS)
- Controls list (the list type is CONTROLS)
- Procedures list (the list type is PROCEDURES)

Arguments List

An arguments list defines the parameters to be specified when the creation subroutine for an object is called at run time. An arguments list also specifies the values for those parameters. Each entry in the arguments list has the following syntax:

```
argument-list-entry ::=
    argument-name "=" value-expression
```

The argument name must be either a built-in argument name or a user-defined argument name that is specified with the ARGUMENT subroutine. If you use a built-in argument name, the type of the value expression must match the allowable type for the argument.

If you use a built-in argument name as an arguments list entry in an object definition, the UIL Compiler checks the argument name to be sure that it is supported by the type of object you are defining. If the same argument name appears more than once in a given arguments list, the last entry that uses that argument name supersedes all previous entries with that name, and the UIL Compiler issues a diagnostic message.

The following arguments are coupled by the UIL Compiler.

Supported Argument	**Coupled Argument**
XmNitems	XmNitems_count.
XmNselectedItems	XmNselectedItemsCount.

When you specify one of the arguments in the left column, the UIL Compiler also sets the argument in the right column. The coupled argument is not available to you.

The following example shows how to declare an arguments list:

```
LIST
    default_size: ARGUMENTS {
        XmNheight = 500;
        XmNwidth = 700;
    };
    default_args: ARGUMENTS {
        ARGUMENTS default_size;
        XmNforeground = white;
        XmNbackground = blue;
    };
```

Callbacks List

Use a callbacks list to define the callback reasons to be processed by a particular widget at run time. Each callbacks list entry has the following syntax:

```
callback–list–entry ::=
    reason–name "=" procedure–reference
procedure–reference ::=
    PROCEDURE procedure–name
        [ "(" [ value–expression ] ";"
            | procedure–list–specification ]
```

For AIXwindows Toolkit widgets, the reason name must be a built–in reason name. For a user–defined widget, you can use a reason name that was previously specified using the REASON subroutine. If you use a built–in reason in an object definition, the UIL Compiler ensures that the reason is supported by the type of object you are defining.

If the same reason appears more than once in a callbacks list, the last entry referring to that name supersedes all previous entries using the same reason, and the UIL Compiler issues a diagnostic message.

You must have declared previously the procedure you refer to in a callbacks list entry. If you specify a named value for the procedure argument (callback tag), the data type of the value must match the type specified for the callback tag in the corresponding procedure declaration.

The following example demonstrates the declaration of a callbacks list:

```
LIST
    default_callbacks : CALLBACKS {
        XmNdestroyCallback = PROCEDURE app_destroy (k_main);
        XmNhelpCallback = PROCEDURE app_help (k_main);
    };
```

The following lines of pseudocode show the interface to the callback procedure:

```
    PROCEDURE procedure–name ( widget by reference,
                               tag by reference,
                               reason by reference
                               RETURNS: no–value);
```

The UIL Compiler produces a UID file rather than an object module (**.o**). Consequently, the binding of the UIL name to the address of the entry point to the procedure is not done by the loader, but is established at run time with the MRM subroutine **MrmRegisterNames**. Call this subroutine before fetching any objects, giving it both the UIL names and the procedure addresses of each callback. The name you register with MRM in the application program must match the name you specified for the procedure in the UIL Module.

Each callback procedure receives the following three parameters: The first two parameters have the same form for each callback. The form of the third parameter varies from object to object.

Parameter	**Description**
widget ID	The address of the data structure maintained by the AIXwindows Toolkit for this object instance.
Address of the value specified in callbacks procedure	The **XmNdestroyCallback** callback in the preceding example has `app_destroy` as its callback list for this procedure.
Address of the integer *k_main*	If you do not specify a parameter, the address is **NULL**.

Notes:

1. The structure of the third parameter must be obtained from the AIXwindows Toolkit reference material. The reason name you specified in the UIL Module is always the first field in this structure.
2. The AIXwindows Toolkit encoding of UIL reasons is contained in the file **XmAppl.uil**.

You can specify multiple procedures for each callback reason in UIL by defining the procedures as a type of list. Just as with other list types, you can define procedures lists either in line, or in a LIST section and referenced by name. If you define a reason more than once (for example, when the reason is defined both in a referenced procedures list and in the callbacks list for the object), previous definitions are overridden by the latest definition.

The following example shows how to specify multiple procedures per callback reason in an object declaration for an **XmPushButton** gadget:

```
OBJECT m_quit_button: XmPushButton {
    ARGUMENTS {
                .
                .
                .
              };
    CALLBACKS {
        XmNactivateCallback = PROCEDURES
              {
                quit_proc ('normal exit');
                shutdown ();
              };
    };
};
```

In the example, the **quit_proc** subroutine and the **shutdown** subroutine are both called in response to the XmNactivateCallback callback reason.

Controls List

A controls list defines which objects are *children* (managed objects) of a particular *parent* (managing object). Each entry in a controls list has the following syntax:

```
control–list–entry ::=
    [ MANAGED | UNMANAGED ] object–definition
```

If you specify the keyword MANAGED, the object is created and managed at run time. If you specify UNMANAGED, the object is created at run time but is not managed by another object. All objects are managed by default.

Note: Unlike the arguments list and the callbacks list, a controls list entry that is identical to a previous entry does not supersede the previous entry. At run time, each controls list entry causes a child to be created when the parent is created. If the same object definition is used for multiple children, multiple instances of the child are created at run time.

The following example shows how to declare a controls list:

```
LIST
    default_main_controls : CONTROLS {
        XmCommand              main_command;
        XmMenuBar              main_menu;
        UNMANAGED XmList       file_menu;
        UNMANAGED XmOptionMenu edit_menu;
    };
```

Procedures List

You can specify multiple procedures for a callback reason in UIL by defining a procedures list. Just as with other list types, lists can be defined inline or in a list section in which they are referenced by name.

If you define a reason more than once (for example, when the reason is defined in a referenced procedures list and in the callbacks list for the object), previous definitions are overridden by the latest definition.

The syntax for a procedures list is as follows:

```
procedure–list–specification ::=
    PROCEDURES procedure–list–spec
procedure–list–spec ::=
    { procedure–list–name
    | "{" [procedure–list–clause...] "}" }
procedure–list–clause ::=
    { procedure–list–specification | procedure–list–ref }
procedure–list–ref ::=
    procedure–list–name [ "(" [ tag–value–expression ] ")" ]
callback–list–entry ::=
    reason–name "=" procedure–list–spec
```

Object Section of a UIL Module

- Structure of an Object Section
- Declaration of an Object
- Specifying Object Variants (Gadgets) in the Module Header
- Specifying the Object Variant in the Object Declaration

The Structure of an Object Section

An object section consists of the keyword OBJECT followed by a sequence of object declarations. It has the following syntax:

```
object-section ::=
    OBJECT object-declaration...
object-declaration ::=
    object-name ":"
        { EXPORTED object-definition
        | PRIVATE object-definition
        | object-definition
        | IMPORTED object-type } ";"
object-definition ::=
    object-type [ procedure-reference ] object-spec
object-spec ::=
    { object-name [WIDGET | GADGET]
    | "{" list-definition... "}" }
```

Use an object declaration to define the objects stored in the UID file. You can reference the object name in declarations that occur elsewhere in the UIL Module in any context in which an object name can be used (for example, in a controls list, as a symbolic reference to a widget ID, or as the tag_value parameter for a callback procedure).

You can declare an object name after you reference it. All references to an object name must be consistent with the type of the object, as specified in the object declaration.

You can specify an object as exported, imported, or private.

The object definition contains a sequence of lists that define the parameters, hierarchy, and callbacks for the widget. You can specify only one list of each type for an object. To specify more than one list of parameters, controls, or callbacks, combine the lists within a single object list, as demonstrated by the following example of an object definition:

```
object some_widget:
    arguments {
        arguments_list1;
        arguments_list2;
    };
```

Note: In the preceding example, arguments_list1 and arguments_list2 are lists of arguments previously defined in a LIST section.

When you declare a user–defined widget, you must include a reference to the widget creation subroutine for the user–defined widget.

Declaration of an Object

The following example shows how to declare an object:

```
OBJECT
   app_main : EXPORTED XmMainWindow {
     ARGUMENTS {
       ARGUMENTS default_args;
       XmNheight = 1000;
       XmNwidth = 800;
     };
     CALLBACKS default_callbacks;
     CONTROLS {
       XmMenuBar main_menu;
       user_defined my_object;
     };
};
```

Specifying Object Variants (Gadgets) in the Module Header

You can include a default object variant clause in the module header. This specifies the default variant of each type of object defined in the UIL Module. The object can be any user interface object that has a gadget variant, including the following objects:

- **XmCascadeButton**
- **XmLabel**
- **XmPushButton**
- **XmSeparator**
- **XmToggleButton**

Note: If you specify an object that does not have a gadget variant, the UIL Compiler issues a diagnostic message.

When you include an object in the default object variant clause, all objects of the same class in the interface default to the variant you specified in the clause. For example, the following default object variant clause specifies that all **XmPushButton** objects in the module are gadgets:

```
OBJECTS = { XmPushButton = GADGET; }
```

Note: If you attempt to specify an object type more than once in the default object variant clause, the UIL Compiler issues a diagnostic message.

You can override a specification in a default object variant clause in the two following ways:

- You can **implicitly** override an existing specification by omitting the default object variant clause (or by omitting an object class from the clause) because the UIL Compiler automatically defaults to *widget* in this circumstance.
- You can **explicitly** override an existing specification by using the keyword WIDGET or GADGET as a resource in the appropriate object declaration. (Include the keyword between the object type and the left brace of the object specification.) These keywords can specify the object type or override the default variant for the object type.

Specifying the Object Variant in the Object Declaration

The syntax of the object declaration is as follows:

```
OBJECT
object-name : object-type GADGET | WIDGET {
.
.
.
};
```

The object type can be any user interface object that has a gadget variant, including the following objects:

- **XmCascadeButton**
- **XmLabel**
- **XmSeparator**
- **XmPushButton**
- **XmToggleButton**

If you specify any other object type as a gadget, the UIL Compiler issues a diagnostic message.

The following example shows how to specify gadgets:

```
MODULE sample
    NAMES = case_insensitive
    OBJECTS =
        { XmSeparator = GADGET; XmPushButton = WIDGET; }
    OBJECT
        a_button : XmPushButton GADGET {
            ARGUMENTS { XmNlabelString = 'choice a'; };
        };
        a_menu : XmPulldownMenu {
            ARGUMENTS { XmNborderWidth = 2; };
            CONTROLS {
             XmPushButton a_button;
             XmSeparator GADGET {};
             XmPushButton {
                ARGUMENTS { XmNlabelString = 'choice b'; };
            };
            XmSeparator WIDGET {};
            XmPushButton c_button;
            XmSeparator {};
        };
    };
    c_button : XmPushButton GADGET {
        ARGUMENTS { XmNlabelString = 'choice c'; };
    };
END MODULE;
```

In the preceding example, the default object variant clause specifies that all separator objects are gadgets and all push button objects are widgets, unless overridden. The object a_button is explicitly specified as a gadget. Object a_menu defaults to a widget.

Note: Notice that the reference to *a_button* in the controls list of a_menu refers to the *a_button* gadget. Include the gadget attribute only on the declaration of *a_button*, not on each reference to *a_button*. The same holds true for *c_button*, even though the reference to c_*button* in the controls list for *a_menu* is a forward reference.

The unnamed push button definition in the controls list for a_menu is a widget because of the default object variant clause; the last separator is a gadget for the same reason.

Specify the GADGET or WIDGET keyword only in the declaration of an object, not when you reference the object. Do not specify the GADGET or WIDGET keyword for a user–defined object; user–defined objects are always widgets.

Identifier Section of a UIL Module

Structure of an Identifier Section

The identifier section allows you to define an identifier, which is a mechanism that, at run time, ensures the binding of data to names in a UIL Module. The identifier section consists of the reserved keyword IDENTIFIER followed by a list of names separated by single semicolons. These names can later appear in the UIL Module as parameter values for an individual object or as the tag value of a callback procedure. At run time, you use the MRM **MrmRegisterNames** subroutine to bind the identifer name to the address of the data associated with the identifier.

UIL has a single name space. Do not use a name previously assigned to a value, object, or procedure as an identifier name.

The following example shows how to use an identifier section in a UIL Module:

```
IDENTIFIER
    my_x_id;
    my_y_id;
    my_destroy_id;
```

The UIL Compiler does not do any type checking on the use of identifiers in a UIL Module. Unlike a UIL value, an identifier does not have a UIL type associated with it. Regardless of what particular type a widget parameter or callback procedure tag is defined to be, you can use an identifier in that context instead of a value of the corresponding type.

To reference these identifier names in a UIL Module, you use the name of the identifier wherever you want its value to be used. Identifiers can be referenced in any context in which a value can be referenced in UIL, although identifiers are primarily used as callback procedure tags and widget parameter values.

The UIL Module in the following example, the identifiers `my_x_id` and `my_y_id`, are used as parameter values for the main window widget, `my_main`. The position of the main window widget may depend on the screen size of the terminal on which the interface is displayed. Using identifiers, you can provide the values of the XmNx and XmNy parameters at run time. The identifier named `my_destroy_id` is specified as the tag to the callback procedure `my_destroy_callback`. In an application, you could allocate a data structure and use the `my_destroy_id` identifier to store the address of the data structure. When the **XmNdestroyCallback** reason occurs, the data structure is passed as the tag to procedure `my_destroy_callback`.

```
MODULE id_example
    NAMES = CASE_INSENSITIVE
        IDENTIFIER
            my_x_id;
            my_y_id;
            my_destroy_id;
        PROCEDURE
            my_destroy_callback ( STRING );
        OBJECT my_main : XmMainWindow {
            ARGUMENTS {
                XmNx = my_x_id;
                XmNy = my_y_id;
            };
            CALLBACKS {
                XmNdestroyCallback = PROCEDURE my_destroy_callback
                    ( my_destroy_id );
            };
    };
END MODULE;
```

Include Directive

The include directive incorporates the contents of a specified file into a UIL Module. This mechanism allows several UIL Modules to share common definitions. The syntax for the include directive is as follows:

```
INCLUDE FILE character–expression ";"
```

The file specified in the include directive is called an include file. The UIL Compiler replaces the include directive with the contents of the include file and processes it as if these contents had appeared in the current UIL source file.

You can nest include files; that is, an include file can contain include directives. The UIL Compiler can process up to 20 references (including the file containing the UIL Module). Therefore, you can include up to 19 files in a single UIL Module, including nested files. Each time a file is opened counts as a reference, so including the same file twice counts as two references.

The character expression is a file specification that identifies the file to be included. The rules for finding the specified file are similar to the rules for finding header, or .h, files using the include directive, #include, with a quoted string in C.

If you do not supply a directory, the UIL Compiler searches for the include file in the directory of the main source file; if the UIL Compiler does not find the include file there, the UIL Compiler looks in the same directory as the source file. If you supply a directory, the UIL Compiler searches only that directory for the file.

The following example shows how to use the include directive:

```
INCLUDE FILE 'constants';
```

Parameter Definitions for Constraint Widgets

The AIXwindows Toolkit and the Enhanced X–Windows Intrinsics Toolkit support the **Constraint** widget class. **Constraint** widget parents grant parameters to their widget children beyond the parameters normally available to widget children. Unlike parameters that define the resources of a particular widget, constraint parameters are used exclusively to define the resources of the children of a particular widget. For example, the **XmForm** widget class inherits from the **Constraint** widget class the ability to grant its children a set of parameters for controlling their position within their container objects.

To supply arguments to the children of a **Constraint** widget, reference the arguments in the arguments list for a child of the **Constraint** widget, as shown in the following example:

```
OBJECT
    my_form : XmForm {
        arguments {
            XmNx = 70;
            XmNy = 20;
            XmNrows = 35;
        };
        CONTROLS {
            XmPushButton {
                ARGUMENTS {
                    XmNleftAttachment = XmATTACH_WIDGET;
                    XmNleftOffset = 10;
                };
            };
        };
    };
```

Symbolic References to Widget IDs

The AIXwindows Toolkit identifies each widget by its widget ID. The UIL Compiler, on the other hand, identifies objects by name because widget IDs are defined only at run time and are therefore unavailable for use in a UIL Module. The UIL Compiler resolves this dilemma by allowing you to reference a widget ID symbolically by using its name.

Whenever you supply a parameter that requires a widget ID, use the UIL name of that widget (and its object type) as the parameter. For example, the **XmForm** widget has a parameter that references a top widget. Give the type and name of the object you want to use for the top attachment as the value for this parameter.

The widget name you reference must be a descendant of the widget being fetched for MRM to find the referenced widget; you cannot reference an arbitrary widget. MRM checks this at run time. For example, a practical use of symbolic references is to specify the default push button (in a bulletin board or radio box).

The following example shows how to use a symbolic reference:

```
MODULE
    NAMES = CASE_INSENSITIVE
        OBJECT my_dialog_box : XmBulletinBoard {
            ARGUMENTS {
                XmNdefaultButton = XmPushButton yes_button;
            };
            CONTROLS {
                XmPushButton yes_button;
                XmPushButton no_button;
            };
        };
        OBJECT yes_button : XmPushButton {
            ARGUMENTS {
                XmNlabelString = 'yes';
            };
        };
        OBJECT no_button : XmPushButton {
            ARGUMENTS {
                XmNlabelString = 'no';
            };
        };
END MODULE;
```

In the preceding example, two **XmPushButton** widgets are defined and named (`yes_button` and `no_button`). In the definition of the **XmBulletinBoard** widget, the name `yes_button` is given as the value for the XmNdefaultButton argument. This parameter usually accepts a widget ID. When you use a symbolic reference (the object type and name of the `yes_button` widget) as the value for the XmNdefaultButton argument, MRM substitutes the widget ID of the `yes_button` push button for its name at run time.

The UIL Compiler contains built–in tables that indicate where symbolic referencing of widget IDs is acceptable by showing the term "object reference" as the data type of the argument.

How to Create Specification Files Containing UIL Modules

Prerequisite Tasks or Conditions

1. Ensure that the UIL Compiler and all the include files and other header files listed in your source code are available and accessible.

Procedure

To create a UIL Specification File containing one or more UIL Modules, complete the following tasks in the order shown:

1. Create and name a UIL Specification File. The name of each UIL Specification File must end with the characters **.uil**. Each UIL Specification File you create and name should contain one UIL Module block (or a legal portion of a Module block) that consists of a series of value, identifier, procedure, list, and object sections.

   a. A UIL Module can contain any number of these sections. The number of files you use to specify the interface depends on the following factors:

      - The complexity of your application
      - The need for interface variations (for example, English and French versions)
      - The size of the development project team. (The UIL Module can be distributed over several files to avoid coding bottlenecks.)

   Note: UIL also supports an include directive that allows you to include the contents of another file in your UIL Module. You can use an include directive to specify one or more complete sections. You can place the include directive anywhere in a UIL specification file where a section would be valid. However, you cannot specify part of a section using an include directive. You can also use the include directive to obtain access to the supplied UIL constants that are useful for specifying values for parameters such as **XmNdialogStyle** and **XmNalignment**.

2. Specify the user interface in a UIL Module. The following example shows the overall structure of a UIL Module:

```
!+
!    Sample UIL Module
!-
Module example    ! Module name
!+
!    Place Module header clauses here.
!-
!+
!    Declare the VALUES, IDENTIFIERS, PROCEDURES, LISTS, and
!    OBJECTS here.
!-
end Module;
```

3. For each UIL Module, declare the module (begin a Module block) by naming the module and making Module–wide specifications using the following (optional) Module header clauses:

Clause	**Purpose**	**Default**	**Example**
Version	Allows you to ensure the correct version of the UIL Module is being used.	None	version = 'v1.0'
Case sensitivity	Specifies whether names in the UIL Module are case sensitive or not.	Case insensitive	names = case_sensitive
Default character set	Specifies the default character set for string literals in the compiled UIL Module.	ISO_LATIN1	character_set = iso_latin6
Object variant	Specifies the default variant of objects defined in the module on a type–by–type basis.	Widget	objects = { `XmSeparator` = gadget; `XmPushButton` = widget; }

The following example shows the module declaration for the **AIXburger** UIL Module:

```
(1)
  Module aixburger_demo
(2)
  version = 'v1.0'
(3)
  names = case_sensitive
(4)
  objects = {
            XmSeparator = gadget ;
            XmLabel = gadget ;
            XmPushButton = gadget ;
            XmToggleButton = gadget ;
              }
(5)
  include file 'XmAppl.uil';
```

The following comments apply to the preceding sample code:

General Comment:	The name you specify in the module declaration is stored in the UID file when you compile the UIL Specification File containing the module. The module declaration for **AIXburger** specifies the following:
Code Segment #1:	MRM will identify the **AIXburger** interface by the name *aixburger_demo.*
Code Segment #2:	This is the first version of this Module.
Code Segment #3:	Names are case sensitive. If you specify that names are case sensitive in your UIL Module, you must put UIL keywords in lowercase letters. Do not use reserved keywords as names in a UIL Module.

Code Segment #4: All **XmSeparator**, **XmLabel**, **XmPushButton**, and **XmToggleButton** objects are gadgets unless overridden in specific object declarations. All other types of objects are widgets.

Code Segment #5: Your search path (defined to the UIL Compiler with the `-I` command line option) must include the **XmAppl.uil** file (described in detail in **step 4**).

4. Include the supplied UIL **XmAppl.uil** constants file.

   The include file containing definitions of UIL constants is named **XmAppl.uil**. When you compile the UIL Module, the UIL Compiler replaces any include directives with the contents of the specified file. The UIL constants file must be included before its contents are referenced, so include the UIL constants file immediately after the UIL Module header.

   The UIL Module for the **AIXburger** application makes use of some of the constants defined in the **XmAppl.uil** include file, including the XmDIALOG_MODELESS and XmVERTCIAL constants used in the following example:

```
object        ! The control panel. All order entry
              ! is done through this bulletin board dialog.
    control_box : XmBulletinBoardDialog {
    parameters {
        XmNdialogTitle = XmDIALOG_MODELESS;
        XmNx = XmNbackground = lightblue;
    };
    controls {
              ! Some labels and decoration.
        XmLabel        burger_label;
        XmLabel        fries_label;
        XmLabel        drink_label;
        XmSeparator    {parameters    {
                           XmNx = 220;
                           XmNy = 20;
                           XmNunitType = XmPIXELS;
                           XmNorientation = XmVERTICAL;
                           XmNheight = 180; };};
              };
```

5. Declare the callback procedures referenced in the object declarations.

   Use a procedure declaration to declare a subroutine that can be used as a callback procedure for an object. You can reference the procedure name in object declarations that occur later in the UIL Module.

   Callback procedures must be defined to accept the following parameters:

   - The widget identifier of the widget that triggers the callback
   - The callback data structure (which is unique to each widget).
   - A tag for user–defined information

   Note: The widget identifier and callback structure parameters are under the control of the AIXwindows Toolkit, but the tag is under the control of your application.

   In a UIL Module, you can specify the data type of the tag to be passed to the corresponding callback procedure at run time by putting the data type in parentheses following the procedure name. When you compile the module, the UIL Compiler checks that the parameter you specify in references to the procedure is of the indicated type. The data type of the tag must be one of the valid UIL types.

For example, the callback procedure named `toggle_proc` in the following procedure declaration is passed an integer tag at run time, so the UIL Compiler checks that the parameter specified in any reference to the `toggle_proc` procedure is an integer:

```
PROCEDURE
    toggle_proc (integer);
```

Note: While it is possible to use any UIL data type to specify the type of a tag in a procedure declaration, you must be able to represent that data type in the high–level language in which you write your application program. Some data types (such as integer, Boolean, and string) are common data types recognized by most programming languages. Other UIL data types (such as string tables) are more complex and may require you to set up an appropriate corresponding data structure in the application to pass a tag of that type to a callback procedure.

UIL Compiler Rules for Checking Parameter Type and Count

The UIL Compiler uses the following rules to check the parameter type and parameter count:

Declaration Type	Rule for Checking Parameter Type and Count
No parameters	No parameter type or parameter count checking. You can supply no parameters or one parameter in the procedure reference.
( )	Checks that the parameter count is zero.
(**any**)	Checks that the parameter count is 1. Does not check the parameter type. Use `any` to prevent type checking on procedure tags.
(*value_type*)	Checks for one parameter of the specified value type.

The UIL Compiler uses these rules to ensure that each parameter type and parameter count corresponds to the way you declared each procedure.

The following example shows that all procedures in the **AIXburger** UIL Module specify that parameter type and parameter count are to be checked when the UIL Module is compiled:

```
procedure
    toggle_proc         (integer);
    activate_proc       (integer);
    create_proc         (integer);
    scale_proc          (integer);
    list_proc           (integer);
    quit_proc           (string);
    show_hide_proc      (integer);
    pull_proc           (integer);
```

You can also use a procedure declaration to specify the creation subroutine for a user–defined widget. In this case, you must not specify any parameters. The procedure is invoked with the following three parameters that are passed to all widget creation subroutines:

- Widget ID
- Tag
- Callback structure unique to the calling object

6. Declare the values (integers, strings, colors, and so forth) that you intend to use in the object declarations.

   A value declaration is a way of giving a name to a value expression. The value name can be referenced by declarations that occur later in the UIL Module in any context where a value can be used. Values must have been previously declared before you refer to them.

   Note: Use meaningful names for values to help you recall their purpose easily.

   Data Types for UIL Values

   any
   argument
   asciz_table
   Boolean
   color
   color_table
   compound_string
   float
   font
   font_table
   pixmap
   integer
   reason
   string
   string_table
   translation_table
   integer_table

   You can control whether values are local to the UIL Module or globally accessible by MRM by specifying one of the following keywords in the value declaration:

   - EXPORTED
   - IMPORTED
   - PRIVATE

   The **AIXburger** application makes use of several kinds of values, as shown in the following examples. There is a separate value section for each type of value to make it easier to find the value declaration during debugging.

   The following value types are supported by UIL:

   - **Integer Values**

   Integer values are defined together in a single value section of the **AIXburger** UIL Module. These integers are used as tags in the callback procedures. A tag provides information to the callback procedure concerning the circumstances under which the procedure is being called. The following sample code shows a segment of this value section:

```
value
    k_create_order       : 1;
    k_order_pdme         : 2;
    k_file_pdme          : 3;
    k_edit_pdme          : 4;
    k_nyi                : 5;
    k_apply              : 6;
    k_dismiss            : 7;
    k_noapply            : 8;
    k_cancel_order       : 9;
    k_submit_order       : 10;
    k_order_box          : 11;
    k_burger_rare        : 12;
    k_burger_medium      : 13;
    k_burger_well        : 14;
    k_burger_ketchup     : 15;
    k_burger_mustard     : 16;
    k_burger_onion       : 17;
    k_burger_mayo        : 18;
    k_burger_pickle      : 19;
    k_burger_quantity    : 20;
```

- **String Values**

A value section can contain string value declarations. These strings are the labels for the various widgets used in the interface. Using values for widget labels rather than hard–coding the labels into the specification makes it easier to modify the interface (for example, changing the national language from English to German). Putting all label definitions together at the beginning of the UIL Module makes it easier to find a label if you want to change it later. Also, a string resource declared as a value can be shared by many objects, thereby reducing the size of the UID file.

The following sample code shows the **AIXburger** value section containing string value declarations:

```
  value
      k_aixburger_title                 : "AIXburger
                                        :      Order–Entry Box";
      k_nyi_label_text                  : "Feature is not yet
                                        :          implemented";

      k_file_label_text                 : "File";
          k_quit_label_text             : "Quit";
(1)
      k_edit_label_text                 : "Edit";
(2)
          k_cut_dot_label_text          : "Cut";
          k_copy_dot_label_text         : "Copy";
          k_paste_dot_label_text        : "Paste";
          k_clear_dot_label_text        : "Clear";
          k_select_all_label_text       : "Select All";
(3)
     k_order_label_text                : "Order";

(4)
          k_show_controls_label_text    : "Show Controls...";
          k_cancel_order_label_text     : "Cancel Order";
          k_submit_order_label_text     : "Submit Order";
```

The following comments apply to the preceding sample code:

General Comment: Because there is no default character set specified in the module header and the individual string values do not specify a character set, the default character set associated with all these compound strings is **ISO_LATIN1**.

Code Segment #1:	The indentation shown is not required but improves the readability of the UIL Module by giving an indication of the widget tree. For example, the widgets labeled Cut, Copy, Paste, Clear, and Select All are children of the widget labeled Edit.
Code Segment #2:	All the string values in this value section (with the exception of *k_order_label_text*) are used as labels. Although the strings are declared as null–terminated strings, the UIL Compiler automatically converts these strings to compound strings because the **XmNlabelString** parameter requires a compound string value.
Code Segment #3:	The *k_order_label_text* string value is used to define a parameter for the text widget. Since this widget does not accept compound strings, the value for *k_order_label_text* must be a null–terminated string.
Code Segment #4:	By convention, a label followed by an ellipsis (...) indicates that an AIXwindows Toolkit **XmBulletinBoard** widget appears when the object bearing this label is selected.

- **String Table Values**

A string table is a convenient way to express a table of strings. Some widgets require a string table parameter (such as the list widget that is used for drink selection in the **AIXburger** application).

The following example shows the definition of a string table value in **AIXburger**:

```
value
    k_drinks_label_text     : "Drinks";
        k_0_label_text      : '0';
        k_drink_list_text   :
            string_table ('Apple Juice', 'Orange Juice',
              'Grape Juice', 'Cola', 'Punch', 'Root beer',
              'Water', 'Ginger Ale', 'Milk', 'Coffee', 'Tea');
        k_drink_list_select : string_table('Apple Juice');
```

Notes:

1. The labels for the types of drinks are elements of the string table named *k_drink_list_text*. Apple Juice is a single element in the string table named *k_drink_list_select*. This value is passed to the *drink_list_box* widget to show apple juice as the default drink selection.
2. The UIL Compiler automatically converts the strings in a string table to compound strings, regardless of whether the strings are delimited by double or single quotation marks.

- **Font Values**

Use the **FONT** subroutine to declare a UIL value as a font.

The following example shows the declaration of a font value in the **AIXburger** UIL Module:

Note: This value is used later as the value for the **XmNfontList** resource of the *apply_button*, *can_button*, and *dismiss_button* **XmPushButton** widgets.

```
value
    k_button_font :
        font('—ADOBE—Courier—Bold—R—Normal——
            14—140—75—75—M—90—ISO8859—1');
```

The UIL Compiler converts a font to a font table when the font value is used to specify a parameter that requires a font table value.

Font names are server–dependent. If you specify a font name that is not defined on your server, the system issues a warning and uses the default font.

- **Color Values**

  By using the **COLOR** subroutine, you can designate a string as specifying a color and then use that string for parameters requiring a color value. The optional keywords FOREGROUND and BACKGROUND identify how the color is to be displayed on a monochrome device.

  The following example shows the value section in the **AIXburger** Module containing color declarations:

```
value
    yellow        : color('yellow', XmNforeground);
    red           : color('red', XmNbackground);
    green         : color('green', XmNforeground);
    magenta       : color('magenta', XmNbackground);
    gold          : color('gold', XmNforeground);
    lightblue     : color('lightblue', XmNbackground);
```

- **Pixmap Values**

  Pixmap values let you specify labels that are graphic images rather than text strings. Pixmap values are not directly supported by UIL. Instead, UIL supports icons, a simplified form of pixmap (which you define directly in UIL), or Xbitmap files (which you create outside UIL).

  You can generate pixmaps in UIL in the following ways:

  – Define an icon inline using the **ICON** subroutine (and, optionally, use the **COLOR_TABLE** subroutine to specify colors for the icon). Use a character to describe each pixel in the icon.

  – Use the **XBITMAPFILE** subroutine, specifying the name of an Xbitmap file that you created outside UIL to be used as the pixmap value.

  The colors you specify when defining a color table must have been previously defined with the **COLOR** subroutine. Color tables must be private because the UIL Compiler must be able to interpret their contents at compilation time to construct an icon. The colors within a color table, however, can be imported, exported, or private.

  The following example shows the value section in the **AIXburger** Module containing a color table declaration:

```
value
    button_ct : color_table(yellow='o', red='.',
                            background color=' ');
```

The following example shows how the *button_ct* color table is used to specify an icon pixmap. Referring to the color table shown in the previous example, each lowercase **o** in the icon definition is replaced with the color yellow, and each period (.) is replaced with the color red. Whatever color is defined as the background color when the application is run replaces the spaces:

Note: In UIL, if you define a parameter of type **pixmap**, you should specify an icon or an Xbitmap file as its value. For example, the icon defined in the following example is given as the value of the label on the drink quantity **XmPushButton** widget:

```
value

    drink_up_icon: icon(color_table=button_ct,
        '                        ',
        ' ..........oo.......... ',
        ' .........oooo......... ',
        ' ........oooooo........ ',
        ' .......oo....oo....... ',
        ' ......oo......oo...... ',
        ' .....oo........oo..... ',
        ' ....oo..........oo.... ',
        ' ...oo............oo... ',
        ' ..oo..............oo.. ',
        ' .oo................oo. ',
        ' oooooooooooooooooooooo ',
        ' oooooooooooooooooooooo ',
        ' .........oooo......... ',
        ' .........oooo......... ',
        ' .........oooo......... ',
        ' .........oooo......... ',
        ' .........oooo......... ',
        ' .........oooo......... ',
        '                        ');
```

Each row in the icon must contain the same number of pixels and therefore must contain the same number of characters. The height of the icon is dictated by the number of rows. For example, the arrow icon defined above is 24 pixels wide and 20 pixels tall. (The rows of spaces at the top and bottom of the pixmap and the spaces at the start and end of each row are included in this count and are defined as the background color in the *button_ct* color table.

A default color table is used if you omit the color table parameter from the **ICON** subroutine. The definition of the default color table is as follows:

```
color_table( background color = ' ', foreground color = '*');
```

You can specify icons as exported, imported, or private.

3. Declare the interface objects (widgets and gadgets)

   Use an object declaration to define each instance of a widget or gadget to be stored in the UID file. You can reference the object name in declarations that occur elsewhere in the UIL Module, usually to specify one object as a child of another object. Some widgets accept a widget name as a parameter. This use of a widget name is called a symbolic reference to a widget identifier. You can also use a widget name as the *tag_value* parameter to a callback subroutine.

   The object declaration contains a sequence of lists that define the resources (attributes), children, and callback subroutines for the object. You can specify only one list of each type for an object.

   Objects can be forward referenced; that is, you can declare an object name after you refer to it. This is useful for declaring a parent widget first, followed by the declarations for all its widget children—the declaration of the parent includes a list of the names of its children. In this way the structure of your UIL Module resembles the widget tree of your interface.

All references to an object name must be consistent with the type you specified when you declared the object. As with values, you can specify an object as exported, imported, or private.

The following example shows how the *file_menu* widget is declared in the **AIXburger** UIL Module.

4.
```
object
    file_menu : XmPulldownMenu {
        controls {
        XmPushButton m_quit_button;
        };
    callbacks {
        MrmcreateCallback = procedure create_proc (k_file_menu);
        };
  };
```

Note: The objects and values in this example have been given meaningful names (for example, *file_menu* and *k_file_label_text*) to remind you of the purpose of the object or value in the user interface.

Object declarations generally consist of the following three parts:

- **Arguments List** (specifies resources)

  Use an arguments list to specify the resources of an object. An arguments list defines the parameters to be specified in the *override_arglist* parameter when the creation subroutine for a particular object is called at run time. An arguments list also specifies the values that these parameters are to have. Identify an arguments list for the UIL Compiler by using the keyword ARGUMENTS.

  Each entry in the list consists of a parameter name and a parameter value. In the previous example the **XmNlabelString** parameter for the *file_menu* pull–down menu is defined as *k_file_label_text*. The value *k_file_label_text* is a compound string defined in a value section at the beginning of the module.

  If you use the same parameter name more than once in an arguments list, the last entry supersedes all previous entries and the compiler issues a message.

- **Controls List** (specifies children widgets)

  Use a controls list to define which widgets are children of, or controlled by, a particular widget. The controls lists for all the widgets in a UIL Module define the widget tree for an interface. If you specify that a child is to be managed (which is the default), at run time the widget is created and managed; if you specify that the child is to be unmanaged at creation (by including the keyword UNMANAGED in the controls list entry), the widget is only created. You identify a controls list to the UIL Compiler by using the keyword CONTROLS.

  In the following example, the bulletin board dialog called *control_box* is a top–level composite widget having a variety of widgets as children:

```
object    ! The control panel. All order entry
          ! is done through this dialog box.
    XmBulletinBoardDialog {
        arguments {
          XmNdialogTitle = k_aixburger_title;
          XmNdialogStyle = XmDIALOG_MODELESS;
          XmNx = 600;
          XmNy = 200;
          XmNmarginWidth = 20;
          XmNbackground = lightblue;
        };
        controls {
                       ! Some labels and decoration.
          XmLabel     burger_label;
          XmLabel     fries_label;
          XmLabel     drink_label;
          XmSeparator {parameters {
                         XmNx = 220;
                         XmNy = 20;
                         XmNunitType = XmPIXELS;
                         XmNorientation = XmVERTICAL;
                         XmNheight = 180; };};
          XmSeparator {parameters {
                         XmNx = 410;
                         XmNy = 20;
                         XmNunitType = XmPIXELS;
                         XmNorientation = XmVERTICAL;
                         XmNheight = 180; };};
          XmRowColumn button_box; ! Command push buttons inside
                                  ! a menu across the bottom.
                                  ! For the hamburger and drink
                                  ! entry we use a different
                                  ! mechanism to demonstrate
                                  ! various widgets and
                                  ! techniques. Hamburger
                                  ! 'doneness' uses a radio box
                                  ! because although it is a '1
                                  ! of N' type of entry, one
                                  ! (and only one) entry is
                                  ! allowed.
          XmRadioBox burger_doneness_box;
            .
            .
            .
```

The following comments apply to the preceding sample code:

General Comment:	Some of the children widgets in the preceding example are also composite widgets, having children of their own. For example, the *button_box* and *burger_doneness_box* widgets are declared later on in the UIL Module, and each of these has its own controls list.

General Comment: The separators in the preceding example are defined locally in the controls list for *control_box* rather than in object sections of their own. As a result, the separators do not have names and cannot be referenced by other objects in this UIL Module. However, the local definitions make it easier to tell that the separators are used only by the *control_box* widget. When you define an object locally, you do not need to create an artificial name for that object.

Unlike the arguments list (and the callbacks list, described in the following section), when you specify the same widget in a controls list more than once, MRM creates multiple instances of the widget at run time when it creates the parent widget.

- **Callbacks List** (specifies callback reasons)

  Use a callbacks list to define which callback reasons are to be processed by a particular widget at application run time. Each entry in a callbacks list has a reason name (in this example, **XmNactivateCallback**) and the name of a callback subroutine (*activate_proc*).

  For AIXwindows Toolkit widgets, the reason names are already built into UIL. For a user–defined widget, you can refer to a user–defined reason name that you previously specified by using the **REASON** subroutine. If you use a built–in reason name in a widget definition, the UIL Compiler ensures that the reason name is supported by the type of widget you are defining.

  If you use the same reason name more than once in a callbacks list, the last entry that uses that reason name supersedes all others, and the UIL Compiler issues a message.

  The callback procedure names you use in a callbacks list must have been previously declared in a procedure section. In the preceding example, the procedure *activate_proc* was declared in the beginning of the UIL Module.

  Because the UIL Compiler produces a UID file rather than an object Module, the binding of the UIL name to the address of the subroutine entry point is not done by the linker. Instead, the binding is established at run time with the MRM **MrmRegisterNames** subroutine. You call this subroutine before fetching any widgets, giving it both the UIL names and the subroutine addresses of each callback. The name you register with MRM in the application program must match the name you specified in the UIL Module.

Suggested Reading

Related Information

How to Create a User Interface at Run Time Using the AIXwindows Resource Manager (MRM) contains detailed information about the AIXwindows Resource Manager.

The **XmBulletinBoard**, **XmCascadeButton**, **XmForm**, **XmLabel**, **XmPushButton**, and **XmToggleButton** widgets.

Using AIXwindows Resource Manager (MRM) Subroutines contains additional information about the MRM subroutines.

Using AIXwindows Resource Manager (MRM) Subroutines

The AIXwindows Resource Manager (MRM) is responsible for creating AIXwindows Toolkit widgets and user–defined widgets based on definitions contained in the User Interface Definition (UID) files created by the User Interface Language (UIL) Compiler. MRM interprets the output of the UIL Compiler and generates the appropriate parameter lists for widgets creation subroutines.

Notes:

1. Literal definitions are created from the exported value definitions in the UIL. The resulting literals can be used for any purpose your application requires.
2. The representation of widgets in a UID file is not exposed in these subroutines. All management and translation of these representations is done internally.
3. All definitions required to use MRM facilities are contained in the following files:
   - **Intrinsic.h**
   - **MrmPublic.h**

Setting Up Storage and Data Structures with the MrmInitialize Subroutine

Use the **MrmInitialize** subroutineto initialize the internal data structures needed by MRM. This subroutine requires the following syntax:

```
void  MrmInitialize()
```

The **MrmInitialize** subroutine must be called to prepare an application to use MRM widget–fetching facilities. You must call this subroutine prior to fetching a widget. However, it is good programming practice to call the **MrmInitialize** subroutine prior to performing any MRM operations.

The **MrmInitialize** subroutine initializes the internal data structures that MRM needs to successfully perform type conversion on parameters and to successfully access widget creation facilities. An application must call the **MrmInitialize** subroutine before it uses other MRM subroutines.

Obtaining UID Database File IDs with the MrmOpenHierarchy Subroutine

An AIXwindows application can access different UID files based on the language preferences of the user. This capability is provided by MRM in a manner consistent with the existing NLS standards as specified in the XOpen Portability Guide. In particular, the capability is compatible with the searching and naming conventions associated with searching and accessing message catalogs.

Use the **MrmOpenHierarchy** subroutine to specify the UID files to open in applications with AIXwindows interfaces. This subroutine requires the following syntax:

```
#include <Mrm/Mrm/Appl.h>
#include <Mrm/MrmPublic.h>
Cardinal MrmOpenHierarchy ( num_files, file_names_list,
    ancillary_structures_list, hierarchy_id )
MrmCount              num_files ;
String                file_names_list[];
MrmOsOpenParamPtr    *ancillary_structures_list;
MrmHierarchy         *hierarchy_id;
```

Parameter	**Description**
num_files	Specifies the number of files in *file_names_list.*
file_names_list	Specifies an array of pointers to character strings that identify the **.uid** files.
ancillary_structures_list	A list of operating system–dependent ancillary structures corresponding to file names, clobber flag, and so forth. This parameter should be **NULL** for most operations. If you need to reference this structure, see the definition of **MrmOsOpenParamPtr** in **MrmPublic.h** for more information.
hierarchy_id	Returns the search hierarchy ID. The search hierarchy ID identifies the list of **.uid** files that MRM searches (in order) when performing subsequent fetch calls.

The **MrmOpenHierarchy** subroutine allows the user to specify the list of UID files that are searched in subsequent fetch operations. All subsequent fetch operations return the first occurrence of the named item encountered while traversing the UID hierarchy from the first list element (UID file specification) to the last list element. This subroutine also allocates a hierarchy ID and opens all the UID files in the hierarchy. It initializes the optimized search lists in the hierarchy. If the **MrmOpenHierarchy** subroutine encounters any errors during its execution, all opened files are closed and a status constant is returned.

Each UID file specified in *file_names_list* can specify either a full directory pathname or a file name. If a UID file does not specify the pathname it will not contain any embedded slashes (/), and will be accessed through the **UIDPATH** environment variable.

The **UIDPATH** environment variable specifies search paths and naming conventions associated with UID files. It can contain the substitution fields **%L** and **%N**, where the current setting of the **LANG** environment variable is substituted for **%L**, and the **.uid** filename which is passed to the **MrmOpenHierarchy** subroutine is substituted for **%N**. For example, the following UID path and **MrmOpenHierarchy** call would cause MRM to open two separate **.uid** files:

```
UIDPATH=/uidlib/%L/%N.uid:/uidlib/%N/%L
static char *uid_files[] = {"/usr/users/me/test.uid", "test2"};
MrmHierarchy *Hierarchy_id;
MrmOpenHierarchy((MrmCount)2, uid_files, NULL, Hierarchy_id)
```

The first file, **/usr/users/me/test.uid**, would be opened as specified because the file specification includes a pathname. The second file, **test2**, would be looked for in **/uidlib/$LANG/test2.uid** first, and then in **/uidlib/test2/$LANG**.

After the **MrmOpenHierarchy** subroutine opens the UID hierarchy, do not delete or modify the UID files until you close the UID hierarchy by calling the **MrmCloseHierarchy** subroutine.

This subroutine returns one of these status return constants:

Status Return Constant	**Meaning**
MrmSUCCESS	The subroutine executed successfully.
MrmNOT_FOUND	File not found.
MrmFAILURE	The subroutine failed.

Closing an MRM Search Hierarchy with the MrmCloseHierarchy Subroutine

Use the **MrmCloseHierarchy** subroutine to close an MRM search hierarchy. This subroutine requires the following syntax:

```
#include <Mrm/MrmAppl.h>
#include <Mrm/MrmPublic.h>
Cardinal MrmCloseHierarchy(hierarchy_id)
MrmHierarchy hierarchy_id;
```

Parameter	Description
hierarchy_id	Specifies the ID of a previously opened UID hierarchy. The *hierarchy_id* parameter is returned from a previous call to the **MrmOpenHierarchy** subroutine.

The **MrmCloseHierarchy** subroutine closes a UID hierarchy previously opened by the **MrmOpenHierarchy** subroutine.

All files associated with the hierarchy are closed by MRM and all associated memory is returned.

This subroutine returns one of the following status return constants:

Status Return Constant	Meaning
MrmSUCCESS	The subroutine executed successfully.
MrmFAILURE	The subroutine failed.

Registering MRM Information and Callbacks

MRM subroutines also provide the following functionality:

- Save information needed to access the widget creation subroutine
- Register a vector of callback subroutines

Registering MRM Information with the MrmRegisterClass Subroutine

Use the **MrmRegisterClass** subroutine to save the information needed to access the widget creation subroutine. The **MrmRegisterClass** subroutine requires the following syntax:

```
#include <Mrm/MrmAppl.h>
#include <Mrm/MrmPublic.h>
Cardinal MrmRegisterClass(class_code, class_name, create_name,
    create_proc, class_record)
MrmType      class_code;
String       class_name;
String       create_name;
Widget       (*create_proc) ();
WidgetClass  class_record;
```

Parameter	Description
class_code	Specifies the code name of the class. For all application–defined widgets, this code name is MRMwcUnknown. For all AIXwindows Toolkit objects, each code name begins with the letters **MRMwc**. The code names for all application widgets are defined in the **Mrm.h** header file.
class_name	Specifies the case–sensitive name of the class. The class names for all AIXwindows widgets are defined in the **Mrm.h** header file. Each class name begins with the letters **MRMwcn**.

create_name	Specifies the case–sensitive name of the low–level widget creation subroutine for the class. An example from the AIXwindows Toolkit is the **XmCreateLabel** subroutine. The following parameters are associated with this subroutine: • *parent_widget* • *name* • *override_arglist* • *override_argcount* For user–defined widgets, the *create_name* parameter is the creation procedure in the UIL that defines this widget.
create_proc	Specifies the address of the creation subroutine named in the *create_name* parameter.
class_record	Specifies a pointer to the class record.

The **MrmRegisterClass** subroutine allows MRM to access user–defined widget classes. This subroutine registers the necessary information for MRM to create widgets of this class. You must call the **MrmRegisterClass** subroutine prior to fetching any user–defined class widget.

The **MrmRegisterClass** subroutine saves the information needed to access the widget creation subroutine and to do type conversion of parameter lists by using the information in MRM databases.

This subroutine returns one of these status return constants:

Status Return Constant	**Meaning**
MrmSUCCESS	The subroutine executed successfully.
MrmFAILURE	The allocation of the class descriptor failed.

Registering a Vector of Callback Subroutines with the MrmRegisterNames subroutine

Use the **MrmRegisterNames** subroutine to register a vector of names of identifiers or callback subroutines for access in MRM. This subroutine requires the following syntax:

```
#include <Mrm/MrmAppl.h>
#include <Mrm/MrmPublic.h>
Cardinal             MrmRegisterNames(register_list,
                                                 register_count)
   MrmRegisterArglist register_list;
   MrmCount           register_count;
```

Parameter	**Description**
register_list	Specifies a list of name/value pairs for the names to be registered. Each name is a case–sensitive, **NULL**–terminated ASCII string. Each value is a 32–bit quantity, interpreted as a procedure address if the name is a callback subroutine, and uninterpreted otherwise.
register_count	Specifies the number of entries in the *register_list* parameter.

The **MrmRegisterNames** subroutine registers a vector of names and associated values for access in MRM. The values can be callback subroutines, pointers to user–defined data, or any other values. The information provided is used to resolve symbolic references occurring in UID files to their run–time values. For callbacks, this information provides the procedure address required by the AIXwindows Toolkit. For names used as identifiers in UIL, this information provides any run–time mapping the application needs.

The names in the list are case–sensitive. The list can be either ordered or unordered.

Callback subroutines registered through the **MrmRegisterNames** subroutine can be either regular or creation callbacks. Regular callbacks have declarations determined by AIXwindows Toolkit and user requirements. Creation callbacks have the same format as any other callback:

```
void CallBackProc(widget_id, tag, callback_data)
Widget              *widget_id;
Opaque             tag;
XmAnyCallbackStruct *callback_data;
```

Parameter	**Description**
widget_id	Specifies the widget ID associated with the widget performing the callback (as in any callback subroutine).
tag	Specifies the tag value (as in any callback subroutine).
callback_data	Specifies a widget–specific data structure. This data structure has a minimum of two members: event and reason. The reason member is always set to XmCRCreate.

Note: The widget name and widget parent are stored in the widget record, which is accessible through the *widget_id* parameter.

This subroutine returns one of these status return constants:

Status Return Constant	**Meaning**
MrmSUCCESS	The subroutine executed successfully.
MrmFAILURE	Memory allocation failed.

Using Creation Callback Subroutines

A creation callback subroutine is called when the UIL object for which it is defined is created. The only way to retrieve the widget ID of each newly–created object as it is created is through use of its creation callback subroutine.

The callback reason is **MrmNcreateCallback**. The callback subroutine has the same syntax as any other callback. It is good coding practice to include this callback for every object in the UIL Specification File.

The following example uses creation callbacks:

```
object
    button : XmPushButton {
        arguments {
                XmNx = 30
                XmNy = 40
                XmNlabelString = compound_string('AIX');
        };
        callbacks {
                XmNactivateCallback = procedure button_pushed();
                MrmNcreateCallback = procedure object_created();
        };
    };
```

Fetching Widgets

This section discusses the MRM subroutines you can use to:

- Fetch all the widgets defined in some interface
- Fetch values stored in UID files
- Fetch any indexed application widget
- Override the parameters of the **MrmFetchWidget** subroutine

Fetching All Widgets Defined in Some Interface with the MrmFetchInterfaceModule Subroutine

Use the **MrmFetchInterfaceModule** subroutine to fetch all the widgets defined in some interface. This subroutine requires the following syntax:

```
#include <Mrm/MrmAppl.h>
#include <Mrm/MrmPublic.h>
Cardinal MrmFetchInterfaceModule(hierarchy_id, module_name,
    parent_widget, widget)
MrmHierarchy     hierarchy_id;
char             *module_name;
Widget           parent_widget;
Widget           *widget;
```

Parameter	**Description**
hierarchy_id	Specifies the ID of the UID hierarchy that contains the interface definition. The *hierarchy_id* parameter is returned from a previous call to the **MrmOpenHierarchy** subroutine.
module_name	Specifies the name of the interface Module, which you specified in the UIL Module header. By convention, this is usually the generic name of the application.
parent_widget	Specifies the parent widget ID for the topmost widgets being fetched from the Module. The topmost widgets are those that have no parents specified in the UIL Module. The parent widget is usually the top–level shell widget returned by the Enhanced X–Windows Intrinsics **XtInitialize** subroutine (in **libXt.a**).
widget	Returns the widget ID for the last main window widget encountered in the UIL Module, or **NULL** if no main window widget is found.

The **MrmFetchInterfaceModule** subroutine fetches all the widgets defined in a UIL Module in the UID hierarchy. Typically, each application has one or more Modules that define its interface. Each must be fetched in order to initialize all the widgets the application requires. Applications do not need to define all their widgets in a single Module.

If the Module defines a main window widget, the **MrmFetchInterfaceModule** subroutine returns its widget ID. If no main window widget is contained in the Module, the **MrmFetchInterfaceModule** subroutine returns **NULL** and no widgets are realized.

The application can obtain the IDs of widgets other than the main window widget by using creation callbacks as described in the section titled Using Creation Callbacks.

This subroutine returns one of these status return constants:

Status Return Constant	Meaning
MrmSUCCESS	The subroutine executed successfully.
MrmFAILURE	The subroutine failed.
MrmNOT_FOUND	The interface Module or topmost widget not found.

Fetching Values Stored in UID Files

use the **MrmFetchSetValues** subroutine to fetch the values to be set from literals stored in UID files. This subroutine requires the following syntax:

```
#include <Mrm/MrmAppl.h>
#include <Mrm/MrmPublic.h>
Cardinal MrmFetchSetValues(hierarchy_id, widget, args, num_args)
MrmHierarchy hierarchy_id;
Widget       widget;
ArgList      args;
Cardinal     num_args;
```

Parameter	Description
hierarchy_id	Specifies the ID of the UID hierarchy that contains the specified literal. The *hierarchy_id* parameter is returned from a previous call to the **MrmOpenHierarchy** subroutine.
widget	Specifies the widget that is modified.
args	Specifies a parameter list that identifies the widget parameters to be modified as well as the index (UIL name) of the literal that defines the value for that parameter. The name part of each parameter `(args[n].name)` must begin with the string **XmN** followed by the name that uniquely identifies this resource tag. For example, **XmNwidth** is the resource name associated with the core parameter `width`. The value part `(args[n].value)` must be a string that gives the index (UIL name) of the literal. You must define all literals in UIL as exported values.
num_args	Specifies the number of entries in the *args* parameter.

The **MrmFetchSetValues** subroutine is similar to the Enhanced X–Windows Intrinsics **XtSetValues** subroutine (in **libXt.a**), except that the values to be set are defined by the UIL named values that are stored in the UID hierarchy. The **MrmFetchSetValues** subroutine fetches the values to be set from literals stored in UID files.

This subroutine sets the values on a widget, evaluating the values as public literal resource references resolvable from a UID hierarchy. Each literal is fetched from the hierarchy, and its value is modified and converted as required. This value is then placed in the parameter list and used as the actual value for a call to the Enhanced X–Windows Intrinsics **XtSetValues** subroutine. The **MrmFetchSetValues** subroutine allows a widget to be modified after creation using UID file values exactly as is done for creation values in the **MrmFetchWidget** subroutine.

As in the **MrmFetchWidget** subroutine, each parameter whose value can be evaluated from the UID hierarchy is set in the widget. Values that are not found or values in which conversion errors occur are not modified.

Each entry in the parameter list identifies a parameter to be modified in the widget. The name part identifies the tag, which begins with **XmN**. The value part must be a string whose value is the index of the literal. Thus, the following code would modify the **XmLabel** resource of the widget to have the value of the literal accessed by the index **OK_button_label** in the hierarchy:

```
args[n].name = XmNlabel;
args[n].value = "OK_button_label";
```

This subroutine returns one of these status return constants:

Status Return Constant	Meaning
MrmSUCCESS	The subroutine executed successfully.
MrmFAILURE	The subroutine failed.

Fetching Indexed Application Widget

Use the **MrmFetchWidget** subroutine to fetch any indexed application widget. This subroutine requires the following syntax:

```
#include <Mrm/MrmAppl.h>
#include <Mrm/MrmPublic.h>
Cardinal MrmFetchWidget(hierarchy_id, index, parent_widget,
    widget, class)
MrmHierarchy hierarchy_id;
String       index;
Widget       parent_widget;
Widget       *widget;
MrmType      *class;
```

Parameter	Description
hierarchy_id	Specifies the ID of the UID hierarchy that contains the interface definition. The *hierarchy_id* parameter is returned from a previous call to the **MrmOpenHierarchy** subroutine.
index	Specifies the UIL name of the widget to fetch.
parent_widget	Specifies the parent widget ID.
widget	Returns the widget ID of the created widget. If this is not **NULL** when you call the **MrmFetchWidgetOverride** subroutine, MRM assumes that the widget has already been created and the **MrmFetchWidgetOverride** subroutine returns MrmFAILURE.
class	Returns the class code identifying the MRM widget class. The widget class code for the main window widget, for example, is **MRMwcMainWindow.** Literals identifying MRM widget class codes are defined in the **Mrm.h** header file.

The **MrmFetchWidget** subroutine fetches and then creates an indexed application widget and its children. The indexed application widget is any widget that is named in UIL and that is not the child of any other widget in the UID hierarchy. In fetch operations, the fetched subtree of the widget is also fetched and created. This widget must not appear as the child of a widget within its own subtree. The **MrmFetchWidget** subroutine does not execute the Enhanced X-Windows **XtManageChild** subroutine for the newly created widget.

The **MrmFetchWidget** subroutine fetches widgets where the **MrmFetchInterfaceModule** subroutine is not used. The **MrmFetchWidget** subroutine provides specific control over which widgets are fetched from a UIL file; the **MrmFetchInterfaceModule** subroutine, on the other hand, fetches all widgets in a single call. An application can fetch any named widget in the UID hierarchy using the **MrmFetchWidget** subroutine. The **MrmFetchWidget** subroutine can be called at any time to fetch a widget not fetched at application startup. The **MrmFetchWidget** subroutine determines if a widget has already been fetched by checking the `widget` parameter for a **NULL** value. Non–**NULL** values signify that the widget already has been fetched, and the **MrmFetchWidget** subroutine fails. The **MrmFetchWidget** subroutine can be used to defer fetching pop–up widgets until they are first referenced (presumably in a callback), and then used to fetch them once.

The **MrmFetchWidget** subroutine can also create multiple instances of a widget (and its subtree). In this case, the UID definition serves as a template; a widget definition can be fetched any number of times. An application can use this to make multiple instances of a widget, for example, in a dialog box box or menu.

The index (UIL name) that identifies the widget must be known to the application.

This subroutine returns one of these status return constants:

Status Return Constant	**Meaning**
MrmSUCCESS	The subroutine executed successfully.
MrmNOT_FOUND	Widget not found in UID hierarchy.
MrmFAILURE	The subroutine failed.

Overriding the Parameters of the MrmFetchWidget Subroutine

Use the **MrmFetchWidgetOverride** subroutine to fetch any indexed application widget and override the parameters of the **MrmFetchWidget** subroutine. This subroutine requires the following syntax:

```
#include <Mrm/MrmAppl.h>
#include <Mrm/MrmPublic.h>
Cardinal MrmFetchWidgetOverride(hierarchy_id, index,
    parent_widget, override_name, override_args,
    override_num_args, widget, class)
MrmHierarchy hierarchy_id;
String       index;
Widget       parent_widget;
String       override_name;
ArgList      override_args;
Cardinal     override_num_args;
Widget       *widget;
MrmType      *class;
```

Parameter	**Description**
hierarchy_id	Specifies the ID of the UID hierarchy that contains the interface definition. The *hierarchy_id* parameter is returned from a previous call to the **MrmOpenHierarchy** subroutine.
index	Specifies the UIL name of the widget to fetch.
parent_widget	Specifies the parent widget ID.
override_name	Specifies the name to override the widget name. Use a **NULL** value if you do not want to override the widget name.
override_args	Specifies the override parameter list, exactly as would be given to the Enhanced X–Windows Intrinsics **XtCreateWidget** subroutine (conversion complete and so forth). Use a **NULL** value if you do not want to override the parameter list.
override_num_args	Specifies the number of parameters in the *override_args* parameter.
widget	Returns the widget ID of the created widget. If this is not **NULL** when you call the **MrmFetchWidgetOverride** subroutine, MRM assumes that the widget has already been created and the **MrmFetchWidgetOverride** subroutine returns MrmFAILURE.

class	Returns the class code identifying the MRM widget class. The widget class code for the main window widget, for example, is MRMwcMainWindow. Literals identifying MRM widget class codes are defined in **Mrm.h** header file.

The **MrmFetchWidgetOverride** subroutine is the extended version of the **MrmFetchWidget** subroutine. It is identical to the **MrmFetchWidget** subroutine, except that it allows the caller to override the name of the widget and any parameters that the **MrmFetchWidget** subroutine would otherwise retrieve from the UID file or one of the defaulting mechanisms. That is, the override parameter list is not limited to those parameters in the UID file.

The override parameters apply only to the widget fetched and returned by this subroutine. Its children (subtree) do not receive any override parameters.

This subroutine returns one of these status return constants:

Status Return Constant	Meaning
MrmSUCCESS	The subroutine executed successfully.
MrmNOT_FOUND	Widget not found in UID hierarchy.
MrmFAILURE	The subroutine failed.

Fetching Literals

The AIXwindows Toolkit provides subroutines with which you can fetch literals from UID files. Specifically, the section discusses how to fetch the following literals:

- A named color literal
- An icon literal
- A literal value

Fetching a Named Color Literal

Use the **MrmFetchColorLiteral** subroutine to fetch a named color literal. This subroutine requires the following syntax:

```
#include <Mrm/MrmAppl.h>
#include <Mrm/MrmPublic.h>
int MrmFetchColorLiteral(hierarchy_id, index, display,
                                          colormap_id, pixel)
MrmHierarchy  hierarchy_id;
String        index;
Display       *display;
Colormap      colormap_id;
Pixel         *pixel;
```

Parameter	Description
hierarchy_id	Specifies the ID of the UID hierarchy that contains the specified literal. The *hierarchy_id* parameter is returned from a previous call to the **MrmOpenHierarchy** subroutine.
index	Specifies the UIL name of the color literal to fetch. You must define this name in UIL as an exported value.
display	Specifies the display used for the pixmap. The *display* parameter specifies the connection to the X server. For more information on the **Display** structure see the Enhanced X–Windows Intrinsics **XOpenDisplay** subroutine (in **libXt.a**).
colormap_id	Specifies the ID of the color map. If **NULL**, the default color map is used.

pixel	Returns the ID of the color literal.

The **MrmFetchColorLiteral** subroutine fetches a named color literal from a UID file, and converts the color literal to a pixel color value.

This subroutine returns one of these status return constants:

Status Return Constant	Meaning
MrmSUCCESS	The subroutine executed successfully.
MrmNOT_FOUND	The color literal is not found in the UIL file.
MrmFAILURE	The subroutine failed.

Fetch an Icon Literal with the MrmFetchIconLiteral Subroutine

Use the **MrmFetchIconLiteral** subroutine to fetch an icon literal. This subroutine requires the following syntax:

```
#include <Mrm/MrmAppl.h>
#include <Mrm/MrmPublic.h>
int MrmFetchIconLiteral(hierarchy_id, index, screen, display,
    fgpix, bgpix, pixmap)
MrmHierarchy hierarchy_id;
String       index;
Screen       *screen;
Display      *display;
Pixel        fgpix;
Pixel        bgpix;
Pixmap       *pixmap;
```

Parameter	Description
hierarchy_id	Specifies the ID of the UID hierarchy that contains the specified icon literal. The *hierarchy_id* parameter is returned from a previous call to the **MrmOpenHierarchy** subroutine.
index	Specifies the UIL name of the icon literal to fetch.
screen	Specifies the screen used for the pixmap. The *screen* parameter specifies a pointer to the Enhanced X–Windows **Screen** structure, which contains the information about that screen and is linked to the **Display** structure. For more information on the **Display** and **Screen** structures see the Enhanced X–Windows Intrinsics **XOpenDisplay** subroutine (in **libXt.a**) and the associated screen information macros.
display	Specifies the display used for the pixmap. The *display* parameter specifies the connection to the X server. For more information on the **Display** structure see the Enhanced X–Windows Intrinsics **XOpenDisplay** subroutine.
fgpix	Specifies the foreground color for the pixmap.
bgpix	Specifies the background color for the pixmap.
pixmap	Returns the resulting Xpixmap value.

The **MrmFetchIconLiteral** subroutine fetches an icon literal from an MRM hierarchy, and converts the icon literal to an Xpixmap.

This subroutine returns one of these status return constants:

Status Return Constant	Meaning
MrmSUCCESS	The subroutine executed successfully.
MrmNOT_FOUND	The icon literal is not found in the hierarchy.
MrmFAILURE	The subroutine failed.

Fetch a Literal Value with the MrmFetchLiteral Subroutine

Use the **MrmFetchLiteral** subroutine to fetch a literal value. This subroutine requires the following syntax:

```
#include <Mrm/MrmAppl.h>
#include <Mrm/MrmPublic.h>
int MrmFetchLiteral(hierarchy_id, index, display, value, type)
MrmHierarchy hierarchy_id;
String        index;
Display       *display;
caddr_t       *value;
MrmCode       *type;
```

Parameter	Description
hierarchy_id	Specifies the ID of the UID hierarchy that contains the specified literal. The *hierarchy_id* parameter is returned from a previous call to the **MrmOpenHierarchy** subroutine.
index	Specifies the UIL name of the literal (pixmap) to fetch. You must define this name in UIL as an exported value.
display	Specifies the display used for the pixmap. The *display* parameter specifies the connection to the X server. For more information on the **Display** structure see the Enhanced X–Windows Intrinsics **XOpenDisplay** subroutine (in **libXt.a**).
value	Returns the ID of the value of the named literal.
type	Returns the data type of the named literal.

The **MrmFetchLiteral** subroutine reads and returns the value and type of a literal (named value) that is stored as a public resource in a single UID file. This subroutine returns a pointer to the value of the literal. For example, an integer is always returned as a pointer to an integer, and a string is always returned as a pointer to a string.

Applications should not use the **MrmFetchLiteral** subroutine for fetching icon or color literals. If this is attempted, the **MrmFetchLiteral** subroutine returns an error.

This subroutine returns one of these status return constants:

Status Return Constant	Meaning
MrmSUCCESS	The subroutine executed successfully.
MrmWRONG_TYPE	The operation encountered an unsupported literal type.
MrmNOT_FOUND	The literal is not found in the UID file.
MrmFAILURE	The subroutine failed.

How to Create a User Interface at Run Time Using the AIXwindows Resource Manager (MRM)

The AIXwindows Resource Manager (MRM) creates interface widgets based on definitions in UID files (the compiled form of UIL Specification Files in a UIL Module). The MRM does not replace the Enhanced X–Windows Resource Manager, but complements it.

The Enhanced X–Windows Resource Database (an in–memory database stored in the **.Xdefaults** file) supplies resource default values. When you use UIL to specify a user interface, you only need to specify parameter values when you want to override the default values of resources stored in this Resource Database. The MRM then generates the *override_arglist* parameter for the appropriate widget creation subroutines at run time.

All definitions required to use MRM are contained in the **MrmAppl.h** header file.

You call MRM subroutines to accomplish the following tasks:

- Initialize MRM
- Provide information required by MRM to interpret information in UID files
- Create objects using UID file definitions
- Defer the fetching of top–level objects to reduce application startup time.
- Allow your application to read literal definitions from UID files.
- Override values specified for widget resources.

The examples in this section are based on the C program for the **AIXburger** application. The **AIXburger** application is part of the AIXwindows software kit. The **AIXburger** application demonstrates the most commonly–used MRM subroutines including the following subroutines:

Subroutine Name	**Description**
MrmCloseHierarchy	Closes a UID hierarchy.
MrmFetchColorLiteral	Fetches a named color literal from a UID hierarchy.
MrmFetchIconLiteral	Fetches a named icon from a UID hierarchy.
MrmFetchInterfaceModule	Fetches all the objects defined in some interface Module in the UID hierarchy.
MrmFetchLiteral	Fetches a named string literal from a UID hierarchy.
MrmFetchSetValues	Fetches the values to be set from literals stored in a UID hierarchy.
MrmFetchWidget	Fetches any named widget in a UID hierarchy.
MrmFetchWidgetOverride	Fetches any named widget and overrides values stored in the UID hierarchy with those supplied in the subroutine call. In effect, a single object definition can be used like a template to create multiple widget instances from a single UIL definition.
MrmInitialize	Prepares an application to use MRM widget fetching facilities.
MrmOpenHierarchy	Allocates a hierarchy descriptor and opens all the files in the UID hierarchy.

MrmRegisterClass	Saves the information needed to access the widget creation subroutine using the information in a UID hierarchy and to perform type conversion of parameters lists.
MrmRegisterNames	Registers a vector of names and associated values for access by MRM.

Prerequisite Tasks or Conditions

1. Ensure that the UIL Compiler and the include files and other header files listed in your source code are available and accessible.
2. Create one or more UIL Specification Files containing one or more UIL Modules.
3. Compile the UIL Module to generate a UID Module.

Procedure

Accessing the UID File at Run Time

To set up an AIXwindows interface specified with UIL, complete the following steps in the order shown:

1. Verify that your application is initializing the following software by making calls to the MRM and the AIXwindows Toolkit subroutines in the order listed:

- Initialize MRM

The MRM **MrmInitialize** subroutine prepares your application to use MRM widget–fetching facilities. This call must come before the call to initialize the AIXwindows Toolkit.

- Initialize the AIXwindows Toolkit

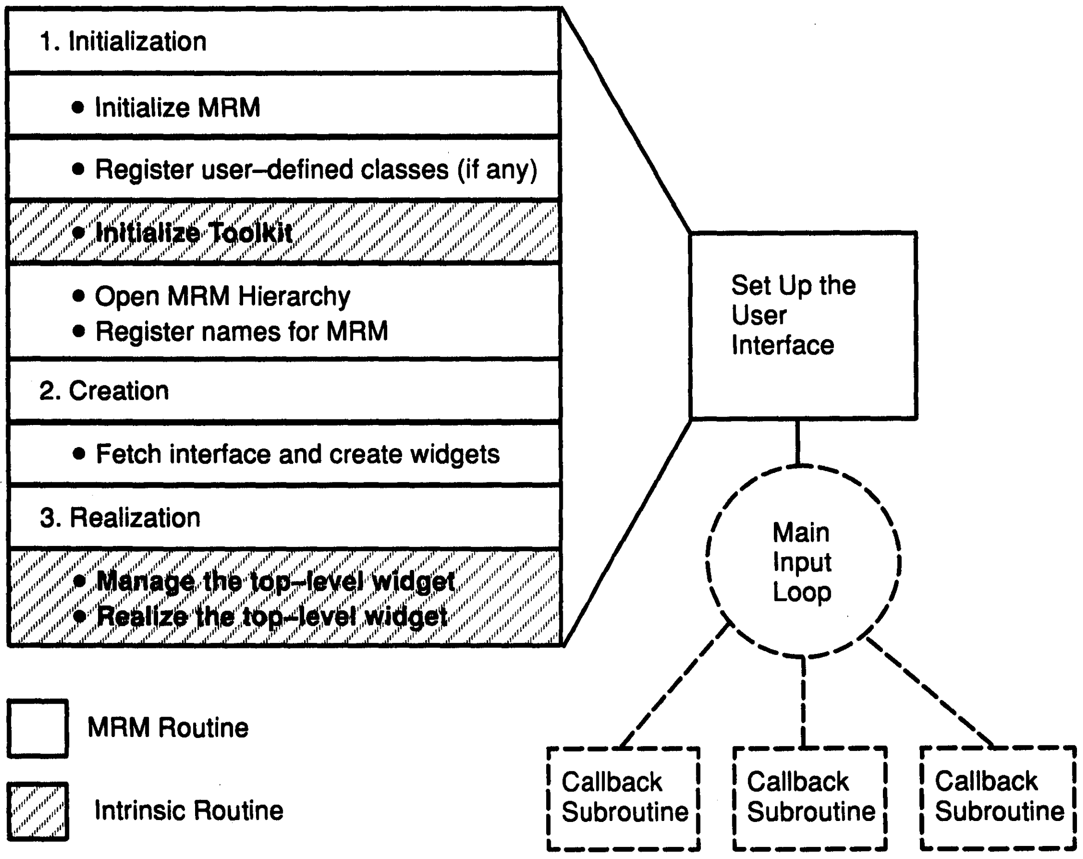
The Enhanced X–Windows Intrinsics **XtInitialize** subroutine (in **libXt.a**) parses the command line used to invoke the application, opens the display, and initializes the AIXwindows Toolkit.

- Open the UID hierarchy

The UID hierarchy is the set of UID files containing the widget definitions for the user interface. The MRM **MrmOpenHierarchy** subroutine opens these UID files.

- Register names for MRM

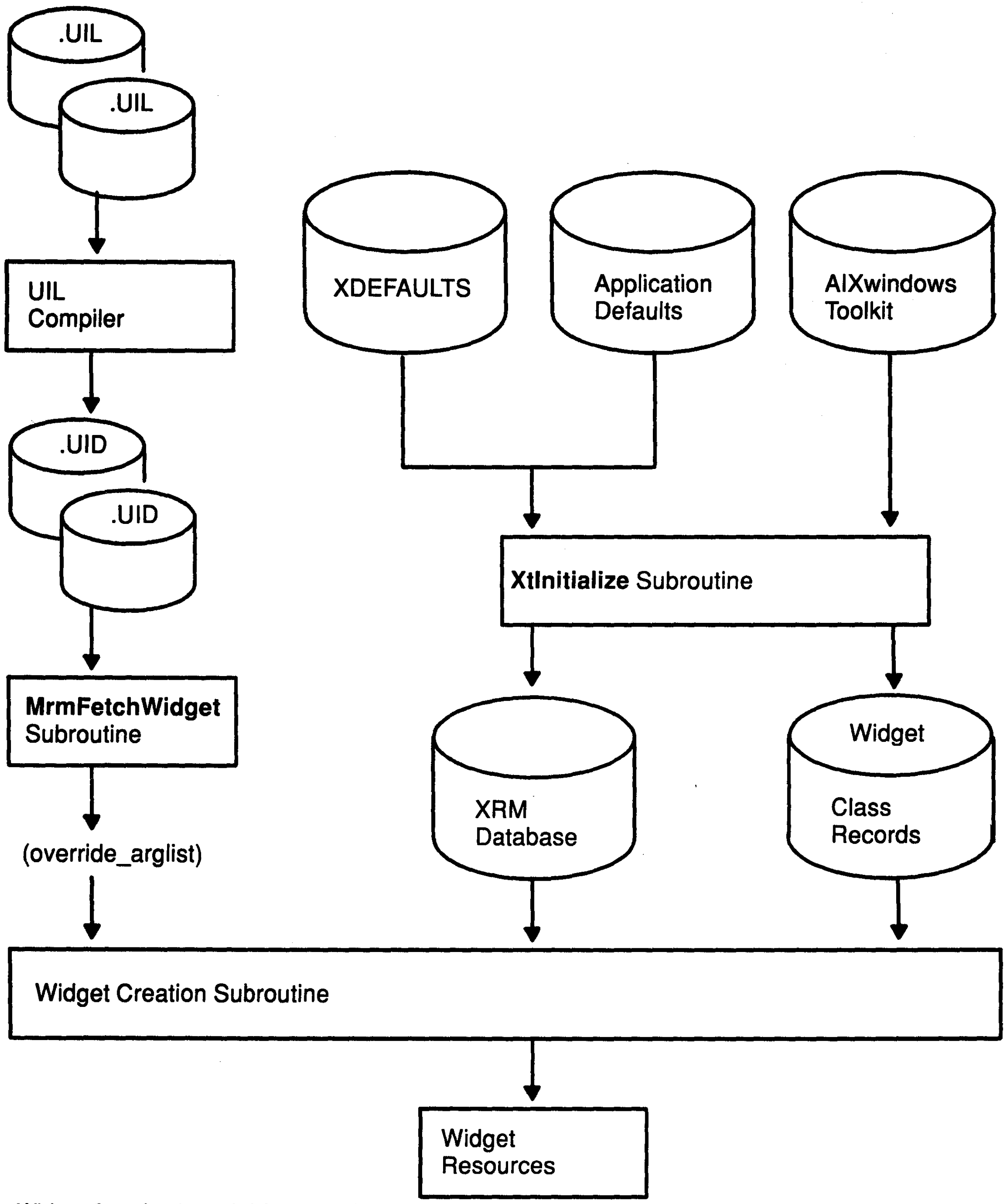
The MRM **MrmRegisterNames** subroutine registers names and associated values for access by the MRM. The values may be callback subroutines, pointers to user–defined data, or any other values. MRM uses this information to resolve symbolic references in UID files to their run–time values.



Setting Up a User Interface Specified with UIL

2. Add the MRM **MrmFetchWidget** subroutine to your code to fetch the user interface.

Fetching is a combination of widget creation and child management.



Widget Creation in an MRM Fetch Operation

The MRM **MrmFetchWidget** subroutine performs the following tasks associated with fetching:

- Locates a widget description in the UID hierarchy
- Creates the widget and recursively creates the children of the widget
- Manages all widget children as specified in the UID hierarchy
- Returns the widget identifier

You specify the top–level widget of the application (usually the main window) and its parent (the widget identifier returned by the call to the Enhanced X–Windows Intrinsics **XtInitialize** subroutine (in **libXt.a**) in the call to **MrmFetchWidget**. As a result of this single call, MRM fetches all widgets in the widget tree below the top–level widget.

It is possible to defer fetching portions of an application interface until they are requested by the end user. For example, you can defer fetching a pull–down menu until the user activates the corresponding cascade button. Consider deferring fetching of some portions of your interface if you need to improve the startup performance of your application.

3. Manage and realize your user interface.

You can ensure the management and realization of your user interface by completing the following tasks in the order shown:

Note: The tasks involved in these processes are the same for interfaces created using UIL and MRM as they are for interfaces created with AIXwindows Toolkit subroutines.

a. Manage the top–level widget.

The Enhanced X–Windows Intrinsics **XtManageChild** subroutine (in **libXt.a**) adds a child to the top–level widget returned by the call to the **XtInitialize** subroutine (in **linXt.a**). The entire widget tree below the top–level widget in the interface (usually the main window) is automatically managed as a result of this call to the **XtManageChild** subroutine.

b. Verify that the call to the **MrmInitialize** subroutine precedes the call to the **XtInitialize** subroutine.

The following example shows the initialization of MRM and the AIXwindows Toolkit in the **AIXburger** application:

```
unsigned int main(argc, argv)
 unsigned int argc;
 char *argv[];
{
 MrmInitialize();
 toplevel_widget = XtInitialize("Welcome to AIXburger",
        "example",
        NULL,
        0,
        &argc,
        argv);
}
```

c. Verify that the compiled interface described in the UIL Module can connect to the application when the UID hierarchy is set up at run time.

The names of the UID files containing the compiled interface definitions are stored in an array. Because compiled UIL files are not object files, this run–time connection is necessary to bind an interface with an application program. The **AIXburger** application has a single UIL Module, so the MRM hierarchy consists of one file. The following example shows the declaration of the UID hierarchy for **AIXburger**.

```
static MrmHierarchy     s_MrmHierarchy;
static MrmType          *dummy_class;
static char             *db_filename_vec[] =
     {"aixburger.uid"
     };
```

d. The name of the UID hierarchy is *s_MrmHierarchy*. The array containing the names of the UID files in the UID hierarchy is *db_filename_vec*. In the following example, the application opens this UID hierarchy:

Note: At this point in the execution of the application, MRM has access to the **AIXburger** interface definition and can fetch widgets. There is a standard order of precedence in which MRM searches for parameter values. Once MRM finds a parameter definition, it stops searching.

```
if (MrmOpenHierarchy(db_filename_num,
    db_filename_vec,
    NULL,
    &s_MrmHierarchy)
    !=MrmSUCCESS)
    s_error("can't open hierarchy");
```

e. Register a vector of names and associated values.

These values can be the names of callback subroutines, pointers to user–defined data, or any other values. MRM uses the information provided in this vector to resolve symbolic references that occur in UID files to their run–time values. For callback procedures, the vector provides procedure addresses required by the AIXwindows Toolkit. For names used as variables in UIL (identifiers), this information provides whatever mapping the application requires.

The following example shows the declaration of the names vector in the **AIXburger** C program. In the **AIXburger** application, the names vector contains only names of callback procedures and their addresses.

```
static MRMRegisterArg reglist[] = {
    {"activate_proc",    (caddr_t) activate_proc},
    {"create_proc",      (caddr_t) create_proc},
    {"list_proc",        (caddr_t) list_proc},
    {"pull_proc",        (caddr_t) pull_proc},
    {"quit_proc",        (caddr_t) quit_proc},
    {"scale_proc",       (caddr_t) scale_proc},
    {"show_hide_proc",   (caddr_t) show_hide_proc},
    {"show_label_proc",  (caddr_t) show_label_proc},
    {"toggle_proc",      (caddr_t) toggle_proc}
};
static int reglist_num = (sizeof reglist / sizeof reglist [0]);
```

The names are registered in a call to the **MrmRegisterNames** subroutine, as shown in the following example:

```
MrmRegisterNames(reglist, reglist_num);
```

f. Realize the top–level widget

The Enhanced X–Windows Intrinsics **XtRealizeWidget** subroutine (in **libXt.a**) displays the entire interface (the widget tree below the top–level widget) on the screen.

The role of MRM in an AIXwindows application is limited primarily to widget creation. MRM makes run–time calls that create widgets from essentially invariant information (information that does not change from one invocation of the application to the next). After MRM fetches a widget (creates it and manages its children), it provides no further services. All subsequent operations on the widget, such as realization, managing and unmanaging children after initialization, and getting and setting resource values, must be done by run–time calls. After widget creation, you modify widgets during application execution by using widget manipulation subroutines.

How to Defer Fetching

MRM allows you to defer fetching off–screen widgets until the application needs to display these widgets. There are two types of off–screen widgets: pull–down menus and dialogs. Whenever MRM fetches an off–screen widget, it also fetches the entire widget tree below that widget. By deferring the fetching of off–screen widgets, you can reduce the time it takes to start up your application.

The **AIXburger** application makes use of deferred fetching. The pull–down menus for the File, Edit, and Order options are not fetched when the main window is fetched. Instead, these menus are fetched and created by individual calls to the **MrmFetchWidget** subroutine when the eorresponding cascade button is activated (selected by the end user). You can use the **MrmFetchWidget** subroutine at any time to fetch a widget that was not fetched at application start–up.

The UIL Module for the **AIXburger** application is set up to allow either deferred fetching or a single fetch to create the entire widget tree. To fetch the entire interface at once, remove the comment character (!) from the controls list for the *file_menu_entry*, *edit_menu_entry*, and *order_menu_entry* widgets. As long as the comment characters remain on the controls list for the pull–down menu entries, their associated pull–down menus are no longer children; they are top–level widgets and can be fetched individually. The following example shows the object declaration for the **XmCascadeButton** object named *file_menu_entry*:

```
object
    file_menu_entry : XmCascadeButton {
        parameters {
            XmNlabelString = k_file_label_text;
        };
        controls {
            XmPulldownMenu file_menu;
        };
        callbacks {
            XmNcascadingCallback = procedure pull_proc
                                   (k_file_pdme);
            MrmcreateCallback = procedure create_proc
                                   (k_file_pdme);
        };
    };
```

When you remove the comment characters, the controls list on each **XmCascadeButton** specifies the pull–down menu as a child. The pull–down menus are no longer top–level widgets; instead, they are loaded when the **XmCascadeButton** is created.

How to Get Literal Values from UID Files

Using the literal fetching subroutines (**MrmFetchColorLiteral**, **MrmFetchIconLiteral**, and **MrmFetchLiteral**), you can retrieve any named, exported UIL value from a UID file at run time. This is useful when you want to use a value in a context other than fetching an object. These subroutines allows you to treat the UID file as a repository for all the programming variables you need to specify your application interface.

The MRM literal fetching subroutines have a wide variety of uses. For example, you can store the following as named, exported literals in a UIL Module for run–time retrieval:

- All the error messages to be displayed in a message box (stored in a string table)
- All string tables used to query the operating system
- Language–dependent strings

For example, in the C program for the **AIXburger** application, the text string displayed in the title bar of the main window is supplied directly to the Enhanced X–Windows Intrinsics **XtInitialize** subroutine (in **libXt.a**), as shown in the following example:

```
toplevel_widget = XtInitialize("Welcome to AIXburger",
    "example",
    NULL,
    0,
    &argc,
    argv);
```

Alternatively, this string could be specified in a UIL Module as a named, exported compound string, and retrieved from the UID file at run time with the **MrmFetchLiteral** subroutine. Since this string appears in the interface, you should declare it as a compound string. Compound strings can be displayed in a variety of character sets, as required by the language of the interface.

How to Set Values at Run Time Using UID Resources

The MRM **MrmFetchSetValues** subroutine allows you to modify at run time an object that has already been created. The **MrmFetchSetValues** subroutine works like the Enhanced X–Windows Intrinsics **XtSetValues** subroutine (in **libXt.a**) except that MRM fetches the values to be set from named, exported resources (literals) in the UID file. The fetched values are converted to the correct data type, if necessary, and placed in the *args* parameter for a call to the **XtSetValues** subroutine. Since the **MrmFetchSetValues** subroutine looks for the literal values in a UID file, the parameter names you provide to the **MrmFetchSetValues** subroutine must be UIL parameter names (not AIXwindows Toolkit resources names).

You can think of the **MrmFetchSetValues** subroutine as a convenience subroutine that packages the subroutines provided by the **MrmFetchLiteral** subroutine and the **XtSetValues** subroutine.

The value member of the name and value pairs passed to **MrmFetchSetValues** is the UIL name of the value, not an explicit value. When the application calls **MrmFetchSetValues,** MRM looks up the names in the UID file, then uses the values corresponding to those names to override the original values in the object declaration. The **MrmFetchSetValues** subroutine, therefore, allows you to keep all values used in an application in the UIL Module and not in the application program. (The values you pass to the **MrmFetchSetValues** subroutine must be named, exported literals in the UIL Module.)

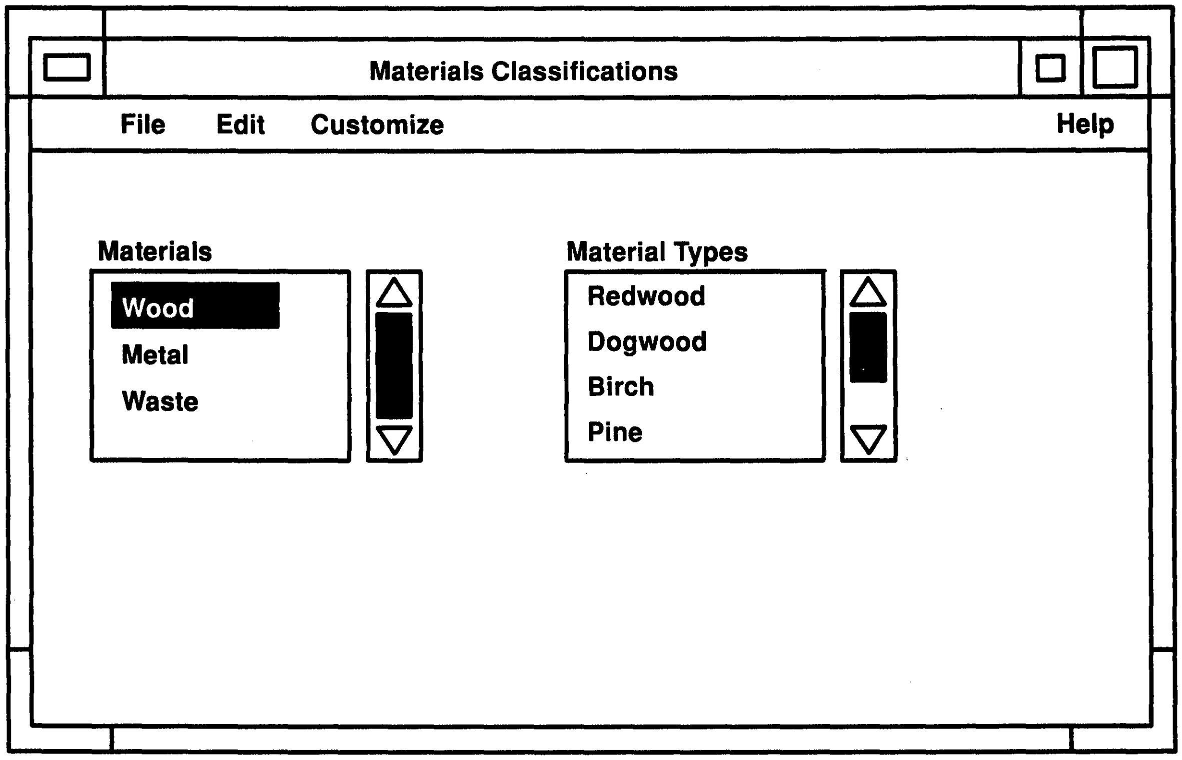
The **MrmFetchSetValues** subroutine offers the following advantages:

- It performs all the necessary UIL resource manipulation to make the fetched UIL values usable by the AIXwindows Toolkit. (For example, the **MrmFetchSetValues** subroutine performs address recomputation for tables of strings and enables a UIL icon to act as a pixmap.)
- It lets you isolate a greater amount of interface information from the application program (to achieve further separation of form and function).

There are some limitations to the **MrmFetchSetValues** subroutine, including the following disadvantages:

- All values in the *args* parameter must be names of exported resources listed in a UIL Module (UID hierarchy); therefore, the application cannot provide computed values from within the program itself as part of the parameter list.
- The subroutine uses the Enhanced X–Windows Intrinsics **XtSetValues** subroutine (in **libXt.a**), ignoring the possibility of the less costly high–level subroutine that the widget itself may provide.

The following examples are based on a simple application with an interactive interface that displays text in two list widgets. The text displayed in the second list widget depends on the selection that the user makes in the first.



Sample Application Using the MRM **MrmFetchSetValues** Subroutine

This application is well–suited to using the MRM subroutine **MrmFetchSetValues** for the following reasons:

- The data (list widget contents) are all known in advance; the values themselves do not need to be computed at run time.
- The data consist of tables of compound strings that appear in the user interface and, therefore, must be translated for international markets. (Strings that must be translated should be stored in a UID file.)
- The structure of compound string tables, if retrieved from the UID file using the **MrmFetchLiteral** subroutine, must be modified by the application program due to the nature of MRM storage methods. (Since string tables are stored in contiguous blocks, indexes to string table values are offsets, not true addresses, and must be added together to compute the actual address at run time.) The **MrmFetchSetValues** subroutine performs this address computation automatically and deallocates memory used to store the fetched tables. Since the program will not be using the fetched string table directly, but intends only to modify the visual appearance of a widget based on items in the table, the **MrmFetchLiteral** subroutine is less convenient to use.

The following examples show the UIL Module for this application and its interactive interface as well as excerpts from the C program. The segment of the UIL Module shown assumes that the module header, procedure declarations, include files, and value declarations for each of the names used in the example are in place.

```
 value
(1)
   cs_wood : compound_string("Wood");
   cst_materials_selected : string_table(cs_wood);
(2)
   cst_materials : exported string_table(
       cs_wood,     ! material type 1
       "Metal",     ! material type 2
       "Waste");    ! material type 3
(3)
   cst_type_1 : exported string_table(!Materials for type 1(wood)
         "Redwood","Dogwood","Birch","Pine","Cherry");
   l_count_type_1 : exported 5;
   cst_type_2 : exported string_table(!Materials for type 2 (metal)
         "Aluminum","Steel","Titanium","Iron","Linoleum");
   l_count_type_2 : exported 5;
   cst_type_3 : exported string_table(!Materials for type 3 (waste)
         "Toxic","Solid","Biodegradable","Party Platforms");
   l_count_type_3 : exported 4;
   k_zero : exported 0;

   object
       materials_ListBox : XmList
       {
           arguments
           {
               XmNx = k_tst_materials_ListBox_x;
               XmNy = k_tst_materials_ListBox_y;
               XmNwidth = k_tst_materials_ListBox_wid;
               XmNvisibleItemCount = 4;
               XmNitems = cst_materials;
               XmNselectedItems = cst_materials_selected;
           };
           callbacks
           {
               MrmNcreateCallback =
                 procedure tst_create_proc(k_tst_materials_ListBox);
               XmNsingleSelectionCallback =
                 procedure tst_single_proc(k_tst_materials_ListBox);
           };
   };
   types_ListBox : XmList
   {
       arguments
       {
           XmNx = k_tst_types_ListBox_x;
           XmNy = k_tst_types_ListBox_y;
           XmNwidth = k_tst_types_ListBox_wid;
           XmNvisibleItemCount = 4;
           XmNitems = cst_type_1;
       };
       callbacks {
           MrmNcreateCallback =
             procedure tst_create_proc(k_tst_types_ListBox);
           XmNsingleSelectionCallback =
             procedure tst_single_proc(k_tst_types_ListBox);
       };
   };
```

The following comments apply to the preceding source code example:

Code Segment #1:	Prefixes on value names indicate the type of value. For example, *cs_* means compound string, *cst_* means compound string table, and *l_* means longword integer.
Code Segment #2:	This string table provides the contents for the Materials list box widget. This string table does not need to follow the naming scheme for the string table in the Material Types list widget (that is, *cst_type_n*) because the contents of the Materials list does not change once the application is realized. The numbering of the string tables is vital to the proper functioning of the Material Types list widget. The string table for the Materials list box widget could have been named anything.
Code Segment #3:	These string tables provide the contents for the various versions of the Materials Types list box widget. Each one of these lists of strings corresponds (in order) to the string names in the first list widget (Materials). These tables are numbered to facilitate programming. When the user selects an item in the Materials list widget, the index of the selected item will be concatenated with the string *cst_type_* to form the name of one of these tables. This named table will be retrieved with the **MrmFetchSetValues** subroutine and placed in the Materials Type list widget.
General Comment:	In addition to the string table, a count of the number of items in the table is declared as an exported value. This is done because using the Enhanced X–Windows Intrinsics **XtSetValues** subroutine (in **libXt.a**) on a list widget requires that three parameters be set: **XmNitems**, **XmNitemCount**, and **XmNselectedItemsCount** (which must be set to zero).

In the following C program segment, the **tst_single_proc** activation subroutine enables the selection made by the user to cause the application to act.

```
 #define k_zero_name "k_zero"
 #define k_table_name_prefix "cst_type_"
 #define k_table_count_name_prefix "l_count_type_"
(1)
 void tst_single_proc(w,object_index,callbackdata)
    Widget              w;
    int                 *object_index;
    XmListCallbackStruct *callbackdata;
 {
(2)
    char *t_number;
(3)
    char t_table_name[32] = k_table_name_prefix;
    char t_table_count_name[32] = k_table_count_name_prefix;
(4)
    Arg r_override_arguments[3] =
    {{XmNitems,NULL},{XmNitemsCount,NULL},
                                  {XmNselectedItemsCount,k_zero_name}};
    switch (*object_index)
    {
```

```
(5)
      case k_tst_materials_ListBox:
      {
(6)
        sprintf(&t_number,"%d",callbackdata->item_number);
(7)
        strcpy(&t_table_name[sizeof(k_table_name_prefix)-1],
                                                 &t_number);
      XtSetArg(r_override_parameters[0],XmNitems,&t_table_name);
(8)
        strcpy(&t_table_count_name[sizeof
                   (k_table_count_name_prefix)-1],&t_number);
      XtSetArg(r_override_arguments[1],XmNitemsCount,
                              &t_table_count_name);
(9)
        MrmFetchSetValues(ar_MRMHierarchy,
        object_ids[k_tst_types_ListBox],
        r_override_arguments,3);
      break;
    };
(10)
       case k_tst_types_ListBox:
   {
   .
   .
   .
       break;
   };
  };
};
```

The following comments apply to the preceding source code example:

Code Segment #1:	This subroutine handles the **XmNsingleSelectionCallback** callback subroutines for any object. When the user selects an item in a list box widget, the contents of the neighboring list box are widget replaced. This subroutine uses the list box widget callback structure named **XmListCallbackStruct**. This structure contains the following fields: reason, event, item, item_length, and item_number.
Code Segment #2:	Forms the string version of the item number.
Code Segment #3:	Local character storage.
Code Segment #4:	Override parameter list for the **MrmFetchSetValues** subroutine.
Code Segment #5:	User has selected an item from the Materials list box widget. The application needs to place a new items list in the Types list widget. The string tables stored in the UID file are named *cst_type_*"index number" and their count names are *l_count_type_*"index number" (where "index number" corresponds to the item's position in the list widget). Using the index of the selected item from this list, the application forms the name of the appropriate compound string table.
General Comment:	Using the item number instead of the text value of the selection separates the function of the application from the form (in this case, the contents of the list widgets) and reduces complexity. If the application used the text value of the selected item as the means to determine what to display, it would need to deal with possible invalid characters for a UIL name in the text and would have to convert the text value (a compound string) to a

null–terminated string so that the string could be passed to the Enhanced X–Windows Intrinsics **XtSetValues** subroutine (in **libXt.a**).

Code Segment #6:	Forms the string version of the item number.
Code Segment #7:	Forms the name of the string table.
Code Segment #8:	Forms the name of the string table count.
Code Segment #9:	Fills the types list widget with a new list of items.
Code Segment #10:	Similar selection recording code goes here.

How to Use an Object Definition as a Template

The **MrmFetchWidgetOverride** subroutine is useful if you have to create many widgets that are very similar. Consider an application interface that has a large number of push buttons contained in a bulletin board. The push buttons are the same except for their **XmNy** position and label. Instead of declaring each push button individually, you can declare one push button and use the MRM **MrmFetchWidgetOverride** subroutine to use that declaration as a template, modifying the **XmNy** position and label at run time.

By calling the **MrmFetchWidgetOverride** subroutine instead of the **MrmFetchWidget** subroutine, the application program creates the widget and overrides the original values in the declaration with the values specified in the **MrmFetchWidgetOverride** call. The parameter list you supply to the **MrmFetchWidgetOverride** subroutine must contain AIXwindows Toolkit attribute names; therefore, it is also possible to override the callbacks for an object using the **MrmFetchWidgetOverride** subroutine (since callbacks are AIXwindows Toolkit resources).

To use the **MrmFetchWidgetOverride** subroutine in an application, you should use UIL identifiers to specify the tag value for each callback procedure. The tag is specified in the callback structure and cannot be changed unless the callback is deleted and replaced. The callback structure is not stored in the widget data, but by the Enhanced X–Windows Toolkit Intrinsics.

If you do not use identifiers for tag values, your callback procedures must contain a check for the parent of the calling widget or some other field of the widget (as opposed to checking only the tag value) because it is not possible to override the tag value with the **MrmFetchWidgetOverride** subroutine. If you do not use an identifier for the tag value, all instances of the fetched object return identical tag values for all callbacks. If the callback subroutine checks only the tag value, the callback subroutine could not distinguish which instance made the call.

Another practical use of the **MrmFetchWidgetOverride** subroutine is to create objects with parameters whose values can only be determined at run time (that is, are not known at UIL Compilation time).

The MRM subroutine **MrmFetchSetValues** works like the **MrmFetchWidgetOverride** subroutine. The **MrmFetchSetValues** subroutine, like the **MrmFetchWidgetOverride** subroutine, accepts a vector of name and value pairs. This vector is passed as the *override_args* parameter for the **MrmFetchWidgetOverride** subroutine and as the *args* parameter for the **MrmFetchSetValues** subroutine. For the **MrmFetchWidgetOverride** subroutine, the name and value pairs consist of the AIXwindows Toolkit attributes name and an explicit value for that attribute.

The value member of the name and value pairs passed to the **MrmFetchSetValues** subroutine is the UIL name of the value, not an explicit value. When the application calls the **MrmFetchSetValues** subroutine, MRM looks up the names in the UID file, then uses the values corresponding to those names to override the original values in the object declaration. The **MrmFetchSetValues** subroutine, therefore, allows you to keep all values used in an application in the UIL Module, and not in the application program.

Note: The values you pass to the **MrmFetchSetValues** subroutine must be named, exported literals in the UIL Module.

The **MrmFetchSetValues** subroutine is a convenience subroutine that packages the subroutines provided by the **MrmFetchLiteral** and **MrmFetchWidgetOverride** subroutines.

Suggested Reading

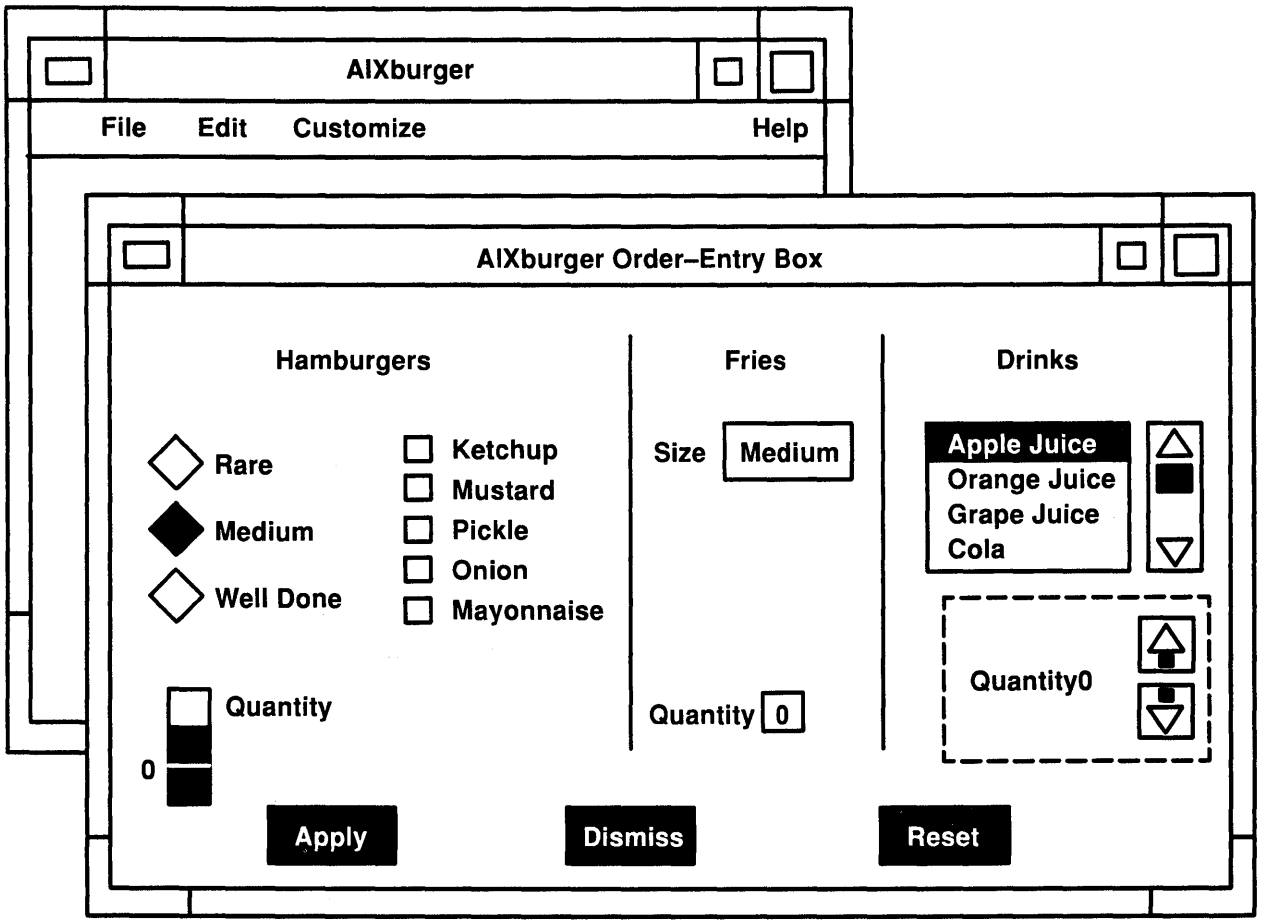
Related Information

Using AIXwindows Resource Manager (MRM) Subroutines contains additional informantion about using MRM subroutines, including the **MrmInitialize, MrmOpenHierarchy, MrmRegisterNames, MrmFetchWidget, MrmFetchColorLiteral, MrmFetchIconLiteral, MrmFetchLiteral, MrmFetchSetValues,** and **MrmFetchWidgetOverride** subroutines.

Using Icons in the AIXwindows User Interface Language (UIL)

Using an Icon as a Widget Label

The drink quantity selector in the user interface for the **AIXburger** application is a good example of a UIL interface in which a widget uses icons for the labels on its push buttons.



Using an Icon in the **AIXburger** Application Interface

When the user clicks on the up arrow icon, the drink quantity increases. When the user clicks on the down arrow icon, the drink quantity decreases.

Using an Icon in the AIXburger Application Interface

The following example shows how to specify the icon named *drink_up_icon* as a pushbutton label in a widget declaration:

Note: In the **AIXburger** UIL Module, the icon named *drink_up_icon* is a pixmap label parameter to the *up_value* push button widget. In turn, the *up_value* widget is controlled by the *drink_quantity* form widget.

```
object
    drink_quantity : XmForm {
        parameters {
            XmNx = 460;
            XmNy = 170;
            XmNunitType = XmPIXELS; };
        controls {
            XmLabel quantity_label;
            XmLabel value_label;
            XmPushButton up_value;
            XmPushButton down_value;
            };
        };
.
.
.
object
    up_value : XmPushButton widget {
        arguments {
            XmNy = 00 ;
            XmNleftAttachment = XmATTACH_WIDGET;
            XmNleftOffset = 20 ;
            XmNleftWidget = XmLabel value_label ;
            XmNlabelType = XmPIXMAP;
            XmNlabelPixmap = drink_up_icon;
            };
        callbacks {
            XmNactivateCallback = procedure activate_proc
                (k_drink_add);
            };
    };
```

How to Compile an AIXwindows User Interface Language (UIL) Specification File to Generate a User Interface Definition (UID) File

Prerequisite Tasks or Conditions

1. Ensure that the UIL Compiler and the include files and other header files listed in your source code are available and accessible.
2. Create one or more UIL Modules and store them in a UIL Specification File.

Procedure

To compile your UIL Specification File and generate a UID file, complete the following steps in the order shown:

1. Invoke the UIL Compiler with the following syntax:

   `uil [ option... ]` *input–file*

   The *input file* is the UIL Specification File to be compiled. Use the following command flags and options to control the output of the UIL Compiler:

Cmd. Flag	Options	Default	Description
–o	*filename*	**a.uid**	Directs the compiler to produce a User Interface Definition (UID) file. By default,

			UIL creates a UID file named **a.uid**. No UID file is produced if the compiler issues any diagnostics categorized as error or severe.
–v	*filename*	No listing is generated	Directs the compiler to produce a listing file. If the **–v** option is not present, no listing is generated by the compiler.
–m		No "machine" code is listed	Directs compiler to place in the listing file a description of the records that it added to the UID file. This helps you isolate errors.
–w		Warning messages are generated	Directs the compiler to suppress all warning and informational messages. Error messages and severe messages generated.
–I	*pathname*	**/usr/include**	If you specify the **–I** option followed by a pathname, with no intervening spaces, the compiler uses specified pathname to locate include files when the include directive is used. A trailing slash (/) on specified pathname is optional; if it is omitted, the compiler inserts it for you.

For example, the **–I** flag causes the compiler to look for include files in the **/usr/include/myuilpath** directory, as demonstrated in the following example:

```
uil –I/usr/include/myuilpath input–file
```

2. Interpret diagnostics issued by the UIL Compiler

The UIL Compiler issues diagnostics to the standard error file. See Appendix B for a listing of UIL Compiler diagnostic messages. The following example shows the form of these messages:

```
value d: font( 1 );
                 *

Error: found integer value when expecting string value
             line: 10 file: value_error.uil
```

The first line is the source line that produced the diagnostic. If for some reason the compiler cannot retrieve the source line from the input file, this line is omitted. Any control characters in the text of the line are printed as question marks (?).

The second line of the diagnostic marks the start of the construct that resulted in the diagnostic. In this case the literal 1 is marked. If the error is not associated with a particular construct, the second line is omitted. If the source line cannot be retrieved, this line is also omitted and the column information is included with the line and file information following the diagnostic message.

The third line is the diagnostic being issued.

The fourth line specifies the file containing the source line being diagnosed and the line number of the source line within that file.

The following four levels of diagnostics are issued by the compiler:

Severity	**Compilation Status**
Informational	Accompanies other diagnostics that should be investigated.

Warning	Compilation continues: check that the compiler did what you expected.
Error	Compilation continues: no UID file will be generated.
Severe	Compilation terminates immediately.

3. Read the UIL Compiler listing

The listing produced by the compiler contains the following information:

- Title

Each new page of the listing starts with a title that gives miscellaneous information about the compilation. The first line of the title identifies the UIL Compiler by name, the version of the compiler used for the compilation, and the time the compilation started. The second line of the title lists the module name for the UIL specification and the version of that Module if provided in the source.

- Source lines of the input file

The printing of each source line is preceded by two numbers:

```
1 (0) Module example VERSION = 'X—1'
```

The number in parentheses designates the file from which the source line was read. By looking at the file summary at the end of the listing, you can see that file (0) is **a.uil**. The first number on the source line is the line number within the source file. If a source line contains any control characters other than tabs, they are replaced in the listing by questions marks (?).

- Source lines of any include files
- Diagnostics issued by the UIL Compiler

Diagnostics for a particular source line follow that line in the listing as indicated in the following example:

```
  2(1)       VALUE red: COLOR( 1 ); VALUE green: COLOR( 2 );
                  1             2
Error: (1) found integer value when expecting string value
Error: (2) found integer value when expecting string value
   8 (0)              { XmNforeground = pink;
                           1
Error: (1) value PINK must be defined before this reference
Error: (1) found error value when expecting color value
```

The line following the source line points to the position in the source line where each of the diagnostics occurred. You can determine the position of each diagnostic by looking at the number in parentheses that follows the diagnostic severity. In this example, diagnostic 1 for source line 2 is at position (1), and diagnostic 2 for the same source line is at position (2). The diagnostics for source line 8 both occur at position (1). If a diagnostic has no associated position on the line, the position is given as (0). If a diagnostic is not associated with a source line, it appears following the last source line. See Appendix B for a listing of UIL Compiler diagnostic messages.

- A summary of the diagnostics issued and a summary of the files read

The listing contains the following summaries:

```
Info: (0) errors: 5 warnings: 0 informationals: 0
    file (0) a.uil
    file (1) colors.uil
```

The first summary is in the form of a diagnostic that tallies the number of error, warning, and informational diagnostics. The second summary lists the files that contributed to the UIL specification. This list is useful in determining from which file a source line in the listing was read. A source line preceded by the sequence 532 (3) in the listing was read from line 532 of file (3) in the summary list.

The following example shows the contents of a sample listing file:

```
AIXwindows UIL Compiler V1.0–000 Wed Dec 16 11:13:31 1989 Pg1
Module: EXAMPLE Version: X–1
  1 (0)   Module example VERSION = 'X–1'
  2 (0)
  3 (0)   INCLUDE FILE 'colors.uil';
  1 (1)
  2 (1)   VALUE red: COLOR( 1 ); VALUE green: COLOR( 2 );
                1              2
Error: (1) found integer value when expecting string value
Error: (2) found integer value when expecting string value
  3 (1)   VALUE blue: COLOR( 'XcolorBlue' );
  4 (1)
  4 (0)
  5 (0)   OBJECT primary_window:
  6 (0)    XmMainWindow
  7 (0)     { ARGUMENTS
  8 (0)       { XmNforeground = pink;
                      1
Error: (1) value PINK must be defined before this reference
Error: (1) found error value when expecting color value
  9 (0)         XmNbackground = blue; }};
                      1
Error: (1) unexpected RIGHT_BRACE token seen – parsing resumes
           after ";"
 10 (0)                 };
 11 (0) END Module;
Info: (0) errors: 5 warnings: 0 informationals: 0
  File (0) a.uil
  File (1) colors.uil
```

Suggested Reading

Related Information

How to Create a User Interface at Run Time Using the AIXwindows Resource Manager (MRM) contains detailed information about the AIXwindows Resource Manager.

Using AIXwindows Resource Manager (MRM) Subroutines contains additional information about the MRM subroutines.

Using Recommended Coding Techniques in AIXwindows User Interface Language (UIL)

The language elements and semantics of the AIXwindows User Interface Language (UIL) are similar to those of other high–level programming languages. The sample source code for the AIXburger UIL Module uses recommended coding practices to increase the flexibility and utility of the resulting user interface.

Naming Values and Objects Meaningfully

The names of constants, labels, colors, icons, and widgets in the AIXburger UIL Module indicate their purpose in the application. For example, the name for the constant having integer value 12 is k_burger_rare. From its name, you can tell that this constant represents the choice Rare on the Hamburgers menu.

Similarly, the names for objects (widgets and gadgets) indicate their purpose in the application. Object names should, in addition, reflect the object type. For example, you can tell by its name that *m_copy_button* is a button of some kind on a menu and is associated with the Copy option.

Grouping Value, Identifier, and Procedure Declarations

You should group value declarations according to purpose and list them near the beginning of the module. The UIL Compiler requires only that values be declared before you reference them.

In the sample source code for the AIXburger UIL Module, separate value sections are used to group values as follows:

- Constants for positioning within forms
- Constants for callback parameters
- Labels
- Fonts
- Colors
- Color tables
- Icons

Constants for callback procedures must be defined in the program as well as in the UIL Module. Therefore, if these constants are in a single value section, it is easier to cut the section from the module and paste it into the application program.

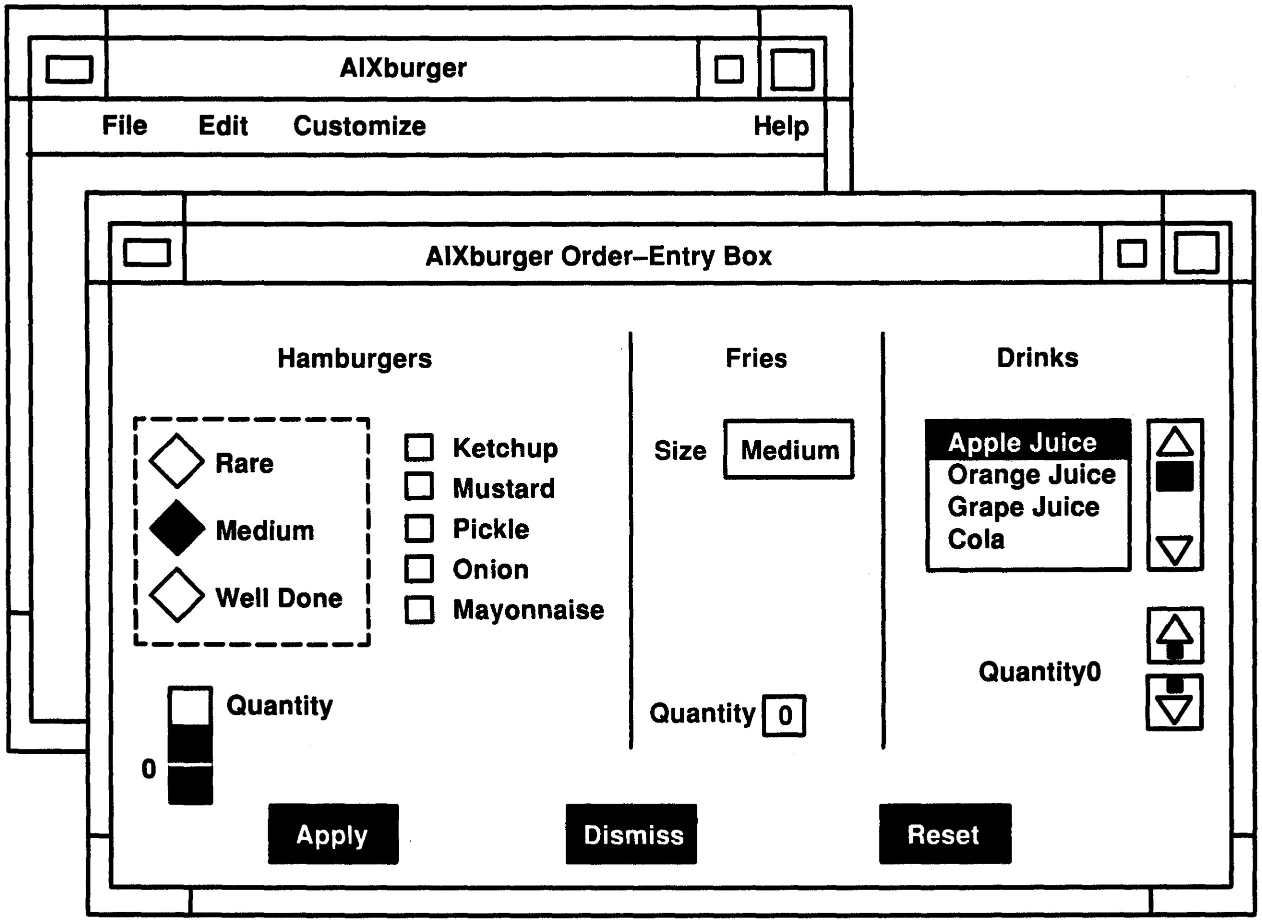
By setting up all labels as compound string values, rather than hardcoding them in the object declarations, you can more easily change the labels from one language to another. Specify a string as a compound string by using the UIL built–in **COMPOUND_STRING** subroutine. Some parameters for the simple text widget and the command window widget accept only null–terminated strings. Labels for these widgets must be declared as null–terminated (ASCIZ) strings, delimited with single quotation marks.

The same technique applies to procedure declarations. In the AIXburger UIL Module all procedure declarations are listed in a single procedure section at the beginning of the module, immediately following the module declaration and include directive.

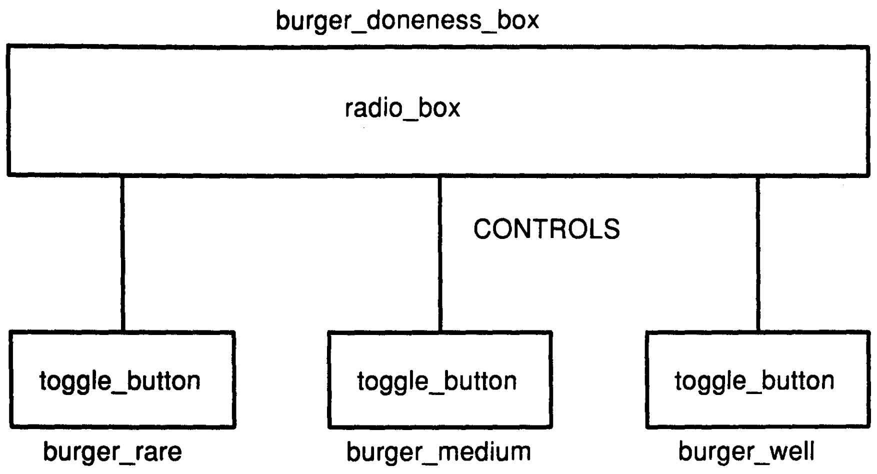
The AIXburger application does not use identifiers. Treat identifier sections as you would treat value sections.

Ordering Object Declarations to Reflect the Widget Tree

Once all your values, identifiers, and procedures are declared, the rest of the UIL Module consists of object declarations. You should structure your Module to reflect the widget tree of the application interface. For example, in the AIXburger UIL Module, the choices for how the hamburger should be cooked are presented in an **XmRadioBox** widget that has three children widgets, all of which are **XmToggleButton** widgets.



The widgets are arranged in a hierarchy, which is defined by the controls list for the radio box named *burger_doneness_box.*



The following example shows the object declaration in the UIL Module for the *burger_doneness_box* widget. Notice that the children of the **XmRadioBox** (the three toggle button widgets named in the controls list) are declared immediately following the declaration of the **XmRadioBox** widget. By ordering object declarations in this way, you mirror the overall widget tree for your interface within the UIL Module.

```
object
    burger_doneness_box : XmRadioBox {
        parameters {
            .
            .
            .
            };
        controls {
            XmToggleButton burger_rare;
            XmToggleButton burger_medium;
            XmToggleButton burger_well;
            };
        };
object
    burger_rare : XmToggleButton {
            .
            .
            .
            };
object
    burger_medium : XmToggleButton {
            .
            .
            .
            };
```

```
object
    burger_well : XmToggleButton {
                .
                .
                .
                };
```

Using Local Definitions for Certain Objects

If you need to define an object that is used as a child of a single parent and will not be referred to by any other object in the UIL Module, define the object in the controls list for its parent rather than in an object section of its own. This simplifies the UIL Module and saves you from having to create an artificial name for that object.

How to Customize an AIXwindows Interface using UIL and MRM

UIL offers the advantage of separating the form an interface takes from the subroutines of the application. The form of the interface can change, while the subroutines the application performs remain the same. By specifying these varying forms of the interface in separate UIL Modules, you can change the interface by changing the definition of the UID hierarchy (the set of UID files) in the application program, and recompiling and relinking the application.

Consider the UIL Module in the following example, which shows the compound string literals for the **AIXburger** interface translated into French.

Note: This is a separate UIL Module, not an edited version of the original **AIXburger** UIL Module.

```
Module french_literals
    version = 'v1.0'
    names = case_sensitive
value
    k_aixburger_title             : exported "AIXburger -
                                    Commandes";
    k_nyi_label_text              : exported "Fonction non
                                    disponible";
    k_file_label_text             : exported "Fichier";
        k_quit_label_text         : exported "Quitter";
    k_edit_label_text             : exported "Edition";
        k_cut_dot_label_text      : exported "Couper";
        k_copy_dot_label_text     : exported "Copier";
        k_paste_dot_label_text    : exported "Coller";
        k_clear_dot_label_text    : exported "Effacer tout";
        k_select_all_label_text   : exported
                                    "S\o'e\(aa'lectionner tout";
    k_order_label_text            : exported "Commande";
        k_show_controls_label_text : exported "Voir codes...";
        k_cancel_order_label_text : exported "Annuler commande";
        k_submit_order_label_text : exported "Transmettre
                                    commande";
    k_hamburgers_label_text       : exported "Hamburgers";
        k_rare_label_text         : exported "Saignant";
        k_medium_label_text       : exported "A point";
        k_well_done_label_text    : exported "Tr\o'e\(ga's cuit";
        k_ketchup_label_text      : exported "Ketchup";
        k_mustard_label_text      : exported "Moutarde";
        k_onion_label_text        : exported "Oignons"
        k_mayonnaise_label_text   : exported "Mayonnaise";
        k_pickle_label_text       : exported "Cornichons";
        k_quantity_label_text     : exported "Quantit\o'e\(aa'";
    k_fries_label_text            : exported "Frites";
        k_size_label_text         : exported "Taille";
        k_tiny_label_text         : exported "Minuscule";
        k_small_label_text        : exported "Petit";
        k_large_label_text        : exported "Gros";
        k_huge_label_text         : exported "Enorme";
    k_drinks_label_text           : exported "Boissons";
        k_0_label_text            : exported '0';
        k_drink_list_text         : exported
            string_table ('Jus de pomme', 'Jus d\'orange', 'Jus de
                          raisin', 'Cola', 'Punch', 'Root beer',
                          'Eau', 'Ginger Ale', 'Lait',
                          'Caf\o'e\(aa'', 'Th\o'e\(aa'');
        k_drink_list_select       : exported string_table("Jus de
                                    pomme");
        k_u_label_text            : exported "U";
        k_d_label_text            : exported "D";
    k_apply_label_text            : exported "Appliquer";
    k_reset_label_text            : exported "Remise
                                    \o'a\(ga' 0";
    k_cancel_label_text           : exported "Annulation";
    k_dismiss_label_text          : exported "Termin\o'e\(aa'";
end Module;
```

To generate a French version of the **AIXburger** application, perform the following steps in the order shown:

1. Change all string literal declarations in the original **AIXburger** UIL Module to be imported compound strings. For example, change the value declaration for the Fries label as follows:

   ```
   k_fries_label_text : imported compound_string;
   ```

   As shown in the previous example, the French UIL Module specifies the corresponding values as exported and gives their definitions.

2. In the original C program for the **AIXburger** application, specify the name of the UID file containing the compiled French UIL Module as the first element of the UID hierarchy array. Assume the name of the UIL specification file containing the French strings is **french_literals.uil**. Change the UID hierarchy array definition as follows:

   ```
   static char *db_filename_vec[] =
       {   "french_literals.uid",
           "aixburger.uid"
       };
   ```

3. Add a line to the script that compiles the French UIL Module, and execute the script.

Suggested Reading

Related Information

Using AIXwindows Resource Manager (MRM) Subroutines contains additional information about the MRM subroutines.

Using the AIXwindows User Interface Language (UIL) Callable Interface Subroutine and the Symbol Table Dump Subroutine

The AIXwindows User Interface Language (UIL) provides two related subroutines that can both be invoked from within an application. The **Uil** subroutine enables you to invoke the UIL Compiler from within an application. This subroutine also returns a data structure describing the User Interface Description (UID) File that was compiled by the UIL Compiler. The **UilDumpSymbolTable** subroutine performs a symbol table dump from within an application.

The UIL Callable Interface Subroutine

Description

The **Uil** subroutine is also known as the Callable Interface subroutine because it provides a callable entry point for the UIL Compiler. The subroutine process a UIL Specification file and generates a UID File, as well as returning a detailed description of the UIL source Module in the form of a symbol table called a parse tree.

Syntax Statement

The **uil** subroutine has the following syntax:

```
#include <uil/UilDef.h>
Uil_status_type Uil(command_desc, compile_desc, message_cb,
message_data, status_cb, status_data)
     Uil_command_type       * command_desc;
     Uil_compile_desc_type  * compile_desc;
     Uil_continue_type       (* message_cb ) ();
    char                    * message_data;
     Uil_continue_type       (* status_cb ) ();
     char                     * status_data ;
```

Parameter	Description
command_desc	Specifies the **uil** command line.
compile_desc	Returns the results of the compilation.
message_cb	Specifies a callback subroutine that is called when the UIL Compiler encounters errors in the UIL source.
message_data	Specifies user data that is passed to the message callback subroutine (**message_cb**). Note that this parameter is not interpreted by UIL and is used exclusively by the calling application.
status_cb	Specifies a callback subroutine that is called to allow Enhanced X–Windows applications to service X events such as updating the screen. This subroutine is called at various check points, which have been hard coded into the UIL Compiler. The *status_update_delay* parameter in command_desc specifies the number of check points to be passed before the **status_cb** subroutine is invoked.
status_data	Specifies user data that is passed to the status callback subroutine (**status_cb**). This parameter is not interpreted by the UIL Compiler and is used exclusively by the calling application.

The data structures `Uil_command_type` and `Uil_compile_desc_type` are detailed in the following example.

```
typedef struct Uil_command_type
{
  char      *source_file;          /* single source to compile */
  char      *resource_file;        /* name of output file */
  char      *listing_file;         /* name of listing file */
  unsigned  int *include_dir_count; /* number of directories in*/
                                   /*include_dir array */
  char      *((*include_dir)[]);   /* directory to search for */
                                   /* include files */
  unsigned  listing_file_flag: 1;  /* produce a listing */
  unsigned  resource_file_flag: 1; /* generate UID output */
  unsigned  machine_code_flag: 1;  /* generate machine code */
  unsigned  report_info_msg_flag: 1;/* report info messages */
  unsigned  report_warn_msg_flag: 1;/* report warnings */
  unsigned  parse_tree_flag: 1;    /* generate parse tree */
  unsigned  int status_update_delay;/* number of times a */
                                   /* status point is passed */
                                   /* before calling status_cb */
                                   /* subroutine. 0 means */
                                   /* called every time*/
}
typedef struct Uil_compile_desc_type
{
  unsigned  int compiler_version;/*version num.of compiler*/
  unsigned  int data_version;    /*version num. of structures */
  char      *parse_tree_root;    /* parse tree output */
  unsigned  int message_count [Uil_k_max_status+1];
                                 /* array of severity counts /*
}
```

The **Uil** subroutine returns one of the following status return values:

Status Return Constant	Meaning
Uil_k_success_status	The operation succeeded.
Uil_k_info_status	The operation succeeded, and an informational message is returned.
Uil_k_warning_status	The operation succeeded, and a warning message is returned.
Uil_k_error_status	The operation failed due to an error.
Uil_k_severe_status	The operation failed due to an error.

The UilDumpSymbolTable Subroutine

Description

The **UilDumpSymbolTable** subroutine dumps the contents of a named UIL symbol table whose pointer was returned by the **Uil** callable interface subroutine. By following the link from the root entry, you can traverse the entire parse tree.

Syntax Statement

The **UilDumpSymbolTable** subroutine requires the following syntax:

```
#include <uil/UilDef.h>
void UilDumpSymbolTable UIL( root_ptr )
sym_root_entry_type * root_ptr ;
```

Parameter	Description
root_pst	Returns a pointer to the symbol table root entry.

UIL Symbol Table Format

Symbol table entries are stored in the following format:

- hex.address symbol.type symbol.data
- prev.source.position
- source.position
- modification.record

Symbol Table Entry	**Description**
hex.address	Specifies the hexadecimal address of this entry in the symbol table.
symbol.type	Specifies the type of this symbol table entry. Some possible types are root, module, value, procedure, and widget.
symbol.data	Specifies data for the symbol table entry. The data varies with the type of the entry. Often it contains pointers to other symbol table entries, or the actual data for the data type.
prev.source.position	Specifies the end point in the source code for the previous source item.
source.position	Specifies the range of positions in the source code for this symbol.

Note: The exact data structures for each symbol type are defined in the **UilSymDef.h file, a** header file that is automatically included when an application includes the **UilDef.h** file.

Sample Symbol Table Dump

This section contains the complete symbol table dump for a sample application named **hellos**. Symbol table entries are presented in the format returned by the **UilDumpSymbolTable** subroutine:

Note: Each line of sample source code is numbered in parentheses. A brief description of the relationship of each line to the subsequent symbol table dump follows the sample source code:

```
(1)  module helloaix
(2)    version = 'v1.0'
(3)    names = case_sensitive
(4)  procedure
(5)    helloaix_button_activate();
(6)  object
(7)    helloaix_main : bulletin_board {
(8)      controls {
(9)        labels helloaix_label;
(10)       push_button helloaix_button;}; };
(11) object
(12)   helloaix_button : push_button {
(13)     arguments {
(14)       x = 15;
(15)       y = 60;
(16)       label_string =
             compound_string('Hello',separate=true) & 'aix!'; };
(17)     callbacks {
```

```
(18)          activate = procedure helloaix_button_activate();};
          };
(19) object
(20)    helloaix_label : label {
(21)      arguments {
(22)        label_string =
                compound_string('Press button once',separate=true)
              & compound_string('to change label;',separate=true)
              &'twice to exit.': }; };
```

Notes:

Line #1:	Module declaration for **helloaix** program. This corresponds to address e26b0 in the symbol table dump.
Line #2:	Specifies version of the UIL Module. This corresponds to address e2754 in the symbol table dump.
Line #3:	Specifies whether names or keywords in the UIL Module are case sensitive.
Line #4:	Procedure section. This corresponds to address e2830 in the symbol table dump.
Line #5:	Declaration of subroutine to be used as a callback subroutine. This corresponds to address e28d0 in the symbol table dump.
Line #6:	Object section. This corresponds to address e2968 in the symbol table dump.
Line #7:	Declaration of helloaix_main bulletin board widget. This corresponds to address e2a00 in the symbol table dump.
Line #8:	Control list declaration for bulletin board widget. This corresponds to address e2adc in the symbol table dump.
Line #9:	Specification of label widget as a child of the bulletin board widget. This corresponds to address e2d7c in the symbol table dump.
Line #10:	Specification of push button widget as a child of the bulletin board widget. This corresponds to address e2c2c in the symbol table dump.
Line #11:	Second object section. This corresponds to address e2e38 in the symbol table dump.
Line #12:	Declaration of push button widget. This corresponds to address e2e84 in the symbol table dump.
Line #13:	Argument list for push button widget. This corresponds to address e2f60 in symbol table dump.
Line #14:	The specification is a built–in private subroutine. This corresponds to address e3060 in the symbol table dump.
Line #15:	The specification is a built–in private subroutine. This corresponds to address e311c in the symbol table dump.
Line #17:	The specification is a built–in private subroutine. This corresponds to address e33a0.
Line #18:	Specification of callbacks for push button widget. This corresponds to address e32bc in the symbol table dump.
Line #19:	Callback entry declaration. This corresponds to address e34e8.

Line #20:	Third object section. This corresponds to address e35ac in the symbol table dump.
Line #21:	Declaration of label widget. This corresponds to address e35f8 in the symbol table dump.
Line #22:	Argument list for label widget. This corresponds to address e36d4 in the symbol table dump.
Line #23:	The specification is a built–in private compound string. This corresponds to address e3788) in the symbol table dump.

The following symbol table dump is the symbol table dump for the **helloaix** sample application:

```
e01d4 root tag: 17 module: e26fc sections: e35ac module tail:
  Beginning source record not present. End source record not
  present. Modified: 0, Child Modified: 0.

e0428 tail section prev section : 0 next section: 0 entries: 0
  Beginning source record not present. End source record not
  present. Modified: 0, Child Modified: 0.

e26b0 name size: 58 next name: 0 object: e26fc name: HELLOAIX
  Previous item ended: file 0, line 0, column 0 Source position:
  file 0, line 21, columns 7 through 17 Modified: 0, Child
  Modified: 0.

e26fc module size: 52 name: e26b0 version: e2754 Beginning source
  record not present. End source record not present. Modified: 0,
  Child Modified: 0.

e2754 value size: 61 name: 0 built-in private string length: 4
  chariest: ** unknown ** R–>L: 0

    "v1.0"

  Previous item ended: file 0, line 21, column 18 Source position:
  file 0, line 22, columns 14 through 20 Modified: 0, Child
  Modified: 0.

e27bc value size: 56 name: 0 error Previous item ended: file 0,
  line 22, column 21 Source position: file 0, line 23, columns 12
  through 26 Modified: 0, Child Modified: 0.

e2830 procedure section prev section : 0 next section: e0428
  entries: e2900 Previous item ended: file 0, line 23, column 27
  Source position: file 0, line 25, columns 0 through 9 Modified:
 0, Child Modified: 0.

e287c name size: 74 next name: 0 object: e28d0 name:
  helloaix_button_activate Beginning source record not present.
  End source record not present. Modified: 0, Child Modified: 0.

e28d0 proc def size: 40 name: e287c check private count: 0 void
  Previous item ended: file 0, line 25, column 10 Source position:
  file 0, line 26, columns 32 through 33 Modified: 0, Child
  Modified: 0.

e2900 section prev section : 0 next section: 0 entries: e28d0
  Beginning source record not present. End source record not
  present. Modified: 0, Child Modified: 0.
```

```
e2968 object section prev section : 0 next section: e2830 entries:
  e2a60 Previous item ended: file 0, line 26, column 34 Source
  position: file 0, line 28, columns 0 through 6 Modified: 0,
  Child Modified: 0.

e29b4 name size: 63 next name: 0 object: e2a00 name:
  helloaix_main Beginning source record not present. End source
  record not present. Modified: 0, Child Modified: 0.

e2a00 widget size: 92 name: e29b4 private bulletin board widget
  controls: e2adc callbacks: 0 arguments: 0 parent_list: 0
  Previous item ended: file 0, line 28, column 7 Source position:
  file 0, line 34, columns 5 through 6 Modified: 0, Child
  Modified: 0.

e2a60 section prev section : 0 next section: 0 entries: e2a00
  Beginning source record not present. End source record not
  present. Modified: 0, Child Modified: 0.

e2a90 external def size: 40 next external: 0 object: e29b4
  Beginning source record not present. End source record not
  present. Modified: 0, Child Modified: 0.

e2adc list size: 68 next: e2d7c type: control count: 2 gadget
  count: 0 Previous item ended: file 0, line 29, column 39 Source
  position: file 0, line 33, columns 2 through 3 Modified: 0,
  Child Modified: 0.

e2b44 widget size: 92 reference: e35f8 label widget controls: 0
  callbacks: 0 arguments: 0 parent_list: 0 Previous item ended:
  file 0, line 30, column 12 Source position: file 0, line 31,
  columns 27 through 28 Modified: 0, Child Modified: 0.

e2ba4 name size: 64 next name: 0 object: e35f8 referenced name:
  helloaix_label Beginning source record not present. End source
  record not present. Modified: 0, Child Modified: 0.

e2c2c control size: 68 managed  obj: e2b44 Beginning source record
  not present. Source position: file 0, line 31, columns 27
  through 28 Modified: 0, Child Modified: 0.

e2c94 widget size: 92 reference: e2e84 push button widget
  controls: 0 callbacks: 0 arguments: 0 parent_list: 0 Previous
  item ended: file 0, line 31, column 29 Source position: file 0,
  line 32, columns 34 through 35 Modified: 0, Child Modified: 0.

e2cf4 name size: 65 next name: 0 object: e2e84 referenced name:
  helloaix_button Beginning source record not present. End
  source record not present. Modified: 0, Child Modified: 0.

e2d7c control  size: 68  next: e2c2c managed  obj: e2c94 Beginning
  source record not present. Source position: file 0, line 32,
  columns 34 through 35 Modified: 0, Child Modified: 0.

e2e38 object section prev section : 0 next section: e2968 entries:
  e2ee4 Previous item ended: file 0, line 34, column 7 Source
  position: file 0, line 36, columns 0 through 6 Modified: 0,
  Child Modified: 0.

e2e84 widget size: 92 name: e2cf4 private push button widget
  controls: 0 callbacks: e32bc arguments: e2f60 parent_list: e3ab4
  Previous item ended: file 0, line 36, column 7 Source position:
  file 0, line 46, columns 5 through 6 Modified: 0, Child
  Modified: 0.

e2ee4 section prev section : 0 next section: 0 entries: e2e84
  Beginning source record not present. End source record not
  present. Modified: 0, Child Modified: 0.

e2f14 external def size: 40 next external: e2a90 object: e2cf4
  Beginning source record not present. End source record not
  present. Modified: 0, Child Modified: 0.

e2f60 list size: 68 next: e33a0 type: argument count: 3 gadget
  count: 0 Previous item ended: file 0, line 37, column 38 Source
  position: file 0, line 42, columns 2 through 3 Modified: 0,
  Child Modified: 0.

e2fc8 value size: 60 name: 0 builtin private argument code: 654804
  Previous item ended: file 0, line 38, column 13 Source position:
  file 0, line 39, columns 5 through 7 Modified: 0, Child
  Modified: 0.

e3014 value size: 60 name: 0 builtin private integer: 15 Beginning
  source record not present. Source position: file 0, line 39,
  columns 7 through 8 Modified: 0, Child Modified: 0.

e3060 argument size: 72 arg name: e2fc8 arg value: e3014 Previous
  item ended: file 0, line 38, column 13 Source position: file 0,
  line 39, columns 11 through 12 Modified: 0, Child Modified: 0.

e30d0 value size: 60 name: 0 builtin private argument code: 654828
  Previous item ended: file 0, line 39, column 13 Source position:
  file 0, line 40, columns 5 through 7 Modified: 0, Child
  Modified: 0.

e311c value size: 60 name: 0 builtin private integer: 60 Beginning
  source record not present. Source position: file 0, line 40,
  columns 7 through 8 Modified: 0, Child Modified: 0.

e3168 argument size: 72 next: e3060 arg name: e30d0 arg value:
  e311c Previous item ended: file 0, line 39, column 13 Source
  position: file 0, line 40, columns 11 through 12 Modified: 0,
  Child Modified: 0.

e31d8 value size: 60 name: 0 builtin private argument code: 652188
  Previous item ended: file 0, line 40, column 13 Source position:
  file 0, line 41, columns 5 through 18 Modified: 0, Child
  Modified: 0.

e3224 value size: 62 name: 0 builtin private string length: 5
  charset: iso_arabic R->L: 0 next table entry: e3308

    "Hello"

  Beginning source record not present. Source position: file 0,
  line 41, columns 0 through 0 Modified: 0, Child Modified: 0.

e3270 value size: 60 name: 0 builtin private boolean: 1 Previous
  item ended: file 0, line 41, column 54 Source position: file 0,
  line 41, columns 53 through 57 Modified: 0, Child Modified: 0.
```

```
e3308 value size: 63 name: 0 builtin private string length: 6
  charset: ** unknown ** R->L: 0 next table entry: 0

    "World!"

  Beginning source record not present. Source position: file 0,
  line 41, columns 0 through 0 Modified: 0, Child Modified: 0.

e3354 value size: 60 name: 0 builtin private compound string first
  component: e9694 Beginning source record not present. Source
  position: file 0, line 41, columns 61 through 69 Modified: 0,
  Child Modified: 0.

e33a0 argument size: 72 next: e3168 arg name: e31d8 arg value:
  e3354 Previous item ended: file 0, line 40, column 13 Source
  position: file 0, line 41, columns 69 through 70 Modified: 0,
  Child Modified: 0.

e32bc list size: 68 next: e34e8 type: callback count: 1 gadget
  count: 0 Previous item ended: file 0, line 42, column 4 Source
  position: file 0, line 45, columns 2 through 3 Modified: 0,
  Child Modified: 0.

e3448 value size: 60 name: 0 builtin private reason code: 650160
  Previous item ended: file 0, line 43, column 13 Source position:
  file 0, line 44, columns 5 through 14 Modified: 0, Child
  Modified: 0.

e3494 procedure size: 72 e3494 proc ref size: 72 proc def: e28d0
  value: 0

  Beginning source record not present. End source record not
  present. Modified: 0, Child Modified: 0.

e34e8 callback  size: 76  reason name: e3448 proc ref: e3494 proc
  ref list: 0 Previous item ended: file 0, line 43, column 13
  Source position: file 0, line 44, columns 54 through 55
  Modified: 0, Child Modified: 0.

e35ac object section prev section : 0 next section: e2e38 entries:
  e3658 Previous item ended: file 0, line 46, column 7 Source
  position: file 0, line 48, columns 0 through 6

  Modified: 0, Child Modified: 0. e35f8 widget size: 92

  name: e2ba4 private label widget controls: 0 callbacks: 0
  arguments: e36d4 parent_list: e3ae4 Previous item ended: file 0,
  line 48, column 7 Source position: file 0, line 55, columns 5
  through 6 Modified: 0, Child Modified: 0.

e3658 section prev section : 0 next section: 0 entries: e35f8
  Beginning source record not present. End source record not
  present. Modified: 0, Child Modified: 0.

e3688 external def size: 40 next external: e2f14 object: e2ba4
  Beginning source record not present. End source record not
  present. Modified: 0, Child Modified: 0.

e36d4 list size: 68 next: e3788 type: argument count: 1 gadget
  count: 0 Previous item ended: file 0, line 49, column 31 Source
  position: file 0, line 54, columns 2 through 3 Modified: 0,
  Child Modified: 0.
```

```
e373c value size: 60 name: 0 builtin private argument code: 652188
  Previous item ended: file 0, line 50, column 13 Source position:
  file 0, line 51, columns 5 through 18 Modified: 0, Child
  Modified: 0.

e37dc value size: 60 name: 0 builtin private boolean: 1 Previous
  item ended: file 0, line 51, column 66 Source position: file 0,
  line 51, columns 65 through 69 Modified: 0, Child Modified: 0.

e38e4 value size: 60 name: 0 builtin private boolean: 1 Previous
  item ended: file 0, line 52, column 57 Source position: file 0,
  line 52, columns 56 through 60 Modified: 0, Child Modified: 0.

e39c8 value size: 90 name: 0 builtin private string length: 33
  charset: iso_arabic R—>L: 0 next table entry: e3890

  "Press button onceto change label;" Beginning source record not
  present. Source position: file 0, line 51, columns 0 through 0
  Modified: 0, Child Modified: 0.

e3890 value size: 71 name: 0 builtin private string length: 14
  charset: ** unknown ** R—>L: 0 next table entry: 0

  "twice to exit." Beginning source record not present. Source
  position: file 0, line 52, columns 0 through 0 Modified: 0,
  Child Modified: 0.

e3930 value size: 60 name: 0 builtin private compound string first
  component: e8e54 Beginning source record not present. Source
  position: file 0, line 53, columns 12 through 28 Modified: 0,
  Child Modified: 0.

e3788 argument size: 72 arg name: e373c arg value: e3930 Previous
  item ended: file 0, line 50, column 13 Source position: file 0,
  line 53, columns 28 through 29 Modified: 0, Child Modified: 0.

e3ab4 parent_list size: 40 parent: e2a00 next: 0 Beginning source
  record not present. End source record not present. Modified: 0,
  Child Modified: 0.

e3ae4 parent_list size: 40 parent: e2a00 next: 0 Beginning source
  record not present. End source record not present. Modified: 0,
  Child Modified: 0.

 size: 72  arg name: e373c arg value: e3930 Previous item ended:
  file 0, line 50, column 13 Source position: file 0, line 53,
  columns 28 through 29 Modified: 0, Child Modified: 0.

e3ab4 parent_list size: 40 parent: e2a00 next: 0 Beginning source
  record not present. End source record not present.
```

How to Create Your Own Widgets and Gadgets with the AIXwindows User Interface Language (UIL)

You can extend the AIXwindows Toolkit by defining and specifying your own widgets. In UIL the widgets you create in this manner are identified by the UIL object type **user_defined**. A user–defined widget can accept any UIL built–in parameter or callback reason. You can, if needed, use UIL to define your own parameters and callback reasons for a user–defined widget. You can specify any AIXwindows Toolkit object as a child of a user–defined widget.

The UIL Compiler has built–in parameters and callback reasons that are supported by objects in the AIXwindows Toolkit. A user–defined widget can be built using only standard AIXwindows Toolkit parameters and reasons as its resources. If your application interface requires a user–defined widget of this type, use the UIL built–in parameter names and callback reasons directly when you declare an instance of the widget. If the user–defined widget supports parameters and reasons that are not already built into the UIL Compiler, define these parameters and reasons with the UIL ARGUMENT and REASON subroutines before specifying them.

Prerequisite Tasks or Conditions

1. You must be familiar with the AIXwindows User Interface Language (UIL).
2. You must be acquainted with the C language data structures and subroutines in the AIXwindows Toolkit.

Procedure

To include a user–defined widget in an application interface using UIL, and to create the widget at run time using the AIXwindows resource Manager (MRM), complete the following steps in the UIL Module:

Note: The examples in this procedure are based on a previously built user–defined widget called the XYZ widget.

1. Identify and define In the UIL Module all parameters for the user–defined widget that are not already built into the UIL Compiler. The strings you pass to the ARGUMENT subroutine must match the names listed in the resource list structure in the widget class record for the XYZ widget. In addition to the strings, specify the data type of the parameter. Just as for built–in parameters, when you declare an instance of the XYZ widget in a UIL Module, the UIL Compiler checks the data type of the values you specify for these parameters. For example, the UIL Compiler checks that the value you specify for the *xyz_indent_margin* parameter is an integer.

   After you complete this step, the argument definition section of your UIL specification file should resemble the following example:

```
value
xyz_font_level_0 :       argument ('fontLevel0',         font);
xyz_font_level_1 :       argument ('fontLevel1',         font);
xyz_font_level_2 :       argument ('fontLevel2',         font);
xyz_font_level_3 :       argument ('fontLevel3',         font);
xyz_font_level_4 :       argument ('fontLevel4',         font);
xyz_indent_margin :      argument ('indentMargin',       integer);
xyz_unit_level :         argument ('unitLevel',          integer);
xyz_page_level :         argument ('pageLevel',          integer);
xyz_root_widget :        argument ('rootWidget',         integer);
xyz_root_entry :         argument ('rootEntry',          integer);
xyz_display_mode :       argument ('displayMode',        integer);
xyz_fixed_width_entries: argument ('fixedWidthEntries', Boolean);
```

2. Identify and define In the UIL Module all callback reasons for the user–defined widget that are not already built into the UIL Compiler. The strings you pass to the REASON subroutine must match the names listed in the resource list structure in the widget class record for the XYZ widget. (Callback reasons, like UIL parameters, are considered to be widget–specific attributes in the AIXwindows Toolkit and are defined as resources.)

   When you complete this step, the reason definition section of your UIL specification file should resemble the following example:

```
value
xyz_select_and_confirm : reason ('selectAndConfirmCallback');
xyz_extend_confirm :     reason ('extendConfirmCallback');
xyz_entry_selected :     reason ('entrySelectedCallback');
xyz_entry_unselected :   reason ('entryUnselectedCallback');
xyz_help_requested:      reason ('helpCallback');
xyz_attach_to_source :   reason ('attachToSourceCallback');
xyz_detach_from_source : reason ('detachFromSourceCallback');
xyz_alter_root :         reason ('alterRootCallback');
xyz_selections_dragged : reason ('selectionsDraggedCallback');
xyz_get_entry :          reason ('getEntryCallback');
xyz_dragging :           reason ('draggingCallback');
xyz_dragging_end :       reason ('draggingEndCallback');
xyz_dragging_cancel :    reason ('draggingCancelCallback');

value
XyzPositionTop :     1;
XyzPositionMiddle : 2;
XyzPositionBottom : 3;
XyzDisplayOutline : 1;
XyzDisplayTopTree : 2;
```

The following comments apply to the preceding sample code:

General Comment:	**Step 1** and **step 2** can be accomplished in one or more sections as shown, or in line when declaring an instance of the user–defined widget.
Code Segment #1:	These 5 lines define integer literals for specifying parameters of the XYZ widget.

3. Declare the creation subroutine for the user–defined widget using the following syntax:

   ```
   procedure XyzCreate();
   ```

4. Declare an instance of the user–defined widget.
5. Specify the object type as user_defined and include the name of the widget creation subroutine in the declaration.

   The following example shows how to specify the XYZ widget in a UIL Module.

   Note: This UIL Module includes the UIL Specification File named **xyz_widget.uil** whose definition sections were shown in **step 1** and **step 2**.

```
    Module xyz_example
    names = case_sensitive
    include file 'XmAppl.uil';
(1)
    include file 'xyz_widget.uil';

(2)
    procedureXyzAttach ();
```

```
    XyzDetach          ();
    XyzExtended        ();
    XyzConfirmed       ();
    XyzGetEntry        ();
    XyzSelected        ();
    XyzUnselected      ();
    XyzDragged         ();
    XyzDragging        ();
    XyzDraggingEnd     ();
    create_proc        ();
    MenuQuit           ();
    MenuExpandAll      ();
    MenuCollapseAll    ();
(3)
    object main : XmMainWindow {
        arguments {
        XmNx = 0; XmNy = 0; XmNheight = 0; XmNwidth = 0;
                  };
        controls { XmMenuBar main_menu; user_defined xyz_widget;
        };
    };
(4)
    xyz_widget : user_defined
    procedure XyzCreate {
        arguments {
        XmNx = 0; XmNy = 0; XmNheight = 600; XmNwidth = 400;
(5)
        xyz_display_mode = XyzDisplayOutline;
                  };
    callbacks {
        xyz_attach_to_source   = procedure XyzAttach();
        xyz_detach_from_source = procedure XyzDetach();
        xyz_get_entry          = procedure XyzGetEntry();
        xyz_select_and_confirm = procedure XyzConfirmed();
        xyz_extend_confirm     = procedure XyzExtended();
        xyz_entry_selected     = procedure XyzSelected();
        xyz_entry_unselected   = procedure XyzUnselected();
        xyz_selections_dragged = procedure XyzDragged();
        xyz_dragging           = procedure XyzDragging();
        xyz_dragging_end       = procedure XyzDraggingEnd();
(6)
        MrmcreateCallback      = procedure create_proc();
              };
    };

(7)
   main_menu: XmMenuBar {
   arguments { XmNorientation = XmHORIZONTAL;
            };
   controls { XmCascadeButton file_menu; };
            };
   file_menu: XmCascadeButton { arguments { XmNlabelString = 'File';
       };
   controls { XmPulldownMenu { controls { XmPushButton
              expand_all_button;
   XmPushButton collapse_all_button;
   XmPushButton quit_button;
           };
        };
      };
   };
   expand_all_button: XmPushButton {
       arguments { XmNlabelString = "Expand All";
             };
        callbacks { XmNactivateCallback = procedure MenuExpandAll();
              };
        };
    collapse_all_button: XmPushButton {
        arguments { XmNlabelString = "Collapse All";
        };
        callbacks {
    XmNactivateCallback = procedure MenuCollapseAll();
              };
    };
    quit_button: XmPushButton {
        arguments { XmNlabelString = "Quit";
              };
        callbacks { XmNactivateCallback = procedure MenuQuit();
              };
    };
    end Module;
```

6. In the application program, register the class of the user–defined widget using the MRM **MrmRegisterClass** subroutine. Part of the information you provide to the **MrmRegisterClass** subroutine is the name of the widget creation subroutine. By registering the class (and creation subroutine), you allow MRM to create a user–defined widget using the same mechanisms that create AIXwindows Toolkit objects. You can specify the widget using UIL and fetch the widget with MRM.

   The following application program displays the XYZ widget defined, declared, and created in **steps 1–4**:

```
  #include <MRM/MrmAppl.h>
(1)
  #include <XmWsXyz.h>
(2)
  globalref int xyzwidgetclassrec;
(3)
  extern void XyzAttach       ();

      extern void XyzDetach       ();
      extern void XyzGetEntry     ();
      extern void XyzConfirmed    ();
      extern void XyzExtended     ();
      extern void XyzSelected     ();
      extern void XyzUnselected   ();
      extern void XyzHelpRoutine  ();
      extern void XyzDragged      ();
      extern void XyzDragging     ();
      extern void XyzDraggingEnd  ();
      extern void create_proc     ();
      extern void MenuQuit        ();
      extern void MenuExpandAll   ();
      extern void MenuCollapseAll ();

(4)
      static MrmRegisterArglist register_vector[] =
      {
        {
        { "XyzAttach",        (caddr_t) XyzAttach        },
        { "XyzDetach",        (caddr_t) XyzDetach        },
        { "XyzGetEntry",      (caddr_t) XyzGetEntry      },
        { "XyzConfirmed",     (caddr_t) XyzConfirmed     },
        { "XyzExtended",      (caddr_t) XyzExtended      },
        { "XyzSelected",      (caddr_t) XyzSelected      },
        { "XyzUnselected",    (caddr_t) XyzUnselected    },
        { "XyzHelpRoutine",   (caddr_t) XyzHelpRoutine   },
        { "XyzDragged",       (caddr_t) XyzDragged       },
        { "XyzDragging",      (caddr_t) XyzDragging      },
        { "XyzDraggingEnd",   (caddr_t) XyzDraggingEnd   },
        { "create_proc,       (caddr_t) create_proc      },
        { "MenuQuit",         (caddr_t) MenuQuit         },
        { "MenuExpandAll",    (caddr_t) MenuExpandAll    },
        { "MenuCollapseAll",  (caddr_t) MenuCollapseAll }
    };
    #define register_vector_length ( (sizeof register_vector) / \
    (sizeof register_vector[0]))
(5)
        static MrmHierarchy hierarchy_id ;
        static char         *vec[]={"xyz_example.uid"};
        static MrmCode      class;

            Widget toplevel;
            Widget mainwindow;
(6)
        int main (argc, argv)
        unsigned int argc;
        char **argv;
    {
(7)
        Arg arguments[1];
(8)
        MrmInitialize();
(9)
    if( MrmRegisterClass
        ( MRMwcUnknown,
          XyzClassName,
          "XyzCreate",
          XyzCreate,
          &xyzwidgetclassrec )
    != MrmSUCCESS )
  {
    printf ("Can't register XYZ widget");
  }
(10)
    toplevel = XtInitialize ("xyz", "xyz", NULL, 0, &argc, argv);
(11)
```

```
        if( MrmOpenHierarchy
            (1,
             vec,
             NULL,
             &hierarchy_id )
        != MrmSUCCESS )
    {
        printf ("Can't open hierarchy");
    }
(12)
        MrmRegisterNames( register_vector, register_vector_length );
        XtSetArg (arguments[0], XmNallowShellResize, TRUE);
        XtSetValues (toplevel, arguments, 1);
(13)
        if( MrmFetchWidget
            ( hierarchy_id,
              "main",
              toplevel,
              &mainwindow,
              &class )
        != MrmSUCCESS )
    {
        printf ("Can't fetch interface");
    }
        XtManageChild (mainwindow);
        XtRealizeWidget (toplevel);
        XtMainLoop();
        return (0);
    }
```

The following comments apply to the preceding sample code:

Code Segment #1:	Including `XYZ declarations.`
Code Segment #2:	Providing a reference to the widget class record for the XYZ widget (named xyzwidgetclassrec).
Code Segment #3:	Declaring callback subroutines defined (but not shown) later in the program.
Code Segment #4:	Defining the mapping between UIL procedure names and their addresses.
Code Segment #5:	Specifying the UID hierarchy list. The UID hierarchy for this application consists of a single UID file, the compiled version of **xyz_example.uil**. (Assume the UIL specification file has the same name as the UIL Module; see the module header. The file named **xyz_example.uil** includes the file **xyz_widget.uil**.
Code Segment #6:	Main subroutine.
Code Segment #7:	Parameters for the widgets.
Code Segment #8:	Initializing MRM.
Code Segment #9:	Registering the XYZ widget class with MRM. This allows MRM to use standard creation mechanisms to create the XYZ widget. The parameters passed to the **MrmRegisterClass** subroutine are as follows:

- MRMwcUnknown—Indicates user–defined class
- XyzClassName—Class name of XYZ widget, defined in **XmWsXyz.h** header file
- "XyzCreate"—Name of the creation subroutine
- *XyzCreate*—Address of the creation subroutine
- xyzwidgetclassrec—Pointer to the widget class record

Code Segment #10:	Initializing the AIX Toolkit.
Code Segment #11:	Defining the UID hierarchy.
Code Segment #12:	Registering callback subroutine names with MRM.
Code Segment #13:	The XYZ user–defined widget is treated like every other AIXwindows Toolkit object. MRM calls the XYZ widget creation subroutine (**XyzCreate**) and passes this subroutine the values for the **XmNx**, **XmNy**, **XmNwidth**, **XmNheight**, and *xyz_display_mode* parameters as specified in the UID file, using the standard creation subroutine format.

Suggested Reading

Related Information

How to Create a User Interface at Run Time Using the AIXwindows Resource Manager (MRM) contains detailed information about the AIXwindows Resource Manager.

How to Create a User Interface at Run Time Using the AIXwindows Resource Manager (MRM) contains detailed information about the AIXwindows Resource Manager.

Chapter 6. Enhanced X–Windows

Enhanced X-Windows Programming Introduction

Enhanced X-Windows is a network-transparent windowing system that runs under the AIX Operating System. Enhanced X-Windows runs on systems with bitmapped display terminals. The X Server distributes user input to and accepts output requests from various client programs located either on the same system or elsewhere in a network. The **Xlib** library is a **C** language subroutine library that client programs use to interface with the windowing system.

The application program you create is the client part of a client-server relationship; you write the program and the X Server provides it independence from the hardware. There is one X Server for each virtual terminal that runs Enhanced X-Windows. The term display refers to a logical virtual terminal with its associated keyboard, locator, and server unless it is explicitly stated otherwise.

In Enhanced X-Windows, a window is a rectangular area on the screen that lets you view 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.

Enhanced X-Windows supports one or more screens containing overlapping windows or subwindows. A screen is a physical monitor, which can be color or black and white, and hardware.

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 display screen. The root window is partially or completely covered by child windows. All windows, except the root window, have parent windows. There is usually at least one window per application program. Child windows can, in turn, have their own child window. In this way, an application program can create a tree of arbitrary depth on the screen.

A child window may be larger than its parent. Part or all of the child window may extend beyond the boundaries of the parent. However, all output to a window is clipped by the boundaries of its parent window. 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 suppressed by the window system. If a window is obscured by a second window, the window obscures only those ancestors of the second window, which are also ancestors of the first window.

A window has a border of zero or more pixels in *Width*, which can be any pattern or solid color. A window usually, but not always, has a background pattern that is repainted by the window system when the window is uncovered. Each window has its own coordinate system. Child windows obscure the parent window unless the child windows have no background. Graphic operations in the parent window are usually clipped by the child windows.

Input from Enhanced X-Windows takes the form of events. Events may be side effects of a command (for example, restacking windows generates exposure events) or completely asynchronous (for example, the keyboard). A client program asks to be informed of events. Enhanced X-Windows never sends events that a program did not ask for.

Enhanced X-Windows 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, and the client program is notified by an exposure event that part or all of the window needs to be repainted. Programs must be prepared to regenerate the contents of windows on demand.

Enhanced X-Windows also provides off-screen storage of graphics objects, called pixmaps. Single plane (depth 1) pixmaps are sometimes referred to as bitmaps. Bitmaps can be used in most graphics subroutines 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.

Most of the subroutines in **Xlib** only add requests to an output buffer. These requests execute asynchronously later on the X Server, the display server. subroutines that return values of information stored in the server do not return; these subroutines block until an explicit reply is received or an error occurs. If a nonblocking call results in an error, the error is generally not reported by a call to an optional error handler until some later blocking call is made.

If Enhanced X-Windows does not want a request to execute asynchronously, a client can follow the request with a call to the **XSync** subroutine, which blocks until all previously buffered asynchronous events have been sent and acted upon. The output buffer is always flushed by a call to any subroutine that returns a value or waits for input (for example, the **XPending**, **XNextEvent**, **XWindowEvent**, **XFlush**, or **XSynchronize** subroutines).

Many **Xlib** subroutines 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 <> has the meaning defined by the **#include** statement of the C compiler and is a file relative to a well known directory. On the AIX Operating System this is **/usr/include**.)

Some subroutines return *Status* which is an integer error indication. If the subroutine fails, *Status* will be 0. If the subroutine returns a status of 0, the subroutine did not update the return parameters. Because C language does not provide multiple return values, many subroutines must return their results by writing into client-passed storage. Any 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 subroutines are used for reading only. By default, errors are handled either by a standard library subroutine or by a library subroutine that you provide. subroutines that return pointers to strings return NULL pointers if the string does not exist.

Input events, for example, key pressed events or mouse moved events, arrive asynchronously from the server and are queued until they are requested by a call to the **XNextEvent** or **XWindowEvent** subroutines. In addition, some of the library subroutines, such as the **XResizeWindow** and **XRaiseWindow** subroutines, 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 may wish to wait for them explicitly by calling the **XSync** subroutine after calling a subroutine that may generate exposure events.

Compiling X Programs with Enhanced X-Windows

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

cc {*Compiler Options*} **-o***Sample Sample*.**c** **libX11.a**

In this example, *Sample*.**c** is the name of your C language source program, *Sample* is the name of your executable C program and **libX11.a** is the Enhanced X-Windows subroutine library (**/usr/lib/libX11.a**).

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 Enhanced X-Windows subroutines. To do so, insert the following statement early in your program:

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

Related Information

The **<X11/X.h>** header file, **/usr/include/X11/Xlib.h** file, **/usr/include/X11/X.h** file.

The **cc** Command.

The **XNextEvent** subroutine, **XRaiseWindow** subroutine, **XResizeWindow** subroutine, **XSync** subroutine, **XWindowEvent** subroutine.

List of Xlib Tasks

The following is a list of the Enhanced X-Windows subroutines grouped according to the type of function provided.

Opening and Closing Displays

XOpenDisplay	Open a display.
XNoOp	Execute a NoOperation protocol request.
XFree	Free in memory data created by an **Xlib** library subroutine.
XCloseDisplay	Close a display.

Creating and Destroying Windows

XCreateWindow	Create unmapped subwindow.
XCreateSimpleWindow	Create unmapped InputOutput subwindow.
XDestroyWindow	Unmap and destroy window and all subwindows.
XDestroySubwindows	Destroy all subwindows of specified window.

Manipulating Windows

XMapWindow	Map the specified window.
XMapRaised	Map and raise the specified window.
XMapSubwindows	Map all subwindows of the specified window.
XUnmapWindow	Unmap the specified window.
XUnmapSubwindows	Unmap all subwindows of the specified window.
XConfigureWindow	Configure the specified window.
XMoveWindow	Move the specified window.
XResizeWindow	Change the specified window's size.
XMoveResizeWindow	Change the specified window's size and location.
XSetWindowBorderWidth	Change the border width of the window.
XRaiseWindow	Raise the specified window.
XLowerWindow	Lower the specified window.
XCirculateSubwindows	Circulate a subwindow up or down.
XCirculateSubwindowsUp	Raise the lowest mapped child of window.
XCirculateSubwindowsDown	Lower the highest mapped child of window.
XRestackWindows	Restack a set of windows from top to bottom.

Changing Window Attributes

XChangeWindowAttributes	Change one or more window attributes.
XSetWindowBackground	Set a window background to a specified pixel.
XSetWindowBackgroundPixmap	Set a window background to a specified pixmap.
XSetWindowBorder	Change window's border to specified pixel.
XSetWindowBorderPixmap	Change window's border tile.
XTranslateCoordinates	Transform coordinates betweens windows.

Obtaining Window Information

XQueryTree	Obtain the IDs of the children and the parent windows.
XGetWindowAttributes	Get current attributes for specified window.
XGetGeometry	Get current geometry of specified drawable.
XQueryPointer	Get pointer coordinates and root window.

Properties and Atoms

XGetAtomName	Get a name for the specified atom ID.
XInternAtom	Get an atom for the specified name.

Manipulating Window Properties

XChangeProperty	Change the property for specified window.
XDeleteProperty	Delete a property for the specified window.
XGetWindowProperty	Get atom type and property format for window.
XListProperties	Get the specified window's property list.
XRotateWindowProperties	Rotate properties in property array.

Setting Window Selections

XConvertSelection	Convert a selection.
XGetSelectionOwner	Get the selection owner.
XSetSelectionOwner	Set the selection owner.

Manipulating Colormaps

XCopyColormapAndFree	Create a new colormap from specified colormap.
XCreateColormap	Create a colormap.
XFreeColormap	Free the specified colormap.
XQueryColor	Query the RGB value for a specified pixel.
XQueryColors	Query the RGB values for array of pixels.
XSetWindowColormap	Set the colormap of the specified window.

Manipulating Color Cells

XAllocColor	Allocate a read only color cell.
XAllocColorCells	Allocate read-write color cells.
XAllocColorPlanes	Allocate read/write color resources.
XAllocNamedColor	Allocate a read only color cell by name.
XFreeColors	Free colormap cells.
XLookupColor	Look up colorname.
XStoreColor	Store an RGB value into a single colormap cell.
XStoreColors	Store RGB values into colormap cells.
XStoreNamedColor	Set a pixel color to the named color.

Creating and Freeing Pixmaps

XCreatePixmap	Create a pixmap of a specified size.
XFreePixmap	Free all storage associated with specified pixmap.

Manipulating Graphics Contexts (GCs)

XChangeGC	Change the components in the specified GC.
XCreateGC	Create a new GC.
XCopyGC	Copy components from a source GC to a destination GC.
XFreeGC	Free the specified GC.
XGContextFromGC	Obtain the GContext resource ID for the GC.
XQueryBestSize	Get the best size of tile, stipple, or cursor.
XQueryBestStipple	Get the best stipple shape.
XQueryBestTile	Get the best fill tile shape.
XSetArcMode	Set the *arc_mode* field of the specified GC.

XSetBackground Set the backgound of the specified GC.
XSetClipMask Set the *clip_mask* field of the specified GC to a specified pixmap.
XSetClipOrigin Set the *clip_origin* field of the specified GC.
XSetClipRectangles Set the *clip_mask* field of the specified GC to a list of rectangles.
XSetDashes Set the dashed line style components of the specified GC.
XSetFillRule Set the fill rule of the specified GC.
XSetFillStyle Set the fill style of the specified GC.
XSetFont Set the current font of the specified GC.
XSetForeground Set the foregound of the specified GC.
XSetFunction Set display function in specified GC.
XSetGraphicsExposures Set the *graphics_exposure* field of the specified GC.
XSetLineAttributes Set the line drawing components of the GC.
XSetPlaneMask Set the plane mask of the specified GC.
XSetState Set foreground, background, plane mask and function in GC.
XSetStipple Set the stipple of the specified GC.
XSetSubwindowMode Set the subwindow mode of the specified GC.
XSetTSOrigin Set the tile or stipple origin of the specified GC.
XSetTile Set the fill tile of the specified GC.

Clearing and Copying Areas

XClearArea Clear a rectangular area of window.
XClearWindow Clear the entire window.
XCopyArea Copy drawable area between drawables of the same root and the same depth.
XCopyPlane Copy single bit plane of drawable.

Drawing Lines

XDrawArc Draw single arc in drawable.
XDrawArcs Draw multiple arcs in specified drawable.
XDrawLine Draw a single line between two points in drawable.
XDrawLines Draw multiple lines in the specified drawable.
XDrawPoint Draw a single point in specified drawable.
XDrawPoints Draw multiple points in specified drawable.
XDrawRectangle Draw outline of single rectangle in drawable.
XDrawRectangles Draw outline of multiple rectangles in drawable.
XDrawSegments Draw multiple line segments in specified drawable.

Filling Areas

XFillArc Fill single arc in drawable.
XFillArcs Fill multiple arcs in drawable.
XFillPolygon Fill a polygon area in drawable.
XFillRectangle Fill single rectangular area in drawable.
XFillRectangles Fill multiple rectangular areas in drawable.

Loading and Freeing Fonts

XFreeFont Unload font and free storage used by font.
XFreeFontInfo Free the font information array.
XFreeFontNames Free a font name array.
XFreeFontPath Free data returned by the **XGetFontPath** subroutine.
XGetFontPath Get the current font search path.
XGetFontProperty Get the specified font property.
XListFonts Get a list of available font names.
XListFontsWithInfo Get names and information about loaded fonts.
XLoadFont Load a font.

XLoadQueryFont Loads and queries font in one operation.
XQueryFont Get information about a loaded font.
XSetFontPath Set the font search path.
XUnloadFont Unload the specified font.

Querying Character String Sizes

XQueryTextExtents Get 1 byte character string bounding box from server.
XQueryTextExtents16 Get 2 byte character string bounding box from server.
XTextExtents Get bounding box of 1 byte character string.
XTextExtents16 Get bounding box of 2 byte character string.
XTextWidth Get the width of an 8 bit character string.
XTextWidth16 Get the width of a 2 byte character string.

Drawing Text

XDrawImageString Draw 8 bit image text in specified drawable.
XDrawImageString16 Draw 2 byte image text in specified drawable.
XDrawString Draw 8 bit text in specified drawable.
XDrawString16 Draw 2 byte text in specified drawable.
XDrawText Draw 8 bit complex text in specified drawable.
XDrawText16 Draw 2 byte complex text in specified drawable.

Transferring Images

XGetImage Get image from rectangle in drawable.
XGetSubImage Copy rectangle on display to image.
XPutImage Put image from memory into rectangle in drawable.

Manipulating Cursors

XCreateFontCursor Create a cursor from a standard font
XCreateGlyphCursor Create a cursor from font glyphs.
XDefineCursor Define a cursor for a window.
XFreeCursor Free a cursor.
XQueryBestCursor Get useful cursor sizes.
XRecolorCursor Change the color of a cursor.
XUndefineCursor Undefine a cursor for a window.

Handling Window Manager Functions

XAddToSaveSet Add a window to the client's save-set.
XAllowEvents Allow events to be processed after a device is frozen.
XChangeActivePointerGrab Change the active pointer grab.
XChangePointerControl Change the interactive feel of pointer device.
XChangeSaveSet Add or remove a window from the client's save-set.
XGetInputFocus Get the current input focus.
XGetPointerControl Get the current pointer parameters.
XGrabButton Grab a mouse button.
XGrabKey Grab a single key of the keyboard.
XGrabKeyboard Grab the keyboard.
XGrabPointer Grab the pointer.
XGrabServer Grab the server.
XInstallColormap Install a colormap.
XKillClient Remove a client.
XListInstalledColormaps Get a list of currently installed colormaps.
XRemoveFromSaveSet Remove a window from the client's save-set.
XReparentWindow Change the parent of a window.
XSetCloseDownMode Change the close down mode of a client.

XSetInputFocus Set the input focus.
XUngrabButton Ungrab a mouse button.
XUngrabKey Ungrab a key.
XUngrabKeyboard Ungrab the keyboard.
XUngrabPointer Ungrab the pointer.
XUngrabServer Ungrab the server.
XUninstallColormap Uninstall a colormap.
XWarpPointer Move the pointer to arbitrary point on the screen.

Manipulating Keyboard Settings

XAutoRepeatOff Turn off keyboard auto-repeat.
XAutoRepeatOn Turn on keyboard auto-repeat.
XBell Set the volume of the bell.
XChangeKeyboardControl Change keyboard settings.
XChangeKeyboardMapping Change the mapping of symbols to keycodes.
XDeleteModifiermapEntry Delete an entry for **XModifierKeymap** structure.
XDisplayKeycodes Return minimum and maximum number of keycodes supported by the specified display.
XFreeModifiermap Free **XModifierKeymap** structure.
XGetKeyboardControl Get the current keyboard settings.
XGetKeyboardMapping Get the mapping of symbols to keycodes.
XGetModifierMapping Get keycodes to be modifiers.
XGetPointerMapping Get the mapping of buttons on the pointer.
XInsertModifiermapEntry Add an entry from **XModifierKeymap** structure.
XNewModifiermap Create the **XModifierKeymap** structure.
XQueryKeymap Get the state of the keyboard keys.
XSetModifierMapping Set keycodes to be modifiers.
XSetPointerMapping Set the mapping of buttons on the pointer.

Controlling the Screen Saver

XActivateScreenSaver Activate the screen saver.
XForceScreenSaver Turn the screen saver on or off.
XGetScreenSaver Get the current screen saver settings.
XSetScreenSaver Set the screen saver.

Manipulating Hosts and Access Control

XAddHost Add a host.
XAddHosts Add multiple hosts.
XDisableAccessControl Disable access control.
XEnableAccessControl Enable access control.
XListHosts Get the list of hosts.
XRemoveHost Remove a host.
XRemoveHosts Remove multiple hosts.
XSetAccessControl Change access control.

Handling Events

XCheckIfEvent Check event queue for specified event without blocking.
XCheckMaskEvent Remove the next event that matches a specified mask without blocking.
XCheckTypedEvent Get the next event that matches event type.
XCheckTypedWindowEvent Get the next event for specified window.
XCheckWindowEvent Remove next event that matches the specified window and mask without blocking.
XDisplayMotionBufferSize Return the size of the motion buffer.

XEventsQueued	Check the number of events in the event queue.
XFlush	Flush the output buffer.
XGetMotionEvents	Get the motion history for specified window.
XIfEvent	Check event queue for specified event and remove it.
XMaskEvent	Remove the next event that matches a specified mask.
XNextEvent	Get the next event and remove it from the queue.
XPeekEvent	Peek at the event queue.
XPeekIfEvent	Check event queue for specified event.
XPending	Return the number of events that are pending.
XPutBackEvent	Push event back to top of event queue.
XSelectInput	Select events to be reported to the client.
XSendEvent	Send an event to a specified window.
XSync	Flush the output buffer and wait until all requests are completed.
XWindowEvent	Remove next event that matches the specified window and mask.

Enabling or Disabling Synchroniztion

XSetAfterFunction	Set the previous after function.
XSynchronize	Enable or disable synchronization.

Using Default Error Handling

XDisplayName	Get name of display currently being used.
XGetErrorDatabaseText	Get error text from the error database.
XGetErrorText	Get error text for specified error code.
XSetErrorHandler	Set error handler.
XSetIOErrorHandler	Set error handler for fatal I/O errors.

Communicating with Window Managers

XFetchName	Get the name of a window.
XGetClassHint	Get the class of a window.
XGetIconName	Get the name of an icon window.
XGetIconSizes	Get the values of icon size atom.
XGetNormalHints	Get size hints for window in normal state.
XGetSizeHints	Get the values of type **WM_SIZE_HINTS** properties.
XGetStandardColormap	Get colormap associated with specified atom.
XGetTransientForHint	Get **WM_TRANSIENT_FOR** property for window.
XGetWMHints	Get the value of the window manager's hints atom
XGetZoomHints	Get values of the zoom hints atom.
XSetClassHint	Set the class of a window.
XSetCommand	Set the value of the command atom.
XSetIconName	Assign a name to an icon window.
XSetIconSizes	Set the values of icon size atom.
XSetNormalHints	Set size hints for window in normal state.
XSetSizeHints	Set the values of type**WM_SIZE_HINTS** properties.
XSetStandardColormap	Set colormap associated with specified atom.
XSetStandardProperties	Specify a minumum set of properties.
XSetTransientForHint	Set **WM_TRANSIENT_FOR** property for window.
XSetWMHints	Set the value of the window manager's hints atom.
XSetZoomHints	Set values of the zoom hints atom.
XStoreName	Assign a name to a window.

Keyboard Event Functions

XKeycodeToKeysym	Convert keycode to a keysym value.
XKeysymToKeycode	Convert keysym value to keycode.
XKeysymToString	Convert keysym value to keysym name.

XLookupKeysym Translate keyboard event into keysym value.
XLookupMapping Get mapping of keyboard event from keymap file.
XLookupString Translate keyboard event into character string.
XRebindCode Change the keyboard mapping in keymap file.
XRebindKeysym Map character string to specified keysym and modifiers.
XRefreshKeyboardMapping Refresh stored modifier and keymap information.
XStringToKeysym Convert keysym name to keysym value.
XUseKeymap Change keymap files.

Handling Default Geometry Color

XGeometry Parse the window geometry, given the padding and font values.
XGetDefault Get default window options.
XParseColor Obtain the RGB values from the color name.
XParseGeometry Parse standard window geometry options.
XResourceManagerString Return the RESOURCE_MANAGER property from the screen of the root window of screen 0.

Manipulating Regions

XClipBox Generate the smallest enclosing rectangle in region.
XCreateRegion Create a new empty region.
XDestroyRegion Free storage associated with specified region.
XEmptyRegion Determine if specified region is empty.
XEqualRegion Determine if two regions are the same.
XIntersectRegion Compute intersection of two regions.
XOffsetRegion Move specified region by specified amount.
XPointInRegion Determine if point lies in specified region.
XPolygonRegion Generate a region from points.
XRectInRegion Determine if rectangle lies in specified region.
XSetRegion Set the GC to the specified region.
XShrinkRegion Reduce specified region by specified amount.
XSubtractRegion Subtract two regions.
XUnionRectWithRegion Create a union of source region and rectangle.
XUnionRegion Compute union of two regions.
XXorRegion Get difference between union and intersection of regions.

Using Cut and Paste Buffers

XFetchBuffer Get data from specified cut buffer.
XFetchBytes Get data from first cut buffer.
XRotateBuffers Rotate the cut buffers.
XStoreBuffer Store data in specified cut buffer.
XStoreBytes Store data in first cut buffer.

Querying Visual Types

XGetVisualInfo Get list of visual information structures.
XMatchVisualInfo Get visual information matching screen depth and class.
XVisualIDFromVisual Return the visual ID for the specified visual type.

Manipulating Images

XAddPixel Increment each pixel in pixmap by constant value.
XCreateImage Allocate memory for **XImage** structure.
XDestroyImage Free memory for **XImage** structure.
XGetPixel Get a pixel value in an image.
XPutPixel Set a pixel value in an image.
XSubImage Create image that is subsection of specified image.

Manipulating Bitmaps

XCreateBitmapFromData Include bitmap in **C** program.
XCreatePixmapFromBitmapData Create pixmap using bitmap data.
XReadBitmapFile Read in a bitmap from a file.
XWriteBitmapFile Write out a bitmap to a file.

Using the Resource Manager

Xpermalloc Allocate memory which is never freed.
XrmGetFileDatabase Create a database from specified file.
XrmGetResource Retrieve a resource from a database.
XrmGetStringDatabase Create a database from specified string.
XrmInitialize Initialize the resource manager.
XrmMergeDatabases Merge two databases.
XrmParseCommand Store command options into a database.
XrmPutFileDatabase Copy database into specified file.
XrmPutLineResource Store single resource entry into database.
XrmPutResource Store resource into database.
XrmPutStringResource Store string resource into database.
XrmQGetResource Retrieve a quark from a database.
XrmQGetSearchList Get a resource search list of database levels.
XrmQGetSearchResource Get a quark search list of database levels.
XrmQPutResource Store binding and quarks into database.
XrmQPutStringResource Store string binding and quarks into database.
XrmQuarkToString Convert a quark to a character string.
XrmStringToBindingQuarkList Convert strings to bindings and quarks.
XrmStringToQuark Convert character string to a quark.
XrmStringToQuarkList Convert character strings to quark list.
XrmUniqueQuark Allocate a new quark.

Using the Context Manager

XDeleteContext Delete data associated with window and context type.
XFindContext Get data associated with window and context type.
XSaveContext Store data associated with window and context type.
XUniqueContext Allocate a new context.

Enhanced X-Windows Display Subroutines Overview

Before a program can use a display, the program must establish a connection to the X Server driving the display. Once you have established a connection, you can then use the Xlib macros and functions to return information about this display such as image format size and depth.

Opening an Enhanced X-Windows Display

To open a connection to the X Server controlling a specified display, use the **XOpenDisplay** subroutine. This subroutine 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 **TCP** or **UNIX** domain sockets. If the *Hostname* is a host system name and a colon (:) separates the host name and display number, the **XOpenDisplay** subroutine connects the host and the display using **TCP** sockets. If the *Hostname* does not exist or is **unix** and a colon (:) separates it from the display number, the **XOpenDisplay** subroutine 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 that has the format:

```
Hostname:Number.Screen
```

Hostname	Specifies the name of the host system where the display is physically attached. The *Hostname* 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** subroutine.

For example, the following would 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 some subroutines or display macros.

Closing an Enhanced X-Windows Display

When the X Server connection to a client is closed, either by an explicit call to the **XCloseDisplay** subroutine or by a process that exits, the X Server performs the following operations automatically:

- Disowns all selections owned by the client.
- Performs the **XUngrabPointer** and **XUngrabKeyboard** subroutines if the client application has actively grabbed the pointer or the keyboard.
- Performs the **XUngrabServer** subroutine if the client has grabbed the server.
- Releases all passive grabs made by the client application.
- Marks all resources (including colormap entries) allocated by the client application as permanent or temporary, depending on whether the close-down mode is the value of **RetainPermanent** or **RetainTemporary**. However, this does not prevent other client applications from explicitly destroying the resources.

When the X Server connection to a client is closed in the **DestroyAll** mode, the X Server destroys all of the resources of the client application as follows:

- Examines each window in the client save set to determine if the save set window is an inferior or subwindow of a window created by the client. (The save set is a list of other client windows that are referred to as save set windows.) If so, the X Server reparents the save set window to the closest ancestor so that the save set window is not an inferior of a window created by the client. The reparenting leaves unchanged the absolute coordinates (with respect to the root window) of the upper-left outer corner of the save-set window.
- Performs a **MapWindow** request on the save set window if the save set window is unmapped. The X Server performs a **MapWindow** request even if the save set window is not an inferior of a window created by the client.
- Examines each window in the client save set and destroys all windows created by the client.

- Performs the appropriate free request on each non-window resource created by the client in the server (for example, **Font, Pixmap, Cursor, Colormap**, and **GContext**).
- Frees all colors and colormap entries allocated by a client application.

Additional processing occurs when the last connection to the X Server closes. An X Server goes through a cycle of having no connections and having some connections. When the last display connection to the X Server closes as a result of a connection closing with the **DestroyAll** close–mode, the X Server:

- Resets its state, as if it had just been started. The X Server begins by destroying all lingering resources from clients that were terminated in **RetainPermanent** or **RetainTemporary** mode.
- Deletes all but the predefined atom identifiers.
- Deletes all properties on all root windows.
- Resets all device maps and attributes (for example, key click, bell volume, and acceleration) and the access control list.
- Restores the standard root tiles and cursors.
- Restores the default font path.
- Restores the input focus to state **PointerRoot**.

However, the X Server does not reset if a connection with a close-mode is set to **RetainPermanent** or **RetainTemporary**.

Related Information

The **DisplayCells** macro, **DisplayHeight** macro, **DisplayHeightMM** macro, **DisplayOfScreen** macro, **DisplayPlanes** macro, **DisplayString** macro, **DisplayWidth** macro, **DisplayWidthMM** macro, **DefaultScreen** macro.

The **XCloseDisplay** subroutine, **XFree** subroutine, **XNoOp** subroutine, **XOpenDisplay** subroutine, **XSetCloseDownMode** subroutine, **XSetSelectionOwner** subroutine, **XUngrabPointer** subroutine, **XUngrabKeyboard** subroutine, **XUngrabServer** subroutine.

Enhanced X-Windows Events and Event-Handling Subroutines Overview

Most applications simply are event loops. That is, they wait for an event, decide what to do with it, execute some amount of code, which, in turn, results in changes to the display, and then wait for the next event. An event is data generated asynchronously by the X Server as a result of some device activity or as side effects of a request sent by an **Xlib** library subroutine.

A client application communicates with the X Server through the connection established with the **XOpenDisplay** subroutine. A client application sends requests to the X Server through this connection. These requests are made by the **Xlib** library subroutines that are called by the client applications. The X Server sends replies or events back to the client application. Most requests made by the **Xlib** library subroutines do not generate replies. Other requests generate multiple replies. Numerous **Xlib** library subroutines cause the X Server to generate events. In addition, typing or moving the pointer can generate events asynchronously.

Using Enhanced X-Windows to Define Event Types

An event type describes a specific event generated by the X Server. For each event type, there is a corresponding constant name defined in the **<X11/X.h>** file. When referring to the event types, use the constant name defined in this file. Grouping one or more event types into logical categories is often useful. For example, exposure processing refers to the processing that occurs for the exposure events **Expose**, **GraphicsExpose**, and **NoExpose**.

Device-related events propagate from the source window to ancestor windows until some client application has selected that event type or until the event is explicitly discarded. The X Server sends an event to a client application only when the client application specifically asked to be informed of that event type, usually by calling the **XSelectInput** subroutine. However, **KeymapNotify** events are always sent.

The following table lists the event categories and their associated event types.

Event Category	**Event Type**
Keyboard events	KeyPress
	KeyRelease
Pointer motion events	ButtonPress
	ButtonRelease
	MotionNotify
Window crossing events	EnterNotify
	LeaveNotify
Input focus events	FocusIn
	FocusOut
Key map state notification event	KeymapNotify
Exposure events	Expose GraphicsExpose NoExpose
Structure control events	CirculateRequest ConfigureRequest MapRequest ResizeRequest
Window state notification events	CirculateNotify ConfigureNotify CreateNotify DestroyNotify GravityNotify MapNotify MappingNotify ReparentNotify UnmapNotify VisibilityNotify
Colormap state notification event	ColormapNotify

Client communication events	ClientMessage PropertyNotify SelectionClear SelectionNotify SelectionRequest
Dial events	DialRotate
Lpfk events	LPFKeyPress
Focus and mapping change events	AIXFocusIn AIXFocusOut AIXDeviceMappingNotify

Using Enhanced X-Windows to Define Event Structures

Each event type has a corresponding structure declared in the **<X11/Xlib.h>** header file. All event structures have the common fields found in the **XAnyEvent** data structure.

The X Server may send events at any time in the input stream, even during the time the client application sends a request and receives a reply. The **Xlib** library stores, in an event queue for later use, any events received while waiting for a reply. It provides several subroutines that check events in the event queue.

In addition to the individual structures declared for each event type, there is also an **XEvent** structure. The **XEvent** structure is a union of the individual structures declared for each event type. After you determine the event type, use the structures declared in the **<X11/Xlib.h>** header file when referring to it in a client application. Depending on the *type* field of the event, you should access elements of each event by using the **XEvent** structure.

Using Enhanced X-Windows to Define Event Masks

The event mask describes the event or events to be returned to the client application by the X Server. Client applications select event reporting of most events relative to a window. An event mask is passed in the *EventMask* parameter to an **Xlib** library event-handling subroutine. The event masks are in the **<X11/X.h>** file.

Most events are not reported to clients when they are generated unless the client has specifically asked for them. However, **GraphicsExpose** and **NoExpose** events are reported by default as a result of **XCopyPlane** and **XCopyArea** subroutines, unless the client suppresses them by setting the *graphics_exposures* field in the **GC** to the value of **False. SelectionClear**, **SelectionRequest**, **SelectionNotify** or **ClientMessage** events cannot be masked, but generally are sent only to clients cooperating with selections. A **MappingNotify** event is always sent to clients when the keyboard mapping is changed.

The following table lists the event mask constants passed in the *EventMask* parameter and the circumstances in which the event mask is used.

Event Mask	Circumstances
NoEventMask	No events wanted
KeyPressMask	Keyboard down events wanted
KeyReleaseMask	Keyboard up events wanted
ButtonPressMask	Pointer button down events wanted
ButtonReleaseMask	Pointer button up events wanted
EnterWindowMask	Pointer window entry events wanted
LeaveWindowMask	Pointer window leave events wanted

PointerMotionMask	Pointer motion events wanted
PointerMotionHintMask	Pointer motion hints wanted
Button1MotionMask	Pointer motion while button 1 down
Button2MotionMask	Pointer motion while button 2 down
Button3MotionMask	Pointer motion while button 3 down
Button4MotionMask	Pointer motion while button 4 down
Button5MotionMask	Pointer motion while button 5 down
ButtonMotionMask	Pointer motion while any button down
KeymapStateMask	Any keyboard state change wanted
ExposureMask	Any exposure wanted
VisibilityChangeMask	Any change in visibility wanted
StructureNotifyMask	Any change in window structure wanted
ResizeRedirectMask	Redirect resize of this window
SubstructureNotifyMask	Substructure notification wanted
SubstructureRedirectMask	Redirect substructure of window
FocusChangeMask	Any change in input focus wanted
PropertyChangeMask	Any change in property wanted
ColormapChangeMask	Any change in Colormap wanted
OwnerGrabButtonMask	Automatic grabs should activate when the *OwnerEvents* parameter of the specific grab subroutine is the value of **True**
DialRotateMask	Dial rotate events wanted
LPFKeyPressMask	Lpfk key-down events wanted
AIXDeviceMapChangeMask	Device state change events wanted
AIXFocusChangeMask	Dial or lpfk input focus events wanted

Using Enhanced X-Windows to Process Events

The event types reported to a client application during event processing depend on which event masks are passed to the *EventMask* parameter of the **XSelectInput** subroutine. Some event masks have a one-to-one correspondence between the event mask constant and the event type constant. For example, if the event mask **ButtonPressMask** is passed, the X Server sends back only **ButtonPress** events. Most events contain a *time* field that indicates the time at which an event occurred.

In other cases, one event mask constant can map to several event type constants. For example, if the event mask **SubstructureNotifyMask** is passed, the X Server can send back **CirculateNotify**, **ConfigureNotify**, **CreateNotify**, **DestroyNotify**, **GravityNotify**, **MapNotify**, **ReparentNotify**, or **UnmapNotify** events.

In another case, two event mask constants map to one event type constant. For example, if the event mask **PointerMotionMask** or **PointerMotionHintMask** is passed, the X Server sends back a **MotionNotify** event.

The following table lists the event mask, the associated event type and the structure name associated with the event type. (NA appears in columns for which the information is not applicable.)

Event Mask	Event Type	Data Structure
ButtonMotionMask	MotionNotify	XPointerMovedEvent
Button1MotionMask	MotionNotify	XPointerMovedEvent
Button2MotionMask	MotionNotify	XPointerMovedEvent
Button3MotionMask	MotionNotify	XPointerMovedEvent
Button4MotionMask	MotionNotify	XPointerMovedEvent
Button5MotionMask	MotionNotify	XPointerMovedEvent
ButtonPressMask	ButtonPress	XButtonPressedEvent
ButtonReleaseMask	ButtonRelease	XButtonReleasedEvent
ColormapChangeMask	ColormapNotify	XColormapEvent
EnterWindowMask	EnterNotify	XEnterWindowEvent
ExposureMask	Expose	XExposeEvent
GCGraphicsExposures in GC	GraphicsExpose	XGraphicsExposeEvent
GCGraphicsExposures in GC	NoExpose	XNoExposeEvent
LeaveWindowMask	LeaveNotify	XLeaveWindowEvent
FocusChangeMask	FocusIn	XFocusInEvent
FocusChangeMask	FocusOut	XFocusOutEvent
KeymapStateMask	KeymapNotify	XKeymapEvent
KeyPressMask	KeyPress	XKeyPressedEvent
KeyPressMask	KeyRelease	XKeyReleasedEvent
OwnerGrabButtonMask	NA	NA
PointerMotionMask	MotionNotify	XPointerMovedEvent
PointerMotionHintMask	NA	NA
PropertyChangeMask	PropertyNotify	XPropertyEvent
ResizeRedirectMask	ResizeRequest	XResizeRequestEvent
StructureNotifyMask	CirculateNotify	XCirculateEvent
StructureNotifyMask	ConfigureNotify	XConfigureEvent
StructureNotifyMask	DestroyNotify	XDestroyWindowEvent
StructureNotifyMask	GravityNotify	XGravityEvent
StructureNotifyMask	MapNotify	XMapEvent
StructureNotifyMask	ReparentNotify	XReparentEvent
StructureNotifyMask	UnmapNotify	XUnmapEvent
SubstructureNotifyMask	CirculateNotify	XCirculateEvent
SubstructureNotifyMask	ConfigureNotify	XConfigureEvent
SubstructureNotifyMask	CreateNotify	XCreateWindowEvent
SubstructureNotifyMask	DestroyNotify	XDestroyWindowEvent
SubstructureNotifyMask	GravityNotify	XGravityEvent

SubstructureNotifyMask	MapNotify	XMapEvent
SubstructureNotifyMask	ReparentNotify	XReparentEvent
SubstructureNotifyMask	UnmapNotify	XUnmapEvent
SubstructureRedirectMask	CirculateRequest	XCirculateRequestEvent
SubstructureRedirectMask	ConfigureRequest	XConfigureRequestEvent
SubstructureRedirectMask	MapRequest	XMapRequestEvent
NA	ClientMessage	XClientMessageEvent
NA	MappingNotify	XMappingEvent
NA	SelectionClear	XSelectionClearEvent
NA	SelectionNotify	XSelectionEvent
NA	SelectionRequest	XSelectionRequestEvent
VisibilityChangeMask	VisibilityNotify	XVisibilityEvent
AIXDeviceMapChangeMask	AIXDeviceMappingNotify	AIXDeviceMappingEvent
AIXFocusChangeMask	AIXFocusIn	AIXFocusInEvent
	AIXFocusOut	AIXFocusOutEvent
DialRotateMask	DialRotate	XDialRotatedEvent
LPFKeyPressMask	LPFKeyPress	XLPFKeyPressedEvent

Using Enhanced X-Windows to Process Specific Pointer Button Events

When a pointer button is pressed with the pointer in a window and no active pointer grab is in progress, the X Server searches the ancestors of the specified window, from the root down, for a passive grab to activate. If there is no matching passive grab on the button, the X Server automatically starts an active grab for the client receiving the event and sets the last-pointer-grab time to the current server time. The effect is equivalent to the **XGrabButton** subroutine with the following client parameters:

window	The event window.
event_mask	The selected pointer-motion events of the client on the event window.
pointer_mode	**GrabModeAsync**.
keyboard_mode	**GrabModeAsync**.
owner_events	The value of **True**, if the client selected **OwnerGrabButtonMask** on the event window. Otherwise, it is the value of **False**.
confine_to	**None**.
cursor	**None**.

The grab is automatically terminated when all buttons are released. Clients can modify the active grab by calling the **XUngrabPointer** and **XChangeActivePointerGrab** subroutines.

Using Enhanced X-Windows to Process Common Keyboard and Pointer Events

The X Server can report **KeyPress** and **KeyRelease** events to clients requesting information about when a key is pressed or released. The X Server generates these events whenever a key changes state, that is, whenever a key is pressed or released. These events are generated for all keys, even keys mapped to modifier bits.

The X Server reports **ButtonPress** and **ButtonRelease** events to clients requesting information about when a pointer button is pressed or released. The X Server generates these events whenever a pointer button changes state, that is, whenever a pointer button is pressed or released.

The X Server reports **MotionNotify** events to clients requesting information about when the pointer moves. The X Server generates this event whenever the pointer changes state, that is, whenever the pointer is moved and the pointer motion begins and ends in the window. The granularity of **MotionNotify** events is not guaranteed, but a client that selects this event type is guaranteed to receive at least one event when the pointer moves and then rests.

To receive the **KeyPress**, **KeyRelease**, **ButtonPress**, and **ButtonRelease** events in a client application, you pass a window ID and **KeyPressMask**, **KeyReleaseMask**, **ButtonPressMask**, and **ButtonReleaseMask** as the *EventMask* parameter to the **XSelectInput** subroutine.

To receive **MotionNotify** events in a client application, pass a window ID and one or more of the following event masks as the *EventMask* parameter to the **XSelectInput** subroutine:

Button1MotionMask-Button5MotionMask	The client application receives **MotionNotify** events only when one or more of the specified buttons is pressed.
ButtonMotionMask	The client application receives **MotionNotify** events only when at least one button is pressed.
PointerMotionMask	The client application receives **MotionNotify** events independent of the state of the pointer buttons.
PointerMotionHintMask	The X Server is free to send only one **MotionNotify** event (with the *is_hint* field of the **XPointerMovedEvent** structure set to **NotifyHint**) to the client for the event window, until either the key or button state changes, or the pointer leaves the event window, or the client calls the **XQueryPointer** or **XGetMotionEvents** subroutines.

The source of the input event is the smallest window containing the pointer. The window used by the X Server to report these events depends on its position in the window hierarchy and whether any intervening window prohibits the generation of these events. The X Server searches up the window hierarchy, starting with the source window until it locates the first window specified by a client as having an interest in these events. If one of the intervening windows has its *do_not_propagate_mask* field set to prohibit generation of the event type, the event of those types will be suppressed. Clients can modify the actual window used for reporting by performing active grabs and, in the case of keyboard events, by using the focus window. The window in which the event is reported is the event window.

Using Enhanced X-Windows for Keyboard Event Subroutines

The **Xlib** library provides subroutines to manipulate keyboard events or to determine information about a key symbol. These subroutines include keyboard event subroutines.

The X Server does not predefine the keyboard to be ASCII characters. Knowing that the `"a"` key was just pressed or possibly that it was just released is often useful. When a key is pressed or released, the X Server sends keyboard events to client programs. The structures associated with keyboard events contain a *keycode* field that assigns a number to each physical key on the keyboard.

Because keycodes are arbitrary and may differ from server to server, client programs dealing with ASCII text, for example, must explicitly convert the keycode value into ASCII. The transformation of keycodes to ASCII or other character sets is arbitrary. Keyboards often differ and writing code that assumes the existence of a particular key on the main keyboard can create portability problems. It can also be difficult to receive **KeyRelease** events on certain X Server implementations because of hardware or software restrictions. Therefore, the **Xlib** library provides subroutines to customize the keyboard layout.

Keyboard events are usually sent to the smallest enclosing window that is interested in that type of event underneath the pointer position. It is also possible to assign the keyboard input focus to a specific window. When the input focus is attached to a window, keyboard events go to the client that selects input on that window rather than to the window under the pointer.

Some implementations cannot support **KeyRelease** events. While some applications exploit **KeyRelease** events to provide superior user interfaces, portability should be considered in designing software that uses **KeyRelease** events or changes key assignments. For example, it is possible to guarantee the existence of the a-z, spacebar, and carriage return keys, but not all keys on all keyboards.

Using Enhanced X-Windows Window Crossing Events

This section describes the processing that occurs for the window crossing events. If a pointer motion or a window hierarchy change causes the pointer to be in a different window than before, the X Server reports **EnterNotify** or **LeaveNotify** events to clients that have selected these events.

EnterNotify and **LeaveNotify** events caused by a hierarchy change are generated after any hierarchy event, **UnmapNotify**, **MapNotify**, **ConfigureNotify**, **GravityNotify**, **CirculateNotify**, caused by that change; but the ordering of **EnterNotify** and **LeaveNotify** events with respect to **FocusOut**, **VisibilityNotify**, and **Expose** events is not constrained by the X protocol.

This type of processing contrasts with **MotionNotify** events, which are also generated when the pointer moves, but the pointer motion begins and ends in a single window. An **EnterNotify** or **LeaveNotify** event can also be generated when a client application calls the **XGrabPointer** and **XUngrabPointer** subroutines.

To receive **EnterNotify** events, pass a window ID and **EnterWindowMask** as the *EventMask* parameter to **XSelectInput** subroutine.

To receive **LeaveNotify** events, pass the window ID and **LeaveWindowMask** as the *EventMask* parameter to the **XSelectInput** subroutine.

Using Enhanced X-Windows to Process Normal Entry or Exit Events

The **EnterNotify** and **LeaveNotify** events are generated when the pointer moves from one window to another window. Normal events are identified by the **XCrossingEvent** data structure, which is the structure for both the **XEnterWindowEvent**, or **XLeaveWindowEvent** data structures with the *mode* field set to **NotifyNormal**.

- When the pointer moves from window A to window B, and window A is an inferior of window B, the X Server generates:
  - A **LeaveNotify** event on window A that has the **XLeaveWindowEvent** data structure with **NotifyAncestor** as the *detail* field.
  - A **LeaveNotify** event on each window between window A and window B exclusive, that has the **XLeaveWindowEvent** data structure with **NotifyVirtual** as the *detail* field.
  - An **EnterNotify** event on window B that has the **XEnterWindowEvent** data structure with **NotifyInferior** as the *detail* field.
- When the pointer moves from window A to window B, and window B is an inferior of window A, the X Server generates:

  - A **LeaveNotify** event on window A that has the **XLeaveWindowEvent** data structure with **NotifyInferior** as the *detail* field.
  - An **EnterNotify** event on each window between window A and window B exclusive, that has the **XEnterWindowEvent** data structure with **NotifyVirtual** as the *detail* field.
  - An **EnterNotify** event on window B that has the **XEnterWindowEvent** data structure with **NotifyAncestor** as the *detail* field.
- When the pointer moves from window A to window B, and window C is their least common ancestor, the X Server generates:
  - A **LeaveNotify** event on window A that has the **XLeaveWindowEvent** data structure with **NotifyNonlinear** as the *detail* field.
  - A **LeaveNotify** event on each window between window A and window C exclusive, that has the **XLeaveWindowEvent** data structure with **NotifyNonlinearVirtual** as the *detail* field.
  - An **EnterNotify** event on each window between window C and window B exclusive, that has the **XEnterWindowEvent** data structure with **NotifyNonlinearVirtual** as the *detail* field.
  - An **EnterNotify** event on window B that has the **XEnterWindowEvent** data structure with **NotifyNonlinear** as the *detail* field.
- When the pointer moves from window A to window B on different screens, the X Server generates:
  - A **LeaveNotify** event on window A that has the **XLeaveWindowEvent** data structure with **NotifyNonlinear** as the *detail* field.
  - If window A is not a root window, the X Server generates a **LeaveNotify** event on each window above window A, up to and including its root window, that has the **XLeaveWindowEvent** data structure with **NotifyNonlinearVirtual** as the *detail* field.
  - If window B is not a root window, the X Server generates an **EnterNotify** event on each window from the root of window B, down to but not including window B, that has the **XEnterWindowEvent** data structure with **NotifyNonlinearVirtual** as the *detail* field.
  - An **EnterNotify** event on window B that has the **XEnterWindowEvent** data structure with the **NotifyNonlinear** as the *detail* field.

Using Enhanced X-Windows to Process Grab and Ungrab Entry or Exit Events

Pseudo-motion mode **EnterNotify** and **LeaveNotify** events are generated when a pointer grab activates or deactivates. Events in which the pointer grab activates are identified by the **XEnterWindowEvent** or **XLeaveWindowEvent** data structures with the *mode* field set to **NotifyGrab**. Events in which the pointer grab deactivates are identified by the **XEnterWindowEvent** or **XLeaveWindowEvent** data structures with the *mode* field set to **NotifyUngrab**.

When a pointer grab activates after any initial warp into a *confine_to* window, and before generating any actual **ButtonPress** event that activates the grab, G is the *grab_window* for the grab, and P is the window the pointer is in, the X Server does the following:

- It generates the **EnterNotify** and **LeaveNotify** events with the *mode* fields of the **XEnterWindowEvent** and **XLeaveWindowEvent** data structures set to **NotifyGrab**. These events are generater as if the pointer warped suddenly from its current position in P (the window the pointer is in) to some position in G (the *grab_window* for the grab).

However, the pointer does not warp, and the X Server uses the pointer position as the initial and final positions for the events.

When a pointer grab deactivates after generating any actual **ButtonRelease** event that deactivates the grab, the G is the *grab_window* for the grab, and P is the window the pointer is in, the X Server does the following:

- It generates the **EnterNotify** and **LeaveNotify** events with the *mode* fields of the **XEnterWindowEvent** and **XLeaveWindowEvent** data structures set to **NotifyUngrab**. These events are generater as if the pointer warped suddenly from its current position in P (the window the pointer is in) to some position in G (the *grab_window* for the grab). However, the pointer does not warp, and the X Server uses the pointer position as the initial and final positions for the events.

Using Enhanced X-Windows to Process Input Focus Events

The X Server reports **FocusIn** or **FocusOut** events to client applications requesting information about changes to **input focus** or **focus window**. The input focus or focus window is the point or window at which the keyboard is attached and input is received. The focus window can be the root window or a top-level window. The focus window and the position of the pointer determines which window receives keyboard input. Clients may need to know when the focus window changes. Input focus or the focus window changes when a client application uses the **XGrabKeyboard** or **XUngrabKeyboard** subroutines.

To receive **FocusIn** and **FocusOut** events in a client application, pass a window ID and **FocusChangeMask** as the *EventMask* parameter to the **XSelectInput** subroutine.

The **XFocusChangeEvent** data structure is the data structure for both **XFocusInEvent** and **XFocusOutEvent**.

All **FocusOut** events caused by a window unmap are generated after any **UnmapNotify** event, but the ordering of **FocusOut** events with respect to generated **EnterNotify**, **LeaveNotify**, **VisibilityNotify**, and **Expose** events is not constrained by the X protocol.

Using Enhanced X-Windows to Process Normal Focus Events and Focus Events While Grabbed

Normal focus events are identified by the **XFocusInEvent** or **XFocusOutEvent** data structures with **NotifyNormal** as the *mode* field. Focus events while grabbed are identified by **XFocusInEvent** or **XFocusOutEvent** data structures with **NotifyWhileGrabbed** as the *mode* field. The X Server processes normal focus events and while grabbed focus events according to the following focus scenarios:

- When the focus moves from window A to window B, window A is an inferior of window B and the pointer in window P, the X Server does the following:
  - It generates a **FocusOut** event on window A, with the *detail* field of the **XFocusOutEvent** data structure set to the value of **NotifyAncestor**.
  - It generates a **FocusOut** event on each window between window A and window B exclusive, with the *detail* field of the **XFocusOutEvent** data structure set to the value of **NotifyVirtual**.
  - It generates a **FocusIn** event on window B, with the *detail* field of the **XFocusOutEvent** data structure set to the value of **NotifyInferior**.
  - If window P is an inferior of window B, but window P is not window A or an inferior or ancestor of window A, the X Server generates a **FocusIn** event on each window below window B, down to and including, window P, with the *detail* field of the **XFocusInEvent** data structure set to the value of **NotifyPointer**.

- When the focus moves from window A to window B, and window is an inferior window of window A with the pointer in window P, the X Server generates:
  - If window P is an inferior of window A, but window P is not window A or an inferior window of window B or an ancestor of window B, the X Server generates a **FocusOut** event on each window from window P up to, but not including, window A (in that order), that has the **XFocusOutEvent** data structure with **NotifyPointer** as the *detail* field.
  - A **FocusOut** event on window A that has the **XFocusOutEvent** data structure with **NotifyInferior** as the *detail* field.
  - A **FocusIn** event on each window between window A and window B exclusive, that has the **XFocusInEvent** data structure with **NotifyVirtual** as the *detail* field.
  - A **FocusIn** event on window B that has the **XFocusInEvent** data structure with **NotifyAncestor** as the *detail* field.
- When the pointer moves from window A to window B, and window C is their least common ancestor, and the pointer is in window P, the X Server generates:
  - If window P is an inferior of window A, the X Server generates a **FocusOut** event on each window from window P up to, but not including, window A, that has the **XFocusOutEvent** data structure with **NotifyPointer** as the *detail* field.
  - A **FocusOut** event on window A that has the **XFocusOutEvent** data structure with **NotifyNonlinear** as the *detail* field.
  - A **FocusOut** event on each window between window A and window C exclusive that has the **XFocusOutEvent** data structure with **NotifyNonlinearVirtual** as the *detail* field.
  - A **FocusIn** event on each window between window C and window B exclusive that has the **XFocusInEvent** data structure with **NotifyNonlinearVirtual** as the *detail* field.
  - A **FocusIn** event on window B that has the **XFocusInEvent** data structure with **NotifyNonlinear** as the *detail* field.
  - If window P is an inferior of window B, the X Server generates a **FocusIn** event on each window below window B down to, and including, window P, that has the **XFocusInEvent** data structure with **NotifyPointer** as the *detail* field.
- When the focus moves from window A to window B on different screens with the pointer in window P, the X Server generates:
  - If window P is an inferior of window A, the X Server generates a **FocusOut** event on each window from window P up to, but not including, window A, that has the **XFocusOutEvent** data structure with **NotifyPointer** as the *detail* field.
  - A **FocusOut** event on window A that has the **XFocusOutEvent** data structure with **NotifyNonlinear** as the *detail* field.
  - If window A is not a root window, the X Server generates a **FocusOut** event on each window above window A, up to and including its root window, that has the **XFocusOutEvent** data structure with **NotifyNonlinearVirtual** as the *detail* field.
  - If window B is not a root window, the X Server generates a **FocusIn** event on each window from the root of window B down to, but not including, window B, that has the **XFocusInEvent** data structure with **NotifyNonlinearVirtual** as the *detail* field.
  - A **FocusIn** event on window B that has the **XFocusInEvent** data structure with **NotifyNonlinear** as the *detail* field.

- If window P is an inferior of window B, the X Server generates a **FocusIn** event on each window below window B down to, and including, window P that has the **XFocusInEvent** data structure with **NotifyPointer** as the *detail* field.

- When the focus moves from window A to **PointerRoot** (events sent to the window under the pointer) or the value of **None** (discards the event), and the pointer is in Window P, the X Server does the following:
  - If window P is an inferior of window A, the X Server generates a **FocusOut** event on each window from window P up to, but not including, window A that has the **XFocusOutEvent** data structure with **NotifyPointer** as the *detail* field.
  - A **FocusOut** event on window A that has the **XFocusOutEvent** data structure with **NotifyNonlinear** as the *detail* field.
  - If window A is not a root window, the X Server generates a **FocusOut** event on each window above window A up to, and including, its root window, that has the **XFocusOutEvent** data structure with **NotifyNonlinearVirtual** as the *detail* field.
  - A **FocusIn** event on the root window of all screens that has the **XFocusInEvent** data structure with **NotifyPointerRoot** or **NotifyDetailNone** as the *detail* field.
  - If the new focus window is **PointerRoot**, the X Server generates a **FocusIn** event on each window from the root window of window P down to, and including window P that has the **XFocusInEvent** data structure with **NotifyPointerRoot** as the *detail* field.
- When the focus moves from **PointerRoot** or the value of **None** to window A with the pointer in window P, the X Server generates:
  - If the old focus in **PointerRoot**, the X Server generates a **FocusOut** event on each window from window P up to, and including the root window of window P that has the **XFocusOutEvent** data structure with **NotifyPointer** as the *detail* field.
  - A **FocusOut** event on all root windows that have the **XFocusOutEvent** data structure with **NotifyPointerRoot** or **NotifyDetailNone** as the *detail* field.
  - If window A is not a root window, the X Server generates a **FocusIn** event on each window from the root of window A down to, but not including, window A that has the **XFocusInEvent** data structure with **NotifyNonlinearVirtual** as the *detail* field.
  - A **FocusIn** event on window A that has the **XFocusInEvent** data structure with **NotifyNonlinear** as the *detail* field.
  - If window P is an inferior of window A, the X Server generates a **FocusIn** event on each window below window A down to, and including, window P that has the **XFocusInEvent** data structure with **NotifyPointer** as the *detail* field.
- When the focus moves from **PointerRoot** to the value of **None**, or vice versa, with the pointer in window P, the X Server generates:
  - If the old focus is **PointerRoot**, the X Server generates a **FocusOut** event on each window from window P up to, and including the root of window P that has the **XFocusOutEvent** data structure with **NotifyPointer** as the *detail* field
  - Generates a **FocusOut** event on all root windows that have the **XFocusOutEvent** data structure with **NotifyPointerRoot** or **NotifyDetailNone** as the *detail* field.
  - A **FocusIn** event on all root windows that have the **XFocusInEvent** data structure with **NotifyDetailNone** or **NotifyPointer** as the *detail* field.

- If the new focus is **PointerRoot**, the X Server generates a **FocusIn** event on each window from the root of window P down to, and including, window P that has the **XFocusInEvent** data structure with **NotifyPointer** as the *detail* field.

Using Enhanced X-Windows to Process Focus Events Generated by Grabs

Focus events in which the keyboard grab activates are identified by **XFocusInEvent** or **XFocusOutEvent** data structures whose *mode* field is **NotifyGrab**.

When a keyboard grab activates before generating an actual **KeyPress** event to the grab with window G (the *grab_window*) and to window F (the current focus window), the X Server generates **FocusIn** and **FocusOut** events if the **XFocusInEvent** and **XFocusOutEvent** data structures have **NotifyGrab** as the *mode* fields. These events are generated as if the focus had changed from window F to window G.

Focus events in which the keyboard grab deactivates are identified by **XFocusInEvent** or **XFocusOutEvent** data structures whose *mode* field is **NotifyUngrab**.

When a keyboard grab deactivates after generating an actual **KeyRelease** event that deactivates the grab with window G (the *grab_window*) and window F (the current focus window), the X Server generates **FocusIn** and **FocusOut** events if **XFocusInEvent** and **XFocusOutEvent** data structures have **NotifyUngrab** as the *mode* fields. These events are generated as if the focus had changed from window G to window F.

Using Enhanced X-Windows to Process Keymap State Notification Events

The X Server reports **KeymapNotify** events to clients requesting information about changes in the keyboard state. To receive **KeymapNotify** events in a client application, pass a window ID and **KeymapStateMask** as the *EventMask* parameter to the **XSelectInput** subroutine. The X Server generates this event immediately after every **EnterNotify** and **FocusIn** event.

These event types use the **XKeymapEvent** data structure.

Using Enhanced X-Windows to Process Expose Events

The X protocol does not guarantee to preserve the contents of window regions when the windows are obscured or reconfigured. Some implementations can preserve the contents of windows, but many other implementations destroy the contents of windows when the windows are exposed. Client applications should be designed to restore the contents of an exposed window region. (An exposed window region describes a formerly obscured window whose region or piece of a region becomes visible.) the X Server sends exposure events describing the window and the region of the window that has been exposed. Some client applications redraw the entire window, while other client applications redraw only the exposed region.

The X Server reports **Expose** events to clients requesting information about when the contents of the window regions have been lost. The X Server generates **Expose** events when a window region becomes viewable. It reports **Expose** events contiguously when all the regions are exposed by some action, such as raising a window. The X Server cannot generate **Expose** events to **InputOnly** windows.

To receive **Expose** events in a client application, pass a window ID and **ExposureMask** as the *EventMask* parameter to the **XSelectInput** subroutine.

Use the **XExposeEvent** data structure with **Expose** events.

Using Enhanced X-Windows to Process GraphicsExpose and NoExpose Events

The X Server reports **GraphicsExpose** events to clients requesting information when a destination region could not be computed during a graphics request. Clients initiate graphics request with the **XCopyArea** or **XCopyPlane** subroutines.

The X Server generates a **GraphicsExpose** event whenever a destination region cannot be computed because of an obscured or out-of-bounds source region. The X Server reports contiguously all of the regions exposed by a graphics request (for example, copying an area of a drawable to a destination drawable).

The X Server generates **NoExpose** events whenever a graphics request that might produce a **GraphicsExpose** event does not produce any graphics events. In other words, the client requests a **GraphicsExpose** event, but receives a **NoExpose** event instead.

To receive **GraphicsExpose** or **NoExpose** events in a client application, set the *graphics_exposures* field of the **XGCValues** data structure to the value of **True**. The *graphics_exposures* field is set using the **XCreateGC** or **XSetGraphicsExposures** subroutines.

Use the **XGraphicsExposeEvent** and the **XNoExposeEvent** data structures for these event types.

Using Enhanced X-Windows to Process CirculateNotify Events

The X Server reports **CirculateNotify** events to clients requesting information about when a window changes its position in the stack. The X Server generates this event type whenever a window is actually restacked as a result of a client application calling the **XCirculateSubwindows**, **XCirculateSubwindowsUp**, or **XCirculateSubwindowsDown** subroutines.

To receive **CirculateNotify** events in a client application, set the *event_mask* field of the window to **StructureNotifyMask** or the *event_mask* field of the parent window to **SubstructureNotifyMask**.

Use the **XCirculateEvent** data structure for this event type.

Using Enhanced X-Windows to Process ConfigureNotify Events

The X Server reports **ConfigureNotify** events to clients requesting information about actual changes to the state of a window, for example, size, position, border, and stacking order. It generates a **ConfigureNotify** event when a client application completes one of the following configure window subroutines:

- **XConfigureWindow** which reconfigures window size, position, border, and stacking order.
- **XLowerWindow**, **XRaiseWindow**, or **XRestackWindows** which change the window position in the stacking order.
- **XMoveWindow** which moves the window.
- **XResizeWindow** which changes the window size.
- **XMoveResizeWindow** which changes the size and location of the window.
- **XMapRaised** which changes the position of a mapped window in the stacking order.
- **XSetWindowBorderWidth** which changes the border width of a window.

To receive a **ConfigureNotify** event, set the *event_mask* field of the window to **StructureNotifyMask** or the *event_mask* field of the parent window to **SubstructureNotifyMask.**

Use the **XConfigureEvent** data structure for this event type.

Using Enhanced X-Windows to Process CreateNotify Events

The X Server reports **CreateNotify** events to clients requesting information about creation of windows. The X Server generates this event when a client application creates a window with the **XCreateWindow** or **XCreateSimpleWindow** subroutine.

To receive **CreateNotify** events, set the *event_mask* field of the window to **SubstructureNotifyMask**. Creating any children then generates an event.

Use the **XCreateWindowEvent** data structure for this event type.

Using Enhanced X-Windows to Process DestroyNotify Events

The X Server reports **DestroyNotify** events to clients requesting information about windows that are destroyed. It generates this event whenever a client application destroys a window with the **XDestroyWindow** or **XDestroySubwindows** subroutines.

A **DestroyNotify** event is generated on all inferiors of a window before being generated on the window itself. The ordering among siblings and across subhierarchies is not constrained otherwise by the X protocol.

To receive **DestroyNotify** events, set the *event_mask* field of the window to **StructureNotifyMask** or the *event_mask* field of the parent to **SubstructureNotifyMask.**

Use the **XDestroyWindowEvent** data structure for this event type.

Using Enhanced X-Windows to Process GravityNotify Events

The X Server reports **GravityNotify** events to clients requesting information when a window is moved because of a change in the size of the parent window. The X Server generates this event when a client application moves a child window as a result of resizing its parent using the **XConfigureWindow, XMoveResizeWindow**, or **XResizeWindow** subroutine.

To receive **GravityNotify** events, set the *event_mask* field of the window to **StructureNotifyMask** or the *event_mask* field of the parent to **SubstructureNotifyMask.**

Use the **XGravityEvent** data structure for this event type.

Using Enhanced X-Windows to Process MapNotify Events

The X Server reports **MapNotify** events to clients requesting information about which windows are mapped. It generates this event type whenever a client application changes the state of a window from unmapped to mapped with the **XMapWindow, XMapRaised,** or **XMapSubwindows** subroutines.

To receive **MapNotify** events, set the *event_mask* field of the window to **StructureNotifyMask** or the *event_mask* field of the parent to **SubstructureNotifyMask.**

Use the **XMapEvent** data structure for this event type.

Using Enhanced X-Windows to Process MappingNotify Events

The X Server reports **MappingNotify** events to all clients. There is no mechanism to cancel generation of this event. The X Server generates this event whenever a client calls one of the following subroutines:

XSetModifierMapping	Specifies the keycodes to be used as modifiers.
XChangeKeyboardMapping	Changes the keyboard mapping.
XSetPointerMapping	Sets the pointer mapping.

Use the **XMappingEvent** data structure for this event.

Using Enhanced X-Windows to Process ReparentNotify Events

The X Server reports **ReparentNotify** events to clients requesting information about changing the parent of a window. It generates this event when a client application calls the **XReparentWindow** subroutine and the window is actually reparented.

To receive **ReparentNotify** events, set the *event_mask* field of the window to **StructureNotifyMask** or the *event_mask* field of either the old or the new parent to **SubstructureNotifyMask**, in which case reparenting any child generates an event.

Use the **XReparentEvent** data structure for this event type.

Using Enhanced X-Windows to Process UnmapNotify Events

The X Server reports **UnmapNotify** events to clients requesting information about which windows are unmapped. The X Server generates this event whenever a client application changes the window state from mapped to unmapped with the **XUnmapWindow** or **XUnmapSubwindows** subroutine.

To receive **UnmapNotify** events, set the *event_mask* field of the window to **StructureNotifyMask** or the *event_mask* field of the parent to **SubstructureNotifyMask**, in which case unmapping the child generates the event.

Use the **XUnmapEvent** data structure for this event type.

Using Enhanced X-Windows to Process VisibilityNotify Events

The X Server reports **VisibilityNotify** events to clients requesting any change in the visibility of the specified window. (A region of a window is visible if it can actually be seen on the screen.) The X Server generates this event whenever the visibility of the window changes state. However, **VisibilityNotify** events are not generated for **InputOnly** windows or subwindows.

All **VisibilityNotify** events caused by a hierarchy change are generated after any hierarchy event (**UnmapNotify**, **MapNotify**, **ConfigureNotify**, **GravityNotify**, **CirculateNotify**) caused by that change. Any **VisibilityNotify** event on a window is generated before any **Expose** events on that window, but it is not required that all **VisibilityNotify** events on all windows are generated before all **Expose** events on all windows. The ordering of **VisibilityNotify** events with respect to **FocusOut**, **EnterNotify**, and **LeaveNotify** events is not constrained by the X protocol.

The X Server ignores all the subwindows in determining the visibility state of a specified window and processes **VisibilityNotify** events according to the following:

- When the window changes state from partially obscured, fully obscured, or not viewable to viewable and completely unobscured, the X Server generates the event with **VisibilityUnobscured** as the *state* field of the **XVisibilityEvent** structure.
- When the window changes state from viewable and completely unobscured or not viewable to viewable and partially obscured, the X Server generates the event with **VisibilityPartiallyObscured** as the *state* field of the **XVisibilityEvent** structure.
- When the window changes state from viewable and completely unobscured, viewable and partially obscured, or not viewable to viewable and fully obscured, the X Server generates

the event with **VisibilityFullyObscured** as the *state* field of the **XVisibilityEvent** structure.

To receive **VisibilityNotify** events, set the *event_mask* field of the window to **VisibilityChangeMask**.

Use the **XVisibilityEvent** data structure for this event type.

Using Enhanced X-Windows to Process CirculateRequest Events

The X Server reports **CirculateRequest** events to clients requesting information about when another client initiates a window request on a specified window. The X Server generates this event type when a client initiates a circulate window request on a window and a subwindow needs to be restacked. The client initiates a circulate window request with the **XCirculateSubwindows**, **XCirculateSubwindowsUp**, or **XCirculateSubwindowsDown** subroutines.

To receive a **CirculateRequest** event, set the *event_mask* field of the window to **SubstructureRedirectMask**. Then, in the future, a circulate window request for the specified window will not be executed and the position of the window in the stack is not changed. Instead, the client will receive a **CirculateRequest** event.

Use the **XCirculateRequestEvent** data structure for this event type.

Using Enhanced X-Windows to Process ConfigureRequest Events

The X Server reports **ConfigureRequest** events to clients requesting information about when another client initiates a configure request on any child of a specified window. The configure window request attempts to reconfigure the window size, position, border, or stacking order. The X Server generates this event whenever a different client initiates a configure window request on a window by using the **XConfigureWindow**, **XLowerWindow**, **XRaiseWindow**, **XMapRaised**, **XMoveResizeWindow**, **XMoveWindow**, **XResizeWindow**, **XRestackWindows**, or **XSetWindowBorderWidth** subroutines.

To receive **ConfigureRequest** events, set the *event_mask* field of the window to **SubstructureRedirectMask**.

Use the **XConfigureRequestEvent** data structure for this event type.

Using Enhanced X-Windows to Process MapRequest Events

The X Server reports **MapRequest** events to clients wanting information about a request by another client to map windows. (A window is considered mapped when a map window request completes.) The X Server generates this event whenever a different client initiates a map window request on an unmapped window whose *override_redirect* field is the value of **False**. Clients can initiate map window requests using the **XMapWindow**, **XMapRaised**, or **XMapSubwindows** subroutines.

To receive **MapRequest** events, set the **SubstructureRedirectMask** bit in the *event_mask* field of the window. This means that another client's attempts to map a child window by calling one of the map window request subroutines is intercepted, and you are sent a **MapRequest** event instead. Thus, this event gives your window manager client the ability to control the placement of subwindows.

Use the **XMapRequestEvent** data structure for this event type.

Using Enhanced X-Windows to Process ResizeRequest Events

The X Server reports **ResizeRequest** events to clients requesting information when another client attempts to change the size of a window. A client can attempt to change the size of a window by calling the **XConfigureWindow**, **XResizeWindow**, or **XMoveResizeWindow** subroutines.

To receive **ResizeRequest** events, set the **ResizeRedirect** bit in the *event_mask* field of the window. Any attempts by other clients to resize the window are then redirected.

Use the **XResizeRequestEvent** data structure for this event type.

Using Enhanced X-Windows to Process Colormap State Notification Events

The X Server reports **ColormapNotify** events to clients requesting information about when the colormap changes and when a colormap is installed or uninstalled. The X server generates this event type whenever a client application:

- Changes the *colormap* field of the **XSetWindowAttributes** structure using the **XChangeWindowAttributes**, **XFreeColormap**, or **XSetWindowColormap** subroutines.
- Installs or uninstalls the colormap using the **XInstallColormap** or **XUninstallColormap** subroutines.

To receive **ColormapNotify** events, set the **ColormapChangeMask** bit in the *event_mask* field of the window.

Use the **XColormapEvent** data structure for this event type.

Using Enhanced X-Windows to Process ClientMessage Events

The X Server generates **ClientMessage** events when a client calls the **XSendEvent** subroutine. The **XSendEvent** subroutine identifies the destination window, determines which clients should receive the specified events, and ignores any active grabs.

Use the **XClientMessageEvent** data structure for this event type.

Using Enhanced X-Windows to Process PropertyNotify Events

The X Server reports **PropertyNotify** events to clients requesting information about property changes for a specified window. The X Server generates this event when a client application calls the **XChangeProperty**, **XDeleteProperty**, **XRotateWindowProperties**, or **XGetWIndowProperty** subroutines.

To receive **PropertyNotify** events, set the **PropertyChangeMask** bit in the *event_mask* field of the window.

Use the **XPropertyEvent** data structure for this event type.

Using Enhanced X-Windows to Process SelectionClear Events

The X Server reports **SelectionClear** events to the current owner of a selection. This event is generated on the window losing ownership of the selection to a new owner. The X Server generates this event whenever a client calls the **XSetSelectionOwner** subroutine.

Use the **XSelectionClearEvent** data structure for this event type.

Using Enhanced X-Windows to Process SelectionRequest Events

The X Server reports **SelectionRequest** events to the owner of a selection when a client requests a selection conversion using the **XConvertSelection** subroutine and the specified selection is owned by a window.

The client that owns the selection should do the following:

- Convert the selection based on the atom contained in the *target* field.

- Store the result as the property specified on the *requestor* window and send a **SelectionNotify** event to creator of the *requestor* window, if a value was specified in the *Property* field.
- Choose a property name on the *requestor* window and send a **SelectionNotify** event giving the actual name, if the *property* field is the value of **None.**
- Send a **SelectionNotify** event with the *property* field set to the value of **None**, if the selection cannot be converted as requested.

Use the **XSelectionRequestEvent** data structure for this event type.

Using Enhanced X-Windows to Process SelectionNotify Events

The X Server generates **SelectionNotify** events in response to an **XConvertSelection** request when there is no owner for the selection. If the selection has an owner, the **SelectionNotify** event should be generated by the owner using the **XSendEvent** subroutine.

If the selection has been converted and stored as a property or a selection conversion could not be performed, the **SelectionNotify** event should be sent to the *requestor* window.

If the value of **None** is specified in the *property* field of the **ConverSelection** protocol request, the selection owner should choose a property name, store the result as that property on the *requestor* window, and then send a **SelectionNotify** event giving that actual property name.

Use the **XSelectionEvent** data structure for this event type.

Using Enhanced X-Windows to Select Events

There are two ways to select the events reported to your client application. One way is to set the *event_mask* field of the **XSetWindowAttributes** data structure when you use the **XCreateWindow** and **XChangeWindowAttributes** subroutine. The second way is to use the **XSelectInput** subroutine.

Events are reported relative to a window. If a window requests an event, it usually propagates to the closest ancestor that does not request an event unless the *do_not_propagate* mask prohibits propagation.

Setting the *event_mask* field of a window overrides any previous call to the same window from the same client, but not from other clients. Multiple clients can select for the same events on the same window with the following restrictions:

- Multiple clients can select events on the same window because the event masks are disjointed. After the X Server generates an event, it reports the event to all interested clients.
- Only one client at a time can select the **CirculateRequest, ConfigureRequest,** or **MapRequest** events, which are associated with the **SubstructureRedirectMask** event mask.
- Only one client at a time can select a **ResizeRequest** event, which is associated with the **ResizeRedirectMask** event mask.
- Only one client at a time can select a **ButtonPress** event, which is associated with the **ButtonPressMask** event mask.

The server reports the event to all interested clients.

Using Enhanced X-Windows to Handle the Output Buffer and the Event Queue

Under some circumstances, event subroutines flush the output buffer. The output buffer is an area used by the **Xlib** library to store requests. Some event subroutines flush the output buffer if the subroutine would block or not return an event. All requests residing in the output buffer that have not yet been sent are transmitted to the X Server. Conversely, these subroutines differ in the additional tasks they might perform. For example, the **XSync** subroutine not only flushes the output buffer, but it can also discard all events in the event queue.

Using Enhanced X-Windows to Select Events Using a Predicate Procedure

The **XIfEvent**, **XCheckIfEvent**, and **XPeekIfEvent** subroutines require a predicate procedure that determines if the event matches the event specified in the corresponding subroutine. The predicate procedure must decide only if the event is useful and must not call the **Xlib** library subroutines. This predicate procedure is called from within the event routine, which must lock so that the event queue is consistent in a multi-threaded environment.

The predicate procedure for these subroutines is defined as:

Bool (*predicate)(*DisplayPtr*, *Event*, *Arguments***);**
Display **DisplayPtr*;
XEvent **Event*;
char **Arguments*;

displayptr	Specifies the connection to the X Server.
event	Specifies a pointer to the **XEvent** structure.
arguments	Specifies the arguments passed in from the **XIfEvent**, **XCheckIfEvent**, or **XPeekIfEvent** subroutines.

The predicate procedure is called once for each event in the queue until it finds a match between the event in the queue and the event specified by the corresponding subroutine. This procedure returns the value of **True** if it finds a match. It returns the value of **False** if it does not find a match.

Using the Enhanced X-Windows Default Error Handler

Two default error handlers reside in the library. One is used for typically fatal conditions (for example, the connection to the display fails due to a system crash), while the other is used for error events from the X Server. These error handlers can be changed to user-supplied routines as often as necessary. To reinvoke either error handler, pass the value of **NULL** pointer. The default is to print an explanatory message and exit.

Use the **XErrorEvent** data structure for this event type.

Related Information

The **XAnyEvent** data structure, **XButtonPressedEvent** data structure, **XButtonReleasedEvent** data structure, **XCirculateEvent** data structure, **XCirculateRequestEvent** data structure, **XClientMessageEvent** data structure, **XColormapEvent** data structure, **XConfigureEvent** data structure, **XConfigureRequestEvent** data structure, **XCreateWindowEvent** data structure, **XCrossingEvent** data structure **XDestroyWindowEvent** data structure, **XEnterWindowEvent** data structure, **XErrorEvent** data structure, **XEvent** data structure, **XExposeEvent** data structure, **XFocusChangeEvent** data structure,, **XFocusInEvent** data structure, **XFocusOutEvent** data structure, **XGCValues** data structure **XGraphicsExposeEvent** data structure, **XGravityEvent** data structure,

XKeyPointerMovedEvent data structure, **XKeyPressedEvent** data structure, **XKeyReleasedEvent** data structure, **XKeymapEvent** data structure, **XLeaveWindowEvent** data structure, **XMapEvent** data structure, **XMappingEvent** data structure, **XMapRequestEvent** data structure, **XNoExposeEvent** data structure, **XPropertyEvent** data structure, **XReparentEvent** data structure, **XResizeRequestEvent** data structure, **XSelectionClearEvent** data structure, **XSelectionEvent** data structure, **XSelectionRequestEvent** data structure, **XUnmapEvent** data structure, **XVisibilityEvent** data structure.

The **ButtonPress** event, **ButtonRelease** event, **CirculateNotify** event, **CirculateRequest** event, **ClientMessage** event, **ColormapNotify** event, **ConfigureNotify** event, **ConfigureRequest** event, **CreateNotify** event, **DestroyNotify** event, **EnterNotify** event, **Expose** event, **FocusIn** event, **FocusOut** event, **GraphicsExposure** event, **GravityNotify** event, **KeyPress** event, **KeyRelease** event, **KeymapNotify** event, **LeaveNotify** event, **MapNotify** event, **MapRequest** event, **MappingNotify** event, **MotionNotify** event, **NoExposure** event, **PropertyNotify** event, **ReparentNotify** event, **ResizeRequest** event, **SelectionClear** event, **SelectionNotify** event, **SelectionRequest** event, **UnmapNotify** event, **VisibilityNotify** event.

The **<X11/X.h>** header file, **<X11/Xlib.h>** header file, **<X11/Xproto.h>** header file.

The **IsCursorKey** macro, **IsFunctionKey** macro, **IsKeypadKey** macro, **IsMiscFunctionKey** macro, **IsModifierKey** macro, **IsPFKey** macro.

The **XChangeActivePointerGrab** subroutine, **XChangeGC** subroutine, **XChangeKeyboardMapping** subroutine, **XChangeProperty** subroutine, **XChangeWindowAttributes** subroutine, **XCheckIfEvent** subroutine, **XCheckMaskEvent** subroutine, **XCheckTypedEvent** subroutine, **XCheckTypedWindowEvent** subroutine, **XCheckWindowEvent** subroutine, **XCirculateSubwindows** subroutine, **XCirculateSubwindowsDown** subroutine, **XCirculateSubwindowsUp** subroutine, **XConfigureWindow** subroutine, **XConvertSelection** subroutine, **XCopyArea** subroutine, **XCopyPlane** subroutine, **XCreateGC** subroutine, **XCreateSimpleWindow** subroutine, **XCreateWindow** subroutine, **XDeleteProperty** subroutine, **XDestroySubwindows** subroutine, **XDestroyWindow** subroutine, **XDisplayName** subroutine, **XEventsQueued** subroutine, **XFlush** subroutine, **XFreeColormap** subroutine, **XGetErrorDatabaseText** subroutine, **XGetErrorText** subroutine, **XGetInputFocus** subroutine, **XGetWindowProperty** subroutine, **XGrabButton** subroutine, **XGrabKeyboard** subroutine, **XGrabPointer** subroutine, **XIfEvent** subroutine, **XInstallColormap** subroutine, **XKeycodeToKeysym** subroutine, **XKeysymToKeycode** subroutine, **XKeysymToString** subroutine, **XLookupKeysym** subroutine, **XLookupString** subroutine, **XLowerWindow** subroutine, **XMapRaised** subroutine, **XMapSubwindows** subroutine, **XMapWindow** subroutine, **XMaskEvent** subroutine, **XMoveResizeWindow** subroutine, **XMoveWindow** subroutine, **XNextEvent** subroutine, **XPeekEvent** subroutine, **XPeekIfEvent** subroutine, **XPending** subroutine, **XPutBackEvent** subroutine, **XRaiseWindow** subroutine, **XRebindKeysym** subroutine, **XRefreshKeyboardMapping** subroutine, **XReparentWindow** subroutine, **XResizeWindow** subroutine, **XRestackWindows** subroutine, **XRotateWindowProperties** subroutine, **XSelectInput** subroutine, **XSendEvent** subroutine, **XSetErrorHandler** subroutine, **XSetGraphicsExposures** subroutine, **XSetInputFocus** subroutine, **XSetIOErrorHandler** subroutine, **XSetModifierMapping** subroutine, **XSetPointerMapping** subroutine, **XSetSelectionOwner** subroutine, **XSetWindowAttributes** data structure, **XSetWindowBorderWidth** subroutine, **XSetWindowColormap** subroutine, **XStringToKeysym** subroutine, **XSync** subroutine, **XWindowEvent** subroutine, **XUngrabKeyboard** subroutine, **XUngrabPointer** subroutine, **XUninstallColormap** subroutine, **XUnmapSubwindows** subroutine, **XUnmapWindow** subroutine, **XWarpPointer** subroutine.

Enhanced X-Windows Graphics Resource Subroutines Overview

Enhanced X-Windows graphics resource subroutines allow you to manipulate colormaps, pixmaps, graphics context (**GC**) state and use **GC** convenience routines.

A number of resources are used when performing graphics operations in Enhanced X-Windows. Most attributes of graphics operations (for example, foreground color, background color, line style, and other graphics) are stored in resources called graphics contexts (**GC**). Most graphics operations take a graphics context (**GC**) as an argument. While sharing **GC**s between applications is possible, it is not encouraged. Applications should use their own **GC**s when performing operations.

Windows have a colormap associated with them that provides a level of indirection between pixel values and color displayed on the screen. Many hardware displays have a single colormap; therefore, the primitives are written to share colormap entries between applications. Because colormaps are associated with windows, Enhanced X-Windows supports displays with multiple colormaps and different types of colormaps. If there are not enough colormap resources in the display, some windows may not be displayed in their true colors. A window manager can set the displayed windows in their true colors if more than one colormap is required for the color resources the applications are using.

Off-screen memory or pixmaps are often used to define frequently-used images for later use in graphics operations. Pixmaps are also used to define tiles or patterns for window backgrounds, borders, or cursors. (A single-bit plane pixmap is sometimes referred to as a bitmap). There may not be an unlimited amount of off-screen memory; therefore, it should be regarded as a precious resource.

Graphics operations can be performed to windows or pixmaps, which are also called drawables. Each drawable exists on a single screen or root window. The drawable can be used only on that root window. **GC**s can be used with drawables of matching root windows and depths.

Creating Enhanced X-Windows Colormaps

The **Xlib** library provides subroutines to create, allocate, free, set and manipulate colormaps. The introduction of color changes the view a programmer should take when dealing with a bitmap display. For example, when printing text, you write in a color or pixel value rather than by setting or clearing bits. Hardware imposes limits (number of significant bits, for example). Typically, you allocate particular pixel values or sets of values. If read-only, the pixel values can be shared among multiple applications. If read/write, the pixel values are exclusively owned by the program, and the color cell associated with the pixel value can be changed at will.

Each pixel on a monochrome (black and white) display has one bit of information associated with it; the bit is either zero (0) or one (1).

Color displays, however, need multiple bits per pixel to properly regulate the red, green, and blue color guns that provide the full range of visible colors available to the particular display. For example, assume a display has 4 bits per pixel, numbered 0, 1, 2, and 3. The display is said to be 4 planes deep. One plane contains all the bit 0s, the second contains the bit 1s, the third contains the bit 2s, and the fourth contains the bit 3s.

Some subroutines manipulate the representation of color on the screen. For each possible value a pixel can take on a display, a color cell is defined in the colormap. For example, if a display is 4 bits deep, pixel values 0 through 15 are defined. A colormap is a collection of the color cells. A color cell consists of a triple of red, green, and blue. As each pixel is read out of display memory, its value is taken and looked up in the colormap. The values of the cell determine what color is displayed on the screen. On a multiplane display with a black and white monitor (grayscale, but not color), these values may be combined to determine the brightness on the screen.

One or more colormaps may reside at one time on certain hardware. The **XInstallColormap** subroutine installs a colormap; while the **DefaultColormap** macro returns the default colormap. The **DefaultVisual** macro returns the default visual type for the specified screen. Colormaps are local to a particular screen. Possible visual types are represented by **StaticGray, GrayScale, StaticColor, PseudoColor, TrueColor**, or **DirectColor**.

Allocating Enhanced X-Windows Colormaps

The **Xlib** library provides subroutines to allocate or deallocate pixel values for colors that you need to display. There are two ways of allocating color cells: explicitly as **read only** entries by pixel value or as **read/write** entries, where you can allocate a number of color cells and planes simultaneously. The read or write cells allocated are not defined colors until the colors are set with the **XStoreColor** or the **XStoreColors** subroutine.

Screens always have a default colormap. Programs typically allocate cells out of a common colormap. Writing applications that monopolize color resources is discouraged. On a screen that cannot load the colormap or cannot have a fully independent colormap, only certain kinds of allocations work.

To determine the color names, the X Server uses a color database. On an AIX-based system, this database is **/usr/lpp/X11/rgb/rgb**. A printable copy of this database is stored in **/usr/lpp/X11/rgb/rgb.txt**.

The name and contents of this file are operating-system specific and possibly screen specific. Although you can change the values in a read/write color cell that is allocated by another application, this is not encouraged.

Manipulating Enhanced X-Windows Standard Colormaps

Applications with color palettes, smooth-shaded drawings, or digitized images demand large numbers of colors. In addition, these applications often require an efficient mapping from color triples to pixel values that display the appropriate colors.

As an example, consider a 3D display program designed to draw a smoothly shaded sphere. At each pixel in the image of the sphere, the program computes the intensity and color of light reflected back to the viewer. The result of each computation is a triple of red, green, and blue co-efficients in the range 0.0 to 1.0. To draw the sphere, the program needs a colormap that provides a large range of uniformly distributed colors. The colormap should be arranged so that the program can convert its RGB triples into pixel values very quickly because drawing the entire sphere requires many such conversions.

On many current workstations the display is limited to 256 or fewer colors. Applications must allocate colors carefully, not only to make sure they cover the entire range needed, but also to make use of as many of the available colors as possible. On a typical X Server display, many applications are active at once. Most workstations have only one hardware lookup table for colors; therefore, only one application colormap can be installed at a given time. The application using the installed colormap is displayed correctly while the other applications are displayed with false colors.

As another example, consider users who run an image processing program to display earth-resources data. The image processing program needs a colormap set up with 8 reds, 8 greens, and 4 blues which is a total of 256 colors. Because some colors are already in use in the default colormap, the image processing program allocates and installs a new colormap.

The users decide to alter some of the colors in the image. They invoke a color palette program to mix and choose colors. The color palette program also needs a colormap with 8 reds, 8 greens, and 4 blues; therefore, just as the image-processing program, the color palette program must allocate and install a new colormap.

Because only one colormap can be installed at a time, the color palette can be displayed incorrectly when the image-processing program is active. Conversely, when the palette program is active, the image can be displayed incorrectly. Users can never match or compare colors in the palette and image. Contention for colormap resources can be reduced if applications with similar color needs share colormaps.

As another example, the image processing program and the color palette program share the same colormap if there exists a convention that describes how the colormap is set up. Whenever either program is active, both are displayed correctly.

The standard colormap properties define a set of commonly used colormaps. Applications that share these colormaps and conventions display true colors more often and provide a better interface to the users.

Using Enhanced X-Windows Standard Colormaps

Standard colormaps allow applications to share commonly used color resources. This feature allows many applications to be displayed in true colors simultaneously, even when each application needs an entirely filled colormap.

Several standard colormaps are described. Usually, these colormaps are created by a window manager. Applications should use the standard colormaps if they already exist. If the standard colormaps do not exist, applications should create new standard colormaps.

Using Enhanced X-Windows Standard Colormap Properties and Atoms

Several standard colormaps are available. Each standard colormap is defined by a property, and each property is identified by an atom. The following is a list of atom names and the colormap associated with each one.

- The **RGB_DEFAULT_MAP** atom names a property with the value of **XStandardColormap**. The property defines an RGB subset of the system default colormap. Some applications need only a few RGB colors and may be able to allocate these colors from the system default colormap. By using the system default colormap, you ensure that there are fewer active colormaps in the system. This, in turn, ensures that your application is more likely to be displayed with the correct colors at all times.

  A typical allocation for the **RGB_DEFAULT_MAP** on 8-plane displays is 6 reds, 6 greens, and 6 blues. This allocation gives 216 uniformly distributed colors (6 intensities of 36 different hues) and still leaves 40 elements of a 256-element colormap available for special-purpose colors for text, borders and other fields.

- The **RGB_BEST_MAP** atom names a property with the value of **XStandardColormap**. The property defines the best RGB colormap available on the display.

  Many image processing and 3D applications need to use all available colormap cells and distribute as many perceptually distinct colors as possible over those colormap cells. This implies that more green values may be available than red values, as well as more green values or red values than blue values.

  On an 8-plane pseudocolor display, **RGB_BEST_MAP** is a `3/3/2` allocation. On a 24-plane direct color display, **RGB_BEST_MAP** is an `8/8/8` allocation. On other displays, **RGB_BEST_MAP** allocation is set by the user or the owner of the display.

- The **RGB_RED_MAP** atom names a property with the value of **XStandardColormaps**. This property defines an all-red colormap, which is used to make color-separated images.

- The **RGB_GREEN_MAP** atom names a property with the value of **XStandardColormaps**. This property defines an all-green colormap, which is used to make color-separated images
- The **RGB_BLUE_MAP** atom names a property with the value of **XStandardColormaps**. This property defines an all-blue colormap, which is used to make color-separated images.

  For example, a user might generate a full-color image on an 8-plane display both by rendering an image three times (once with high color resolution in red, once with high color resolution in green, and once with high color resolution in blue) and by multiple-exposing a single frame in a camera.
- The **RGB_GRAY_MAP** atom names a property with the value of **XStandardColormap**. This property describes the best gray-scale colormap available on the display.

Determining Enhanced X-Windows Resident Colormaps

Window manager applications usually install and uninstall colormaps. Thus, these tasks should not be performed by normal client applications.

The X Server maintains a subset of the installed colormaps in an ordered list called the required list. The length of the required list is the minimum number of installed colormaps specified for the screen when the connection is opened to the server. Initially, only the default colormap for a screen is installed, but it is not in the required list. The X Server maintains the required list as follows:

- If a colormap resource ID is passed to the *ColorMap* argument, the **XInstallColormap** subroutine adds the colormap to the top of the list. If necessary, it truncates a colormap at the bottom of the list so that the maximum length of the list is not exceeded.
- If a colormap resource ID is passed to the *ColorMap* argument and this colormap is in the required list, the **XUninstallColormap** subroutine removes the colormap from the required list.

A colormap is not implicitly added to the required list when it is installed by the server. Nor is a colormap in the required list implicitly uninstalled by the server.

Reading Enhanced X-Windows Entries in a Colormap

The **XQueryColor** and **XQueryColors** subroutines return the red, green, and blue color values stored in the specified colormap for the pixel value passed in the *pixel* field of the **XColor** data structures. The values returned for an unallocated entry are undefined. These subroutines also set the *flags* field in the **XColor** data structure to all three colors.

Creating and Freeing Enhanced X-Windows Pixmaps

A pixmap is a rectangle of pixels that has the width and the height in pixels. A pixmap is as deep as the pixel has bits. Pixmaps are used only on the screen on which they are created. Pixmaps can be represented in storage in either **XYformat** or **Zformat**.

In **XYformat**, each plane in the pixmap is represented as a bitmap; the bitmaps appear in storage from most significant to least significant in sequence.

In **Zformat**, the pixels are stored row by row, top to bottom, left to right within the row.

The **Xlib** library provides subroutines to create or free a pixmap. Pixmaps can be used for operations such as defining cursors as tiling patterns or as the source for certain raster operations. Most graphic requests can operate on a window or a pixmap. Some programs may want to manipulate pixels that are displayed on the screen later. Some subroutines move pixels from the program to the window system or from the window system to the program.

Defining Enhanced X-Windows Bitmaps

A bitmap is a rectangle of bits that has a width in pixels and a height in pixels. A bitmap is a pixmap of depth one. At depth one, Z format and XYformat are identical and are not specified in a bitmap.

Manipulating Enhanced X-Windows Graphics Context or State

Most components of graphics operations are stored in graphics contexts (**GC**s). These components include line width, line style, plane mask, foreground, background, tile, stipple, clipping region, end style, join style, and others. Graphics operations (for example, drawing lines) use these values to determine the actual drawing operation. Extensions to Enhanced X-Windows may add more components to **GC**s. The **Xlib** library provides calls for changing the state of **GC**s.

The **Xlib** library implements a write-back cache for all elements of a **GC** that are not resource IDs to allow it to implement the transparent coalescing changes to **GC**s. **GC**s are neither expected nor encouraged to be shared between client applications; therefore, this write-back caching should not present problems. Applications cannot share **GC**s without external synchronization; therefore, sharing **GC**s between applications is not encouraged.

The specified fields of the new graphics context in *ValuemaskCreate* are set to the values passed in the values parameter. The other fields default to the following values:

Field	**Value**
function	**GXcopy**
plane_mask	All 1s
foreground	0
background	1
line_width	0
line_style	**LineSolid**
cap_style	**CapButt**
join_style	**JoinMiter**
fill_style	**FillSolid**
fill_rule	**EvenOddRule**
arc_mode	**ArcPieSlice**
tile	Pixmap of unspecified size filled with foreground pixel. (If there is no client specified pixel, pixel 0 is used.)
stipple	Pixmap of depth 1 of unspecified size filled with 1s
t_s_x_origin	0
t_s_y_origin	0
font	Implementation Specific
subwindow_mode	**ClipByChildren**
graphics_exposures	**True**
clip_x_origin	0

clip_y_origin	0
clip_mask	**None**
dash_offset	0
dashes	4 (the list [4,4])

The *function* field of the **GC** is used when a section of the destination screen is updated with source bits from somewhere else. The *function* field defines how the new destination bits are to be computed from the source bits and the old destination bits.

The **GXcopy** value is typically the most useful function because it works on a color display, but special applications can use other functions, particularly with certain planes of a color display. The functions, which are defined in **<X11/X.h>**, are:

Function Name	**Hex Code**	**Operation**
GXclear	0x0	0
GXand	0x1	source AND destination
GXandReverse	0x2	source AND NOT destination
GXcopy	0x3	source
GXandInverted	0x4	(NOT source) AND destination
GXnoop	0x5	destination
GXxor	0x6	source XOR destination
GXor	0x7	source OR destination
GXnor	0x8	(NOT source) AND (NOT destination)
GXequiv	0x9	(NOT source) XOR destination
GXinvert	0xa	NOT destination
GXorReverse	0xb	source OR (NOT destination)
GXorInverted	0xd	(NOT source) OR destination
GXnand	0xe	(NOT source) OR (NOT destination)
GXset	0xf	1

Note: Using display functions other than **GXcopy**, such as **GXxor** and **GXinvert**, on a color display can result in undefined pixel values.

For example, if an X Server on a four-plane display performs the **GXinvert** on pixel value `3`, the result is pixel value `12`. The color currently associated with pixel value `12` is displayed, whether or not it is a defined value.

Many graphics operations depend on either pixel values or planes in a **GC**. The *plane_mask* field specifies which planes of the display are to be modified (one bit per plane). A monochrome display has only one plane and this plane is the least-significant bit of the word. As planes are added to the display hardware, they occupy more significant bits in the plane mask.

In graphics operations, given a source and destination pixel, the result is computed bitwise on corresponding bits of the pixels. That is, a Boolean operation is performed in each bit plane. The *plane_mask* field restricts the operation to a subset of planes. The **AllPlanes** macro can be used to refer to all planes of a display simultaneously. The result is computed by the following:

```
((src FUNC dst) AND plane-mask) OR (dst AND (NOT plane-mask))
```

Most operations use a **GC**. The contents of the **GC** are private to the **Xlib** library. Several procedures take structures from the **XGCValues** data structure.

Related Information

The **XColor** data structure, **XGCValues** data structure.

The **XAllocColor** subroutine, **XAllocColorCells** subroutine, **XAllocColorPlanes** subroutine, **XAllocNamedColor** subroutine, **XCopyColormapAndFree** subroutine, **XCreateColormap** subroutine, **XCreateGC** subroutine **XCreatePixmap** subroutine, **XFreeColormap** subroutine, **XFreeColors** subroutine, **XFreePixmap** subroutine, **XGetStandardColormap** subroutine, **XInstallColormap** subroutine, **XListInstalledColormaps** subroutine, **XLookupColor** subroutine, **XQueryColor** subroutine, **XQueryColors** subroutine, **XSetStandardColormap** subroutine, **XSetWindowColormap** subroutine, **XStoreColor** subroutine, **XStoreColors** subroutine, **XStoreNamedColor** subroutine, **XUninstallColormap** subroutine

Enhanced X-Windows Graphics Subroutines Overview

The **Xlib** library graphics subroutines allow you to perform the following tasks:

- Clear and copy areas
- Draw points, lines, rectangles, and arcs
- Fill areas
- Manipulate fonts
- Draw text characters
- Transfer images between clients and server
- Manipulate cursors.

Using Enhanced X-Windows to Clear Areas

Some **Xlib** library subroutines allow you to clear an area or an entire window. The **XClearArea** subroutine clears a specified rectangular area in a window, while the **XClearWindow** subroutine clears the entire window.

Pixmaps do not have defined backgrounds; therefore, pixmaps cannot be filled by the **XClearArea** or **XClearWindow** subroutines. Instead, use the **XFillRectangle** subroutine, which sets the pixmap to a known value.

Using Enhanced X-Windows to Copy Areas

The **XLib** library provides subroutines that you can use to copy an area or a bit plane. When using the **XCopyArea** subroutine to copy a specified area, the source drawable and the destination drawable must have the same root window and the same depth. When using the **XCopyPlane** subroutine to copy a single bit-plane to a drawable with depth >1, the bit plane is color-expanded to the depth of the drawable. In the source drawable, the one (1) bits cause the corresponding pixel in the destination drawable to be set to the foreground specified in the graphics context to alu the function and the planemask. In the same instance, the zero (0) bits in the source drawable cause the corresponding destination pixels to be set to the background color.

Using Enhanced X-Windows to Draw Points, Lines, Rectangles, and Arcs

The **Xlib** library subroutines allow you to draw a single point or multiple points, a single line or multiple lines, a single rectangle or multiple rectangles, and a single arc or multiple arcs.

If the same drawable and **GC** is used for each call, the **Xlib** library batches back-to-back calls to the **XDrawPoint**, **XDrawLine**, **XDrawRectangle**, **XFillArc**, and **XFillRectangle** subroutines.

Using Enhanced X-Windows to Draw Single and Multiple Points

To draw single points, use the **XDrawPoint** subroutine. This subroutine uses the foreground pixel and function components of the **GC** to draw a single point into a drawable. This subroutine is not affected by the tile or stipple in the **GC**.

To draw multiple points, use the **XDrawPoints** subroutine. With this subroutine, specify an array of **XPoint** data structures. The **XDrawPoints** subroutine uses the foreground pixel and function components of the **GC** to draw multiple points into a drawable. This subroutine is not affected by the tile or stipple in the **GC**.

Using Enhanced X-Windows to Draw Single and Multiple Arcs

To draw single and multiple arcs, use the **XDrawArc** subroutine and the **XDrawArcs** subroutine. The **XDrawArc** subroutine draws a single circular or elliptical arc, while the **XDrawArcs** subroutine draws multiple circular or elliptical arcs. Each arc drawn is specified by a rectangle and two angles. The center of the circle or ellipse is the center of the rectangle, and the major and minor axes are specified by the width and height. Positive angles indicate counterclockwise motion, while negative angles indicate clockwise motion. If the magnitude of angle2 is greater than 360 degrees, the **XDrawArc** or **XDrawArcs** subroutine truncates it to 360 degrees.

The x and y coordinates of the rectangle are relative to the origin of the drawable. For an arc specified as `[x,y,w,h,angle1,angle2]`, the origin of the major and minor axes is at `[x+(width/2),y+(height/2)]`.

The infinitely thin path describing the entire circle or ellipse intersects the horizontal axis at `[x,y+(height/2)]` and `[x+width,y+(height/2)]` and the vertical axis at `[x+(width/2),y]` and `[x+(width/2),y+height]`. These coordinates can be fractional. The paths should be defined by the ideal mathematical path.

For a wide line with line-width `lwidth`, the bounding outlines for filling are given by the infinitely thin paths describing the arcs:

```
[x+dx/2, y+dy/2, width-dx, height-dy, angle1, angle2]
              and
[x-lwidth/2, y-lwidth/2, width+lwidth, height+lwidth, angle1,
angle2]
             where
 dx=min(lwidth,width)
 dy=min(lwidth,height)
```

The cap-style and join-style are applied the same as for a line corresponding to the tangent of the circle or ellipse at the endpoint.

For an arc specified as `[x,y,width,height,angle1,angle2]`, the angles must be specified in the effectively skewed coordinate system of the ellipse; for a circle, the angles and coordinate systems are identical. The relationship between these angles and angles expressed in the normal coordinate system of the screen (as measured with a protractor) is as follows:

```
skewed-angle = atan(tan(normal-angle) * width/height) + adjust
```

The skewed-angle and normal-angle are expressed in radians (rather than in degrees scaled by 64) in the range [0,2*PI), and where `atan` returns a value in the range [-PI/2,PI/2], and where adjust is:

```
0       for normal-angle in the range [0,PI/2]
PI      for normal-angle in the range [PI/2,(3*PI)/2)
2*PI    for normal-angle in the range [(3*PI)/2,2*PI)
```

Using Enhanced X-Windows to Fill Areas

To fill single rectanglar, polygon, or arc areas in a specified drawable, use the **XFillRectangle**, **XFillPolygon,** or **XFillArc** subroutine. To fill multiple areas in the specified drawable, use the **XFillRectangles** or **XFillArcs** subroutine. Each subroutine fills the specified area as if the four-point **XFillPolygon** subroutine were specified for each area. The following is an example of the four-point **XFillPolygon** subroutine:

```
[x,y] [x+width, y] [x+width, y+height] [x,y+height]
```

Each subroutine uses x and y coordinates, width and height dimensions, and the **GC** specified.

The **Xlib** library also provides subroutines that can be used to fill the following:

- A single rectangle or multiple rectangles
- A single polygon
- A single arc or multiple arcs

Related Information

The **XPoint** data structure.

The **XClearArea** subroutine, **XClearWindow** subroutine, **XCopyArea** subroutine, **XCopyPlane** subroutine, **XDrawArc** subroutine, **XDrawArcs** subroutine, **XDrawLine** subroutine, **XDrawLines** subroutine, **XDrawPoint** subroutine, **XDrawPoints** subroutine, **XDrawRectangle** subroutine, **XDrawRectangles** subroutine, **XDrawSegments** subroutine, **XFillArc** subroutine, **XFillArcs** subroutine, **XFillPolygon** subroutine, **XFillRectangle** subroutine, **XFillRectangles** subroutine.

Enhanced X-Windows Fonts Overview

The **Xlib** library provides subroutines that enable you to use and manipulate fonts. A font is a graphical family or assortment of characters of a given size or style.

The **Xlib** library provides subroutines that:

- Load and free fonts
- Obtain and free font names
- Set and retrieve the font search path
- Compute character string size
- Return logical extents
- Query character string sizes.

The X Server loads fonts as they are requested by a program. The server can cache fonts for quick lookup. Fonts can be dealt with at several different levels. However, most applications simply use **XLoadQueryFont** to load a font and query the font metrics.

Characters in fonts are regarded as masks. Except for image text requests, the only pixels modified are pixels in which bits are on in the character. This means that it makes sense to draw text using stipples or tiles. For example, many menus gray-out unusable entries.

Using Enhanced X-Windows to Draw Text Characters

To draw 8-bit characters in the specified drawable, the text subroutine **XDrawText** uses the **XTextItem** data structure.

To draw 2-byte characters in the specified drawable, the text subroutine **XDrawText16** uses the **XTextItem16** data structure.

Using Enhanced X-Windows to Transfer Images Between Client and Server

The **Xlib** library subroutines transfer images between a client and the server. Because the server may require diverse data formats, Enhanced X-Windows provides an image object that fully describes the data in memory and provides for basic operations on that data. The data should not be referenced directly, but rather through the image object. Some **Xlib** library implementations can deal with frequently used data formats by replacing routines in the procedure vector with special case routines. These operations include destroying the image, getting a pixel, storing a pixel, extracting a subimage of an image, and adding a constant to an image.

Using Enhanced X-Windows to Manipulate Cursors

Some subroutines load and change cursors associated with windows. Each window can have a different cursor defined for it. Whenever the pointer is in a visible window, it will be set to the cursor defined for that window. If no cursor is defined for that window, the cursor is the cursor defined for the parent window.

For Enhanced X-Windows, a cursor consists of a cursor shape, mask, colors for the shape and mask, and a hot spot. Client programs refer to cursors by using cursor resource IDs.

- The cursor pixmap determines the shape of the cursor.

Enhanced X-Windows provides a set of standard cursor shapes in a special include file named **cursorfont.h**. Use this interface to customize your cursor for individual display. Use the **XCreateFontCursor** subroutine to customize your cursor from this special font, the **XCreateGlyphCursor** subroutine to customize your cursor from another font, and the **XCreatePixmapCursor** subroutine to create a cursor from two bitmaps.

- The mask pixmap determines the bits that are modified by the cursor.
- The colors determines the colors of the shape and mask. The initial colors of a cursor are black foreground and white background. Use the **XRecolorCursor** subroutine to change the color of the cursor.
- The cursor hotspot defines the point on the cursor that is reported when a pointer event occurs. The hotspot in the **XCreatePixmapCursor** subroutine si specified by the x and y coordinate positions.

Hardware can impose limitations on cursor sizes and masks. Enhanced X-Windows supports cursor masks and cursor colors. Use the **XQueryBestCursor** subroutine to find out what size cursors are possible.

Related Information

The **XChar2b** data structure, **XFontStruct** data structure, **XImage** data structure.

The **XAddPixel** subroutine, **XCreateFontCursor** subroutine, **XCreateGlyphCursor** subroutine, **XCreateImage** subroutine, **XDefineCursor** subroutine, **XDestroyImage** subroutine, **XDrawText** subroutine, **XDrawText16** subroutine, **XFreeCursor** subroutine, **XGetFontProperty** subroutine, **XGetImage** subroutine, **XGetPixel** subroutine, **XGetSubImage** subroutine, **XLoadQueryFont** subroutine, **XPutImage** subroutine, **XPutPixel** subroutine, **XQueryBestCursor** subroutine, **XQueryTextExtents** subroutine, **XQueryTextExtents16** subroutine, **XRecolorCursor** subroutine, **XSubImage** subroutine, **XTextItem** Data Structure, **XTextItem16** Data Structure, **XUndefineCursor** subroutine

Enhanced X-Windows Predefined Property Subroutines Overview

There are a number of predefined properties for common information usually associated with windows. The atoms for these properties are in the **<X11/Xatom.h>** file.

The **Xlib** library subroutines perform predefined property operations used in communicating with window managers.

Using Enhanced X-Windows to Communicate with Window Managers

Clients require certain properties and subroutines to communicate with window managers effectively. Some of these properties have complex structures. For example, the data in a single property on the server has to be of the same format (8-bit, 16-bit, or 32-bit), and the structures of property types are not necessarily uniform. Therefore, the **Xlib** library provides Set and Get subroutines for properties with complex structures.

These subroutines define, but do not enforce, minimal policy among window managers. It is encouraged that standard properties be used in creating window managers. However, additional properties may be defined for new window managers.

In addition to Set and Get subroutines for individual properties, the **Xlib** library includes the **XSetStandardProperties** subroutine, which sets all or portions of the WM_NAME, WM_ICON_NAME, WM_NORMAL_HINTS, WM_HINTS, and WM_COMMAND properties. Applications are encouraged to provide the window manager more information than is possible with the **XSetStandardProperties** subroutine. To do so, the Set subroutines for the additional or specific properties required should be used.

To work well with most window managers, an application should specify the name of the application, the name string for the icon, the command used to invoke the application, and the size and window manager hints.

The **Xlib** library does not set defaults for the properties. The window manager determines the defaults which are based on the presence or absence of certain properties. All the properties are considered to be hints to a window manager. When implementing these hints, the window manager decides whether or not to use them.

Predefined properties include the following:

Name	**Type**	**Format**	**Description**
WM_NAME	STRING	8	Application name.
WM_ICON_NAME	STRING	8	Icon name.
WM_NORMAL_HINTS	WM_SIZE_HINTS	32	Size hints for a window in its normal state. (**XSizeHints**)

Name	Type	Format	Description
WM_ZOOM_HINTS	WM_SIZE_HINTS	32	Size hints for a zoomed window. (**XSizeHints**)
Name	**Type**	**Format**	**Description**
WM_HINTS	WM_HINTS	32	Additional hints set by client for the window manager. (**XWMHints**)
WM_COMMAND	STRING	8	The command and arguments, separated by ASCII **NULLS**, used to invoke the application.
WM_ICON_SIZE	WM_ICON_SIZE	32	The window manager can set this property on the root window to specify the icon sizes it supports. (**XIconSize**)
WM_CLASS	STRING	32	Set by application programs to allow window and session managers to obtain the application resources from the resource database.
WM_TRANSIENT_FOR	WINDOW	32	Set by application programs to indicate to the window manager that a transient top-level window, such as a dialog box, is not really a full-fledged window.

The atom names stored in the **<X11/Xatom.h>** file are named **XA_***PROPERTY_NAME.*

The **Xlib** library provides subroutines that can be used to set and get predefined properties. Calling the Set subroutine for a property with complex structure redefines all members in that property, even though only some of those members may have a specified new value. Simple properties for which the **Xlib** library does not provide a Set or Get subroutine can be set using the **XChangeProperty** subroutine. The values for simple properties can be retrieved using the **XGetWindowProperty** subroutine.

Using Enhanced X-Windows to Set and Get Window Names

The **Xlib** library provides subroutines to set and read the name of a window. The **XStoreName** subroutine assigns a name to a window. This window name is usually displayed by the window manager in a titlebar. The **XFetchName** subroutine gets the name of a window. It obtains a window name if the WM_NAME property is set.

Using Enhanced X-Windows to Set and Get Icon Names

The **Xlib** library provides subroutines to set and get the name displayed in the icon of a window. These subroutines set and read the WM_ICON_NAME property. The **XSetIconName** subroutine sets the name displayed in the icon window. The **XGetIconName** subroutine gets the name set in the icon window by the **XSetIconName** subroutine.

Using Enhanced X-Windows to Set and Get Window Manager Hints

The **Xlib** library provides subroutines to set or get window manager hints. All the properties of a window are considered hints to a window manager. The window manager decides whether to implement these hints.

When setting and reading the WM_HINTS property, use the **XWMHints** data structure.

Using Enhanced X-Windows to Set and Get Window Manager Sizing Hints

The **Xlib** library provides subroutines to set or get window sizing hints. All the properties of a window are considered hints to a window manager. The window manager decides whether to implement these hints.

Use the **XSizeHints** data structure to set or get window sizing hints.

Using Enhanced X-Windows to Set and Get Icon Sizing Hints

Applications can cooperate with window managers by providing icons in sizes supported by a window manager. To communicate the supported icon sizes to the applications, a window manager should set the icon size property on the root window. To determine what icon sizes a window manager supports, applications should read the WM_ICON_SIZE property from the root window.

Use the **XIconSize** data structure with the setting and getting icon sizing hints subroutines.

Using Enhanced X-Windows to Set and Get the Class of a Window

The **Xlib** library provides subroutines to set and get the class of a window with the WM_CLASS property.

The name set in the WM_CLASS property may differ from the name set in the WM_NAME property. The WM_NAME property specifies what should be displayed in the title bar and, therefore, can contain temporal information (such as the name of a file currently in the buffer). The WM_CLASS property, however, specifies the formal name of the application that should be used when retrieving the application resources from the resource database.

Use the **XClassHint** data structure to set and get the class of a window.

Using Enhanced X-Windows to Set and Get the Transient Property

An application can indicate to the window manager that a transient top-level window (for example, a dialog box) is operating on behalf of or is transient for another window. To do this, the application sets the WM_TRANSIENT_FOR property of the dialog box to be the window ID of its main window.

Some window managers use this information to unmap dialog boxes of an application (for example, when the main application window is iconified).

Related Information

The **XClassHint** data structure, **XIconSize** data structure, **XSizeHints** data structure, **XWMHints** data structure.

The **<X11/Xatom.h>** header file.

The **XChangeProperty** subroutine, **XFetchName** subroutine, **XGetClassHint** subroutine, **XGetIconName** subroutine, **XGetIconSizes** subroutine, **XGetNormalHints** subroutine, **XGetSizeHints** subroutine, **XGetTransientForHint** subroutine, **XGetWMHints** subroutine, **XGetWindowProperty** subroutine, **XGetZoomHints** subroutine, **XSetClassHint** subroutine, **XSetIconName** subroutine, **XSetIconSizes** subroutine, **XSetNormalHints** subroutine, **XSetSizeHints** subroutine, **XSetStandardProperties** subroutine, **XSetTransientForHint** subroutine, **XSetWMHints** subroutine, **XSetZoomHints** subroutine, **XStoreName** subroutine.

Enhanced X-Windows Resource Manager Overview

The resource manager is a special type of database manager. In most database systems, performing a query using an imprecise specification returns a set of records. The resource manager, however, allows you to specify a large set of values with an imprecise specification, to query the database with a precise specification, and to receive only a single value. The resource manager should be used by applications that need to know what the user prefers for colors, fonts, and other resources.

For example, someone using your application may want to specify that all windows should have a blue background but that all mail-reading windows should have a red background. Presuming that all applications use the resource manager, a user can define this information using only two lines of specification. Your personal resource database usually is stored in a file and is loaded onto a server property when you login. This database is retrieved automatically by the **Xlib** library when a connection is opened.

As an example of how the resource manager works, consider a mail-reading application called **xmh**. Assume that it is designed in such a manner that it uses a complex window hierarchy, all the way down to individual command buttons that can be actual small subwindows. These are often called objects or widgets. These user interface objects can be composed of other objects. Each user interface object can be assigned a name and a class. Fully qualified names or classes can have arbitrary numbers of component names, but a fully qualified name always has the same number of component names as a fully qualified class. This naming convention generally reflects the structure of the application as composed of these objects, starting with the application itself.

For example, the **xmh** mail program has a name **xmh** and is one of a class of *Mail* programs. By convention, the first character of class components is capitalized while the first letter of name components is in lowercase. Each name and class also has an attribute, for example, *foreground* or *font*. If each window is properly assigned a name and a class, it becomes easy for the user to specify attributes of any portion of the application.

At the top level, the application might consist of a paned window (a window divided into several sections) named *toc*. One pane of the window is a button box window named *buttons* filled with command buttons. One of these command buttons is used to retrieve (include) new mail and has the name **include.** This window has a fully qualified name **xmh.toc.buttons.include** and a fully qualified class **Xmh.VPaned.Box.Command**. Its fully qualified names is the name of its parent, **xmh.toc.buttons**, followed by its name, **include**. Its class is the class of its parent window, **Xmh.VPaned.Box**, followed by its particular class, **Command**. The fully qualified name of a resource is the attribute name appended to the fully qualified name of the object, and the fully qualified class is its class appended to the class of the object.

The **include** button requires the following resources:

- Title string
- Font
- Foreground and background color for its inactive state
- Foreground and background color for its active state

Each resource that this button needs is considered an attribute of the button and, as such, has a name and a class. For example, the foreground color for the button in its active state might be **activeForeground** and the class could be **Foreground.**

When an application searches for a resource (for example, a color), it passes the complete name and class of the resource to a lookup routine. Then, the resource manager returns the resource value and the representation type.

The resource manager allows applications to store resources by an incomplete specification of name, class, and representation type, as well as to retrieve them given a fully qualified name and class.

Using Enhanced X-Windows Resource Manager

The definitions for the use of the resource manager are contained in the **<X11/Xresource.h>** header file. The **Xlib** library also uses the resource manager internally to allow for non-English language error messages.

Database values consist of a size, an address, and a representation type. The size is specified in bytes. The representation type allows storage of data tagged by some application-defined type (for example, *Font* or *Color*). It has nothing to do with the C language data type or with its class.

Most uses of the resource manager involve defining names, classes, and representation types as string constants. However, always referring to strings in the resource manager can be slow, because it is used so heavily by some toolkits. To solve this problem, a shorthand name for a string is used in place of the full name of the string in many of the resource manager subroutines. Simple comparisons can be performed rather than string comparisons. The short name for a string is quark. The quark type is **XrmQuark**. You may want to allocate a quark that has no string equivalent on some occasions. (A quark is to a string what an atom is to a property name in the server, but the use of a quark is local to your application.)

Each name, class, and representation type is defined as an **XrmQuark**:

```
typedef int XrmQuark, *XrmQuarkList;
typedef XrmQuark XrmName;
typedef XrmQuark XrmClass;
typedef XrmQuark XrmRepresentation;
```

Lists are represented as null-terminated arrays of quarks. The size of the array must be large enough for the number of components used.

```
typedef XrmQuarkList XrmNameList;
typedef XrmQuarkList XrmClassList;
```

Using Enhanced X-Windows to List Resource Manager Matching Rules

The algorithm for determining which resource name matches a given query is the heart of the database. Resources are stored with only partially specified names and classes, using pattern matching constructs. An asterisk (*) is used to represent any number of intervening components, including none. A period (.) is used to separate immediately adjacent components. All queries fully specify the name and class of the resource needed. A trailing asterisk or period is not removed. The library supports 100 components in a name or a class. The lookup algorithm searches the database for the name that most closely matches (is most specific) to this full name and class. The rules in order of precedence for a match are:

1. The attribute of the name and class must match. For example, queries for

   ```
   aixterm.scrollbar.background     (name)
   AIXTerm.Scrollbar.Background     (class)
   ```

   will not match the database entry

   ```
   aixterm.scrollbar:on
   ```

2. Database entries with a name or class prefixed by a period (.) are more specific than those prefixed by an asterisk (*). For example, `aixterm.geometry` is more specific than `aixterm*geometry`.
3. Names are more specific than classes. For example, `*scrollbar.background` is more specific than `*Scrollbar.Background`.
4. A name or class is more specific than an omission. For example, `Scrollbar*Background` is more specific than `*Background`.
5. Left components are more specific than right components. For example, `*vt100*background` is more specific than `*scrollbar*background` for the query `.vt100.scrollbar.background`.
6. If neither a period (.) nor an asterisk (*) is specified at the beginning, a period is implicit. For example, `aixterm.background` is identical to `.aixterm.background`.

Names and classes can be mixed. As an example of these rules, assume the following user specification:

```
xmh*background:                     red
*command.font:                      8x13
*command.background:                blue
*Command.Foreground:                green
xmh.toc*Command.activeForeground:   black
```

A query for the name

```
xmh.toc.messagefunctions.include.activeForeground
```

and class

```
Xmh.VPaned.Box.Command.Foreground
```

would match

```
xmh.toc*Command.activeForeground
```

and return `black`. However, it also matches

```
*Command.Foreground.
```

Using the precedence algorithm described above, the resource manager would return the value specified by `xmh.toc*Command.activeForeground`.

Using Enhanced X-Windows to Store Information Into a Resource Database

The **Xlib** library provides subroutines to store resources into the database. The **XrmPutResource** and **XrmQPutResource** subroutines take a partial resource specification, a representation type, and a value. This value is copied into the specified database. To add a resource that is specified as a string, use the **XrmPutResource** subroutine. To add a string resource using quarks as a specification, use the **XrmQPutResource** subroutine. To add a resource entry that is specified as a string that contains both a name and a value, use the **XrmPutLineResource** subroutine.

Using Enhanced X-Windows to Retrieve Information from a Resource Database

The **Xlib** library provides subroutine to retrieve a resource from a specified resource database. The **XrmGetResource** and **XrmQGetResource** subroutines take a fully qualified name and class pair, a destination resource representation, and the address of a value (size and address pair). The value and the returned type point into database memory that should not be modified.

The database only frees or overwrites entries on the **XrmPutResource**, **XrmQPutResource**, or **XrmMergeDatabases** subroutines. A client that is not storing new values and is not merging the database should be safe using the address passed back at any time until it exits. If a resource is found, the **XrmGetResource** and **XrmQGetResource** subroutines return the value of **True**.

The **Xlib** library also provides subroutines to search and return lists of resource database levels, as well as to combine, retrieve and store databases.

Most applications and toolkits do not make random probes into a resource database to fetch resources. The X Toolkit access pattern for a resource database is quite stylized. A series of probes (from 1 to 20 in number) are made with only the last name-class pair differing in each probe. The **XrmGetResource** subroutine is at worst a 2^{**n} algorithm, where *n* is the length of the name-class pair list. This can be improved upon by the application programmer by prefetching a list of database levels that might match the first part of a name-class pair list.

Using Enhanced X-Windows to Parse Command Line Options

The **Xlib** library provides a subroutine to load a resource database from a C language command line. The **XrmParseCommand** subroutine can be used to parse the command line arguments to a program and modify a resource database with selected entries from the command line.

Related Information

The **XrmOptionDescList** data structure, **XrmValue** data structure.

The **<X11/Xresource.h>** header file.

The **Xpermalloc** subroutine, **XrmGetFileDatabase** subroutine, **XrmGetResource** subroutine, **XrmGetStringDatabase** subroutine, **XrmMergeDatabases** subroutine, **XrmParseCommand** subroutine, **XrmInitialize** subroutine, **XrmPutFileDatabase** subroutine, **XrmPutLineResource** subroutine, **XrmPutResource** subroutine, **XrmPutStringResource** subroutine, **XrmQGetResource** subroutine, **XrmQGetSearchList** subroutine, **XrmQGetSearchResource** subroutine, **XrmQPutResource** subroutine, **XrmQPutStringResource** subroutine, **XrmQuarkToString** subroutine, **XrmStringToBindingQuarkList** subroutine, **XrmStringToQuark** subroutine, **XrmStringToQuarkList** subroutine, **XrmUniqueQuark** subroutine.

Enhanced X-Windows Context Manager Overview

The context manager provides a way of associating data with a window in your program. The context manager is local to your program; the data is not stored in the server on a property list. Any data in any number of pieces can be associated with a window, and each piece of data has a type associated with it. The context manager requires the window ID and type to store or retrieve data.

Essentially, the context manager can be viewed as a two-dimensional, sparse array. One dimension is subscripted by the window ID and the other by a context type field. Each entry in the array contains a pointer to the data.

The **Xlib** library provides context management subroutines to save and get data values, delete entries, and create a unique context type. The symbols used are in the **<X11/Xutil.h>** header file.

Related Information

The **<X11/Xutil.h>** header file.

The **XDeleteContext** subroutine, **XFindContext** subroutine, **XSaveContext** subroutine, **XUniqueContext** subroutine.

Enhanced X-Windows Window Information Subroutines Overview

After the display is connected to the X Server and a window is created, use **Xlib** window information subroutines to perform the following tasks:

- Obtain information about a window
- Manipulate property lists
- Obtain and change window properties
- Manipulate window selection.

Defining Enhanced X-Windows Coordinates

Like the display screen, each window has its own xy coordinate system with the origin at the upper-left hand corner. The x axis is horizontal (left to right); the y axis is vertical (top to bottom). Each addressable point on the screen is called a pixel and corresponds to one xy point in the coordinate system. Because each window has its own coordinate system, operations within windows can be insensitive to the window position on the screen.

The origin of a window is inside the border, if it has one, and window size is always the usable number of pixels within the border. The window border is maintained by the X Server, and output to the window is clipped so as not to extend into the border.

Obtaining Enhanced X-Windows Window Information

Xlib subroutines obtain information about the window tree, the current attributes of a window, its current geometry, or the current pointer coordinates. Because these subroutines are used primarily by window managers, they return a status to indicate if the window still exists.

The **XGetWindowAttributes** subroutine returns the current attributes for the specified window to an **XWindowAttributes** structure.

Obtaining Enhanced X-Windows Environment Defaults

A program often needs a variety of options in the X environment (for example, fonts, colors, mouse, background, text, and cursor). Specifying these options to run on the command line is inefficient and unmanageable because individual users have different tastes with regard to window appearance. The **XGetDefault** subroutine facilitates finding the default fonts, colors, and other environment options favored by a particular user.

Defaults are usually loaded into the resource manager property on the root window during login. The **XGetDefault** subroutine merges additional defaults from a file in the user's home directory. The **XGetDefault** subroutine provides a simple interface for clients.

Using Enhanced X-Windows Properties and Atoms

A property is a collection of names, typed data. The window system has a a set of predefined properties, such as the name of a window and size hints.

Each property has a name which is an ISO Latin–1 string. For each named property, a unique identifier (atom) is associated with it. Users can define any other arbitrary information and can associate this information with a window. (Use **XInternAtom** to define new properties or to obtain the atom for new properties.)

A property also has a type, for example, a string or an integer. These types are also indicated using atoms. Therefore, arbitrary new types can be defined. The type of a property is defined by other properties, which allows for arbitrary extension. Use the properties mechanism to communicate other information between applications.

A property is stored in one of several possible formats. The X Server can store the information as 8-bit, 16-bit, or 32-bit quantities. This flexibility permits the X Server to present the data in the byte order that the client expects.

Certain properties are predefined in the server for commonly used subroutines. The atoms for these properties are defined in **<X11/Xatom.h>**. To avoid name clashes with user symbols, the macro name for each atom has the **XA_** prefix added.

Atoms occur in five distinct name spaces within the protocol: selections, property names, property types, font properties and types of **ClientMessage** events (none are built into the X Server).

Any particular atom can have some client interpretation within each of the name spaces. The built-in selection properties, which name properties, are **PRIMARY** and **SECONDARY.**

The built-in property names are:

CUT_BUFFER0	RGB_GREEN_MAP	WM_CLIENT_MACHINE
CUT_BUFFER1	RGB_RED_MAP	WM_COMMAND
CUT_BUFFER2	RGB_BEST_MAP	WM_HINTS
CUT_BUFFER3	RGB_BLUE_MAP	WM_ICON_NAME
CUT_BUFFER4	RGB_DEFAULT_MAP	WM_ICON_SIZE
CUT_BUFFER5	RGB_GRAY_MAP	WM_NAME
CUT_BUFFER6	RESOURCE_MANAGER	WM_NORMAL_HINTS
CUT_BUFFER7	WM_CLASS	WM_ZOOM_HINTS
	WM_TRANSIENT_FOR	

The built-in property types are:

ARC	DRAWABLE	RGB_COLOR_MAP
ATOM	FONT	STRING
BITMAP	INTEGER	VISUALID
CARDINAL	PIXMAP	WINDOW
COLORMAP	POINT	WM_HINTS
CURSOR	RECTANGLE	WM_SIZE_HINTS

The built-in font property types are:

MIN_SPACE	UNDERLINE_POSITION	QUAD_WIDTH
NORM_SPACE	UNDERLINE_THICKNESS	WEIGHT
MAX_SPACE	FONT_NAME	POINT_SIZE
END_SPACE	FULL_NAME	RESOLUTION
SUPERSCRIPT_X	STRIKEOUT_DESCENT	COPYRIGHT
SUPERSCRIPT_Y	STRIKEOUT_ASCENT	NOTICE
SUBSCRIPT_X	ITALIC_ANGLE	FAMILY_NAME
SUBSCRIPT_Y	X_HEIGHT	CAP_HEIGHT

Obtaining and Changing Enhanced X-Windows Window Properties

You can attach a property list to every window. Each property has a name, a type, and a value. The value is an array of 8-bit, 16-bit, or 32-bit quantities, whose interpretation is left to the clients.

You can obtain, rotate, or change a window property. In addition, the **Xlib** library provides subroutines for predefined property operations.

Using Enhanced X-Windows Window Selections

Selection is one method of exchanging data between applications. By using the property mechanism, applications can exchange data of arbitrary types and can negotiate the type of data to be exchanged. A selection can be thought of as an indirect property with a dynamic type. Rather than having the property stored in the X Server, the property is maintained by some client (the owner). A selection is global in nature. It belongs to the user but is maintained by the clients, rather than being private to a particular window subhierarchy or a particular set of clients.

The **Xlib** library subroutines set, get, or convert window selection. These subroutines allow applications to implement the notion of current selection, which requires that applications be notified when they no longer own the selection. Applications that support selection often highlight the current selection. To unhighlight the selection, applications must be able to be informed when some other application acquires the selection.

When a client asks for the contents of a selection, it specifies a selection target type. This target type can be used to control the transmitted representation of the contents. For example, if the selection is "the last thing the user clicked on", and currently an image, then the target type might specify whether the contents of the image should be sent in **XYFormat** or **ZFormat**.

The target type can also be used to control the class of contents transmitted, for example, asking for the page format (fonts, line spacing, indentation, and other page format specifications) of a paragraph selection, not the text of the paragraph. The target type can also be used for other purposes. The semantics are not constrained by the protocol.

Related Information

The **XWindowAttributes** data structure.

The **XChangeProperty** subroutine, **XConvertSelection** subroutine, **XDeleteProperty** subroutine, **XGetAtomName** subroutine, **XGetDefault** subroutine, **XGetGeometry** subroutine, **XGetSelectionOwner** subroutine, **XGetWindowAttributes** subroutine, **XGetWindowProperty** subroutine, **XInternAtom** subroutine, **XListProperties** subroutine, **XQueryPointer** subroutine, **XQueryTree** subroutine, **XRotateWindowProperties** subroutine, **XSetSelectionOwner** subroutine.

Enhanced X-Windows Window Manager Subroutines Overview

The **Xlib** library window manager subroutines allow you to create a window manager application that can:

- Control the lifetime of a window
- Manipulate the pointer
- Manipulate keyboard settings and the keyboard encoding
- Control host access

Using Enhanced X-Windows to Control the Lifetime of a Window

The save-set of a window manager is a list of other client windows that should not be destroyed if the client windows are inferior windows of the window manager windows at connection close. To allow an application window to survive when a window manager fails, the **Xlib** library provides the save-set subroutines that change a window manager save-set, add a window to a window manager save-set, or remove a subwindow from a window manager save-set.

Some subroutines are used to control the longevity of subwindows that are normally destroyed when the parent is destroyed. For example, a window manager that wants to add decoration to a window by adding a frame might reparent an application window. When the frame is destroyed, the application window should not be destroyed but returned to its previous place in the window hierarchy.

The X Server automatically removes windows from the save-set when they are destroyed. If a save-set window is an inferior of a window manager window, the save-set window is reparented to the closest ancestor so that the save-set window is not an inferior window of a window created by the window manager. If the save-set window is unmapped, a **MapWindow** request is performed on it. After the save-set list is processed, all windows created by the window manager are destroyed. For each nonwindow resource created by the window manager, the appropriate **Free** request is performed. All colors and colormap entries allocated by the window manager are freed.

Using Enhanced X-Windows to Grab the Pointer

The **Xlib** library subroutines control input from the pointer, which is usually a mouse. The window manager uses these facilities to implement certain styles of user interface. Some applications may use these facilities for special purposes.

Usually, the X Server delivers keyboard and mouse events as soon as they occur to the appropriate client, depending upon the window and input focus. The X Server provides sufficient control over event delivery to allow window managers to support various styles of user interface. Many of the styles 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 are sent to the grabbing client rather than the normal client. If the keyboard or pointer is in asynchronous mode, further mouse and keyboard events continue being processed. If the keyboard or pointer is in synchronous mode, no further events are processed until the grabbing client allows them to be processed. The keyboard or pointer is considered frozen during this interval. The triggering event can also be replayed.

There are two kinds of grabs: an active grab and a passive grab.

Active grab	This occurs when a single client grabs the keyboard or pointer explicitly.
Passive grab	This occurs when clients grab a particular keyboard key or pointer button in a window. Passive grabs are convenient for implementing reliable pop-up menus.

The grab activates when the key or button is actually pressed. For example, you can arrange that the pop-up is mapped before the *up* pointer button event occurs by grabbing a button requesting synchronous behavior. The *down* event triggers the grab and freezes 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 is then correctly processed relative to the pop-up window.

For many operations, there are subroutines that take a *time* argument. The X Server includes a timestamp 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 and the time of the server when the client last performed an active grab, when the keyboard was last grabbed, or when a selection was last changed. This allows you to specify that your request not occur if some other application has in the meantime taken control of the keyboard, pointer, or selection.

Using Enhanced X-Windows to Grab the Server

Some **Xlib** library subroutines grab and ungrab the server. These suroutines can be used to control processing of output on other connections by the window system server. No processing of requests or close downs on any other connection occurs while the server is grabbed. If a client closes its connection to the server, the client automatically ungrabs the server. Grabbing the server is highly discouraged unless it is absolutely necessary.

Using Enhanced X-Windows to Manipulate Keyboard Settings

With **Xlib** library subroutines, you can 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.

You can set many of these options to your preference. The default values for many of these subroutines are determined by command line arguments to the X Server. Not all implementations can actually control all of these parameters.

The **XKeyboardControl** data structure is used in subroutines that change control from a keyboard.

The **XKeyboardState** data structure is used in subroutines that return the current control values for the keyboard.

Using Enhanced X-Windows to Manipulate 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. 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 include:

- the ISO Latin character sets (1-4)
- Katakana
- Arabic
- Cyrillic
- Greek
- Technical
- Special
- Publishing
- APL
- Hebrew
- a special miscellany of keys found on keyboards (such as **RETURN**, **HELP**, and **TAB**)

To the extent possible, these sets are derived from international standards. The list of defined symbols is in the **<X11/keysymdef.h>** header file. If you must use Keysyms not in the ISO Latin 1-4, Greek, and miscellany classes, you may have to define a symbol for those sets. Most applications usually include only **<X11/keysym.h>**, which defines symbols for ISO Latin 1-4, Greek, and miscellany.

A list of Keysyms is associated with each Keycode. The length of the list can vary with each Keycode. The list is intended to convey the set of symbols on the corresponding key. By convention, if the list contains a single Keysym and if that Keysym is alphabetic and case distinction is relevant for it, then it should be treated as equivalent to a two-element list of the lowercase and uppercase Keysyms. For example, if the list contains the single Keysym for uppercase A, the client should treat it as if it were a pair with lowercase a as the first Keysym and uppercase A as the second Keysym.

For any Keycode, the first Keysym in the list should be chosen as the interpretation of a KeyPress when no modifier keys are down.

The second Keysym in the list normally should be chosen when the Shift modifier is on, or when the Lock modifier is on and Lock is interpreted as ShiftLock.

When the Lock modifier is on and is interpreted as CapsLock, it is suggested that the Shift modifier first be applied to choose a Keysym, but if that Keysym is lowercase alphabetic, the corresponding uppercase Keysym should be used instead.

Other interpretations of CapsLock are possible; for example, it may be viewed as equivalent to ShiftLock, but only applying when the first Keysym is lowercase alphabetic and the second Keysym is the corresponding uppercase alphabetic. No interpretation of Keysyms beyond the first two in a list is suggested here. 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 the mapping for reading and writing by clients.

The **XLookupString** subroutine performs simple translation of a key event to an ASCII string.

The **XModifierKeymap** data structure is used by subroutines that modify the keyboard mapping.

Using Enhanced X-Windows to Control Host Access

Enhanced X-Windows does not provide any protection on a per-window basis. If you find out the ID of a resource, you can manipulate it. To provide some minimal level of protection, however, connections are permitted only from systems you trust. This is adequate on single-user workstations, but breaks down on time-sharing machines.

The initial set of hosts allowed to open connections consists of:

- The host the window system is running on.
- On AIX-based systems, the hosts listed in the **/etc/X?.hosts** file. The "?" indicates the number of the display. This file should consist of host names separated by newlines.

If a host is not in the access control list when the access control mechanism is enabled and if the host attempts to establish a connection, the server refuses the connection. To change the access list, the client must reside on the same host as the server and/or must have been granted permission in the initial authorization at connection setup. The initial access control list can be specified by providing a file that the server can read at startup and reset time.

To add, get, or remove hosts, the host access control subroutines use the **XHostAddress** data structure.

Related Information

The **<X11/keysymdef.h>** header file, **<X11/keysym.h>** header file.

The **/etc/X?.hosts** file.

The **XHostAddress** data structure, **XKeyboardControl** data structure, **XKeyboardState** data structure, **XModifierKeymap** data structure.

The **XAddHost** subroutine, **XAddHosts** subroutine, **XAddToSaveSet** subroutine, **XAllowEvents** subroutine, **XAutoRepeatOn** subroutine, **XBell** subroutine,

XChangeActivePointerGrab subroutine, **XChangeKeyboardControl** subroutine, **XChangeKeyboardMapping** subroutine, **XChangeSaveSet** subroutine, **XDeleteModifiermapEntry** subroutine, **XFreeModifiermap** subroutine, **XGetKeyboardControl** subroutine, **XGetKeyboardMapping** subroutine, **XGetModifierMapping** subroutine, **XGetPointerMapping** subroutine, **XGrabButton** subroutine, **XGrabKey** subroutine, **XGrabKeyboard** subroutine, **XGrabPointer** subroutine, **XGrabServer** subroutine, **XInsertmodifiermapEntry** subroutine, **XListHosts** subroutine, **XLookupString** subroutine, **XNewModifiermap** subroutine, **XQueryKeymap** subroutine, **XRemoveFromSaveSet** subroutine, **XRemoveHost** subroutine, **XRemoveHosts** subroutine, **XSetModifierMapping** subroutine, **XSetPointerMapping** subroutine, **XUngrabButton** subroutine, **XUngrabkey** subroutine, **XUngrabKeyboard** subroutine, **XUngrabPointer** subroutine, **XUngrabServer** subroutine.

Enhanced X-Windows Window Subroutines Overview

In Enhanced X-Windows, a window is a rectangular area on the screen that lets you view 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. Use the **XOpenDisplay** subroutine to open a display.

Defining Enhanced X-Windows Visual Types

On some high-end displays, color resources can be used in several ways. For example, you can use the display as a 12-bit display with arbitrary mapping of pixel to color (pseudo-color) or as a 24-bit display with 8 bits of the pixel dedicated for red, green, and blue. These different ways of using visual aspects are called Visuals. For example,

- The screen can be color or grayscale.
- The screen can have a colormap that is writable or read-only.
- A screen can also have a colormap whose indices are decomposed into separate RGB pieces, provided one is not on a grayscale screen. The following table shows the color or grayscale of a colormap:

	Color		Gray Scale	
	Read-Only	Read–Write	Read-Only	Read–Write
Undecomposed Colormap	Static Color	Pseudo Color	Static Gray	Gray Scale
Decomposed Colormap	True Color	Direct Color		

Determining the Appropriate Enhanced X-Windows Visual

For each screen, a list of valid visual types can be supported at different depths of the display. The **Xlib** library uses the **XVisualInfo** data structure to determine which visual to use in the application.

Defining Enhanced X-Windows Window Attributes

All **InputOutput** windows have a border width of zero or more pixels, an optional background, an input mask, an event suppression mask, and a property list. The window border and background can be a solid color or a pattern, which is called a tile. All windows, except the root window, have a parent window and are clipped by the parent window.

- If a window is stacked on top of another window, the top window obscures the lower window, not allowing it to accept input. Input events, such as pointer motion events, are not generated for the obscured area of the lower window.
- If the top window has a background, the top window obscures the lower window, not allowing it to have output. Output attempts to the obscured area of the lower window produce no result.

Windows with a class of **InputOnly** are used to control input events in situations where full-fledged windows are not necessary. **InputOnly** windows are used to control input events in situations where **InputOutput** windows are not necessary. **InputOnly** windows are invisible and can only be used to control such things as cursors, input event generation and grabbing. They cannot be used for any graphics events. **InputOnly** windows cannot have **InputOutput** windows as inferiors. Both **InputOnly** and **InputOutput** windows have the following attributes:

- *win_gravity*
- *event_mask*
- *do_not_propagate_mask*
- *override_redirect*
- *cursor*

If other fields are defined for an **InputOnly** window, a **BadMatch** error is generated.

Windows have borders of a programmable width and pattern, as well as a background pattern or tile. Pixels can be used for solid colors. Refer to the window in a program by using the resource ID of type **Window**. The background and border pixmaps can be destroyed immediately after creating the window if no further explicit references are made.The background of a window can be a solid color or a pattern.

- If the pattern is relative to the parent window, the pattern is shifted appropriately to match the parent window.
- If the pattern is absolute, it is positioned in the window independently of the parent window.

When an application first creates a window, it is not visible (unmapped). Unmapped windows are invisible, and output to an unmapped window is discarded. When a window is eventually mapped to the screen with the **XMapWindow** subroutine, the X Server generates an exposure event for the window if backing store has not been maintained.

A window manager can override the size, border width, and style of a window specified in your program. You should be prepared to use the actual size and position of the top window, which is reported as an event when the window is first mapped. A client program should not resize itself without command input. Instead, the client program should use the space specified. If the space specified is insufficient, the client program can request a resize of the window. Only the border of the top-level windows can be changed by window managers.

Creating Enhanced X-Windows Windows

The **Xlib** library provides basic ways for creating windows. When you create top-level windows or direct children of the root window, the following should be observed to ensure that applications interact properly across differing styles of window management.

- Allow the window manager to determine the size or placement of your top-level windows. Provide the window manager with some standard information or hints with the various window manager hints subroutines.
- An application, by interpreting the first exposure event, must be able to deal with whatever size window it gets, even if this means that the application just prints a message, such as "Please make me bigger", in the window.
- An application should be able to resize or move the children of its top-level window as necessary. An application should only resize or move its top-level window in direct response to a user request. Otherwise, the request may fail.
- Applications should not be written to assume control of the window manager.
- The application should set standard window properties for the top-level window before mapping it. To set standard window properties for a top-level window, use the **XSetStandardProperties** subroutine.

Use the **XCreateWindow** and **XCreateSimpleWindow** subroutines to create an unmapped subwindow for a specified parent window.

- The **XCreateWindow** subroutine allows you to set specific window attributes when you create the window.
- The **XCreateSimpleWindow** subroutine creates a window that inherits its window attributes from the parent window.

InputOnly windows cannot be used for graphics requests, exposure processing, and **VisibilityNotify** events. An **InputOnly** window cannot be used as a source or destination drawable for graphics requests.

Destroying Enhanced X-Windows Windows

Enhanced X-Windows. also allows you to destroy windows and subwindows. You can destroy a window or destroy all subwindows of a window. If a window is destroyed, it should not be referenced by another subroutine. If you specify that a root window be destroyed, no windows are actually destroyed. By default, windows are destroyed when a connection to the X Server is closed.

Mapping Enhanced X-Windows Windows

A window is considered mapped if an **XMapWindow** call is made on the window. The window may not be visible on the screen for one of the following reasons:

- The window is hidden by another opaque sibling window.
- One of the window ancestors is not mapped.
- The window is entirely clipped by an ancestor.

A window manager can control the placement of subwindows. If **SubstructureRedirectMask** has been selected by a window manager on a parent window (usually a root window), a map request initiated by other clients on a child window is not performed, and the window manager is sent a **MapRequest** event.

If the *override_redirect* field on the child is set to **True** (usually only on pop-up menus), the map request is performed.

A tiling window manager can reposition and resize other client windows and then map the window at its final location. Only one client at a time can select **SubstructureRedirectMask**.

Similarly, a single client can select **ResizeRedirectMask** on a parent window. Any attempt to resize the window is suppressed, and the client, which is usually a window manager, receives a **ResizeRequest** event. These mechanisms allow arbitrary placement policy to be enforced by an external window manager.

Exposure events are generated for the window when part or all of it becomes visible on the screen. A client will only receive the exposure events if it requests these events with **XSelectInput**. Windows retain their position in the stacking order when unmapped.

Configuring Enhanced X-Windows Windows

The **Xlib** library subroutines can move a window, resize a window, move and resize a window, or change the border width of a window. The most general interface to configuring windows, the **XConfigureWindow** subroutine, uses the **XWindowChanges** data structure.

Related Information

The **XSetWindowAttributes** data structure, **XVisualInfo** data structure, **XWindowChanges** data structure.

The **XCirculateSubwindows** subroutine, **XCirculateSubwindowsDown** subroutine, **XCirculateSubwindowsUp** Subroutine, **XConfigureWindow** subroutine, **XCreateSimpleWindow** subroutine, **XCreateWindow** subroutine, **XDestroySubwindows** subroutine, **XDestroyWindow** subroutine, **XGetVisualInfo** subroutine, **XLowerWindow** subroutine, **XMapRaised** subroutine, **XMapSubwindows** subroutine, **XMapWindow** subroutine, **XMatchVisualInfo** subroutine, **XMoveWindow** subroutine, **XMoveResizeWindow** subroutine, **XRaiseWindow** subroutine, **XResizeWindow** subroutine, **XRestackWindows** subroutine, **XSetStandardProperties** subroutine, **XSetWindowBorderWidth** subroutine, **XUnmapSubwindows** subroutine, **XUnmapWindow** subroutine, **XVisualIDFromVisual** subroutine.

List of Enhanced X-Windows Xlib Data Structures

The **XAIXDeviceMappingEvent** data structure
The **XVisualInfo** data structure
The **XSetWindowAttributes** data structure
The **XWindowChanges** data structure
The **XWindowAttributes** data structure
The **XColor** data structure
The **XGCValues** data structure
The **XStandardColormap** data structure
The **XSegment** data structure
The **XRectangle** data structure
The **XPoint** data structure
The **XArc** data structure
The **XCharStruct** data structure
The **XFontProp** data structure
The **XChar2b** data structure
The **XFontStruct** data structure
The **XTextItem** data structure
The **XTextItem16** data structure
The **XImage** data structure
The **XKeyboardControl** data structure
The **XKeyboardState** data structure
The **XModifierKeymap** data structure
The **XHostAddress** data structure
The **XAnyEvent** data structure
The **XEvent** data structure
The **XButtonPressedEvent** data structure
The **XButtonReleasedEvent** data structure
The **XKeyPressedEvent** data structure
The **XKeyReleasedEvent** data structure
The **XPointerMovedEvent** data structure
The **XCrossingEvent** data structure
The **XEnterWindowEvent** data structure
The **XLeaveWindowEvent** data structure
The **XFocusInEvent** data structure
The **XFocusOutEvent** data structure
The **XKeymapEvent** data structure
The **XExposeEvent** data structure
The **XGraphicsExposeEvent** data structure
The **XNoExposeEvent** data structure
The **XCirculateEvent** data structure
The **XConfigureEvent** data structure
The **XCreateWindowEvent** data structure
The **XDestroyWindowEvent** data structure
The **XGravityEvent** data structure
The **XMapEvent** data structure
The **XMappingEvent** data structure
The **XReparentEvent** data structure
The **XUnmapEvent** data structure
The **XVisibilityEvent** data structure
The **XCirculateRequestEvent** data structure
The **XConfigureRequestEvent** data structure

The **XMapRequestEvent** data structure
The **XResizeRequestEvent** data structure
The **XColormapEvent** data structure
The **XClientMessageEvent** data structure
The **XPropertyEvent** data structure
The **XSelectionClearEvent** data structure
The **XSelectionRequestEvent** data structure
The **XSelectionEvent** data structure
The **XErrorEvent** data structure
The **XWMHints** data structure
The **XSizeHints** data structure
The **XIconSize** data structure
The **XClassHint** data structure
The **XrmValue** data structure
The **XrmOptionDescList** data structure

XVisualInfo Data Structure

```
#define  VisualNoMask                 0x0
#define  VisualIDMask                 0x1
#define  VisualScreenMask             0x2
#define  VisualDepthMask              0x4
#define  VisualClassMask              0x8
#define  VisualRedMaskMask            0x10
#define  VisualGreenMaskMask          0x20
#define  VisualBlueMaskMask           0x40
#define  VisualColormapSizeMask       0x80
#define  VisualBitsPerRGBMask         0x100
#define  VisualAllMask                0x1FF

typedef struct {
     Visual *visual;
     VisualID visualid;
     int screen;
     unsigned int depth;
     int class;
     unsigned long red_mask;
     unsigned long green_mask;
     unsigned long blue_mask;
     int colormap_size;
     int bits_per_rgb;
} XVisualInfo;
```

The fields of the **XVisualInfo** data structure are as follows:

bits_per_rgb	Specifies the log base 2 of the approximate number of distinct color values (individually) of red, green, and blue (RGB). Actual RGB values are unsigned 16-bit numbers.
blue_mask	Defined only for **DirectColor** and **TrueColor**. Each mask has one contiguous set of bits with no intersections.
class	Specifies the possible visual classes of the screen. It can be one of the following: **PseudoColor**, **GrayScale**, **DirectColor**, **TrueColor**,

StaticColor, or **StaticGray**. Conceptually, as each pixel is read out of video memory, it goes through a lookup stage by indexing into a colormap. Colormaps can be manipulated arbitrarily on some hardware, in a limited way on other hardware, and not at all on other hardware. The visual types affect the colormap and the RGB values in the following ways:

PseudoColor	A pixel value indexes a colormap to produce independent RGB values, and the RGB values can be changed dynamically.
GrayScale	A pixel value indexes a colormap to produce independent RGB values, and the RGB values can be changed dynamically, except that the primary that drives the screen is not defined. Therefore, the client should always store the same value for red, green, and blue in the colormaps.
DirectColor	A pixel value is decomposed into separate RGB subfields, and each subfield separately indexes the colormap for the corresponding value. The RGB values can be changed dynamically.
TrueColor	A pixel value is decomposed into separate RGB subfields, except that the colormap has predefined read-only RGB values. These values are server-dependent, but provide linear or near-linear ramps in each primary.
StaticColor	A pixel value indexes a colormap to produce independent RGB values, and the RGB values can be changed dynamically, except that the colormap has predefined read-only server-dependent RGB values.
StaticGray	A pixel value indexes a colormap to produce independent RGB values, and the RGB values can be changed dynamically, except that the red, green, and blue values are equal for any single pixel value that results in shades of gray. **StaticGray** with a two-entry colormap can be considered monochrome.

colormap_size	Defines the number of available colormap entries in a newly created colormap. For **DirectColor** and **TrueColor**, this number is the size of an individual pixel subfield.
depth	Specifies the depth of the screen.
green_mask	Defined only for **DirectColor** and **TrueColor**. Each mask has one contiguous set of bits with no intersections.
red_mask	Defined only for **DirectColor** and **TrueColor**. Each mask has one contiguous set of bits with no intersections.
screen	Specifies the screen.

visual	Specifies the visual.
visualid	Specifies the visual ID.

Related Information

The **XGetVisualInfo** subroutine, **XMatchVisualInfo** subroutine, **XVisualIDFromVisual** subroutine

XSetWindowAttributes Data Structure

```
#define  CWBackPixmap        (1L<<0)
#define  CWBackPixel         (1L<<1)
#define  CWBorderPixmap      (1L<<2)
#define  CWBorderPixel       (1L<<3)
#define  CWBitGravity        (1L<<4)
#define  CWWinGravity        (1L<<5)
#define  CWBackingStore      (1L<<6)
#define  CWBackingPlanes     (1L<<7)
#define  CWBackingPixel      (1L<<8)
#define  CWOverrideRedirect  (1L<<9)
#define  CWSaveUnder         (1L<<10)
#define  CWEventMask         (1L<<11)
#define  CWDontPropagate     (1L<<12)
#define  CWColormap          (1L<<13)
#define  CWCursor            (1L<<14)

typedef struct {
   Pixmap background_pixmap;
   unsigned long background_pixel;
   Pixmap border_pixmap;
   unsigned long border_pixel;
   int bit_gravity;
   int window_gravity;
   int backing_store;
   unsigned long backing_planes;
   unsigned long backing_pixel;
   Bool save_under;
   long event_mask;
   long do_not_propagate_mask;
   Bool override_redirect;
   Colormap colormap;
   Cursor cursor;
} XSetWindowAttributes;
```

The following table lists the defaults for each window field and indicates if the field is applicable to **InputOutput** or **InputOnly** windows.

Window Field	Default Values	InputOutput	InputOnly
background_pixmap	None	Yes	No
background_pixel	Undefined	Yes	No
border_pixmap	CopyFromParent	Yes	No

border_pixel	Undefined	Yes	No
bit_gravity	ForgetGravity	Yes	No
win_gravity	NorthWestGravity	Yes	Yes
backing_store	NotUseful	Yes	No
backing_planes	All 1s	Yes	No
backing_pixel	0	Yes	No
save_under	False	Yes	No
event_mask	empty set	Yes	Yes
do_not_propagate_mask	empty set	Yes	Yes
override_redirect	False	Yes	Yes
colormap	CopyFromParent	Yes	No
cursor	None	Yes	Yes

The fields of the **XSetWindowAttributes** data structure are as follows:

background_pixmap Specifies the pixmap to be used for a window background. This pixmap can be any size, but some sizes are faster than others. Use the **XQueryBest Sizes** subroutine to determine the optimum size. The *background_pixmap* field can be set to a pixmap ID, or either the value of **None** or **ParentRelative.** The default is the value of **None**. The *background_pixmap* field and the window must have the same depth, or a **BadMatch** error is returned.

Only **InputOutput** windows can have backgrounds.

When regions of the window are exposed and the X Server has not retained the contents of the window, the X Server automatically tiles the regions with the window background as long as the *background_pixmap* field is not the value of **None.**

- If the *background_pixmap* field is set to the value of **None,** the contents of the previous screen are left in place if the window and the parent window have the same depth. Otherwise, the initial contents of the exposed regions are undefined. **Expose** events are then generated for the regions, even if the *background_pixmap* field is the value of **None.**
- If the *background_pixmap* field is set to **ParentRelative,** the following occurs:
  - The *background_pixmap* field of the parent window is used if the child window has the same depth as the parent window, or a **BadMatch** error is returned.
  - The window has no defined background. If the parent window has a *background_pixmap* field of the value of **None**, the window also has a *background_pixmap* field of the value of **None**.
  - A copy of the *background_pixmap* field of the parent window is not made. The *background_pixmap* field of the

parent window is examined each time the *background_pixmap* field of the child window is required.

- The background tile origin always aligns with the background tile origin of the parent window. Otherwise, the background tile origin is always the child window origin.

Setting a new background with the *background_pixmap* field overrides any previous *background_pixmap* field. The *background_pixmap* field can be freed immediately if no further explicit reference is made to it. The X Server keeps a copy to use when needed.

background_pixel

Specifies a pixel value of a single color for the background of the window. This field can be set to any pixel value. The default value for the *background_pixel* field is undefined.

If the *background_pixel* field is specified, it overrides the default *background_pixmap* field or any value set in the *background_pixmap* field, and a pixmap of undefined size is created and filled with the specified pixel and used for the background. All pixels, in the background of the window, will be set to this value. Range checking is not performed on the pixel, as it is truncated to the appropriate number of bits.

Setting a new background with the *background_pixel* field overrides any previous background.

border_pixmap

Specifies the pixmap for the border of a window. This pixmap can be any size. The *border_pixmap* field and the child window must have the same depth, or a **BadMatch** error is returned.

Only **InputOutput** windows can have a border.

Setting a new border with the *border_pixmap* field overrides any previous border. Setting a *border_pixmap* field value overrides the default value. The default value is **CopyFromParent**.

If the *border_pixmap* is **CopyFromParent**, the *border_pixmap* field is copied from the parent window. Subsequent changes to the border attribute of the parent window do not affect the child window.

The pixmap used for the *border_pixmap* field can be freed immediately if no further explicit reference to it is made. If the pixmap used for the *border_pixmap* field is freed, the X Server may or may not keep a copy of it. The X Server can use the same pixmap each time the window is repainted or it may make a copy.

border_pixel

Specifies a pixel value to be used for the window border. The server creates a pixmap of unspecified size filled with the pixel for the window border. The border tile origin is always the same as the background tile origin. The default for the *border_pixel* field is undefined. Range checking is not performed on the pixel, as it is truncated to the appropriate number of bits.

Only **InputOutput** windows can have a border.

If you specify a *border_pixel* field, it overrides the default value or the assigned value of *border_pixmap* field. Then, all pixels in the border of the window are set to the *border_pixel* field value.

The output to a window is always clipped to the inside of the window so that graphics operations are not affected by the border.

bit_gravity

Specifies which region of the window should be retained when an **InputOutput** window is resized. The default *bit_gravity* is the **ForgetGravity** value. Changing the inside width or height of the window causes the contents of the window to be moved or lost depending on the *bit_gravity* field of the window. The values for the *bit_gravity* field include:

ForgetGravity Indicates that the contents of the window are always discarded after a size change, even if a backing store or save under has been requested. The window is tiled with its background, and one or more **Expose** events are generated. If no background is defined, the existing screen contents are not altered. Some X Servers may ignore the specified *bit_gravity* field and always generate exposure events.

StaticGravity Indicates that the contents or origin of the window should not move relative to the origin of the root window. If the change in size of the window is coupled with a change in position (x, y), the change in position of each pixel becomes (-x, -y).

The **StaticGravity** value takes effect only when the width or height of the window is changed, not when the window is moved.

If the inside width or height of a window is not changed and if the window is moved or its border is changed, the contents of the window are not lost but are moved with the window. For a change of width and height, the (x, y) pairs are defined as follows:

Gravity Coordinates	**Direction**
NorthWestGravity	(0, 0)
NorthGravity	(Width/2, 0)
NorthEastGravity	(Width, 0)
WestGravity	(0, Height/2)
CenterGravity	(Width/2, Height/2)

EastGravity	(Width, Height/2)
SouthWestGravity	(0, Height)
SouthGravity	(Width/2, Height)
SouthEastGravity	(Width, Height)

When a window with one of these *bit_gravity* field values is resized, the corresponding pair defines the change in position of each pixel in the window.

win_gravity

Specifies how the **InputOutput** or **InputOnly** window should be repositioned if the parent window is resized. Changing the inside width or height of the window causes child windows to be reconfigured, depending on the specified *win_gravity* field. The default for the *win_gravity* field is the value of **NorthWestGravity**.

If the inside width or height of a window is not changed and if the window is moved or its border is changed, the contents of the window are not lost but are moved with the window. For a change of width and height, the (x, y) pairs are defined as follows:

Gravity Coordinates	**Direction**
NorthWestGravity	(0, 0)
NorthGravity	(Width/2, 0)
NorthEastGravity	(Width, 0)
WestGravity	(0, Height/2)
CenterGravity	(Width/2, Height/2)
EastGravity	(Width, Height/2)
SouthWestGravity	(0, Height)
SouthGravity	(Width/2, Height)
SouthEastGravity	(Width, Height)

When a window with one of these *win_gravity* field values has its parent window resized, the corresponding pair defines the change in position of the window within the parent window. When the window is repositioned, a **GravityNotify** event is generated.

StaticGravity

If the change in size of the window is coupled with a change in position (x, y), the change in position of a child window when the parent window is resized becomes (-x, -y).

The **StaticGravity** value takes effect only when the width or height of the window is changed, not when the window is moved.

UnmapGravity	This is like the **NorthWestGravity** value; the window is not moved, but the child window is unmapped when the parent window is resized, and an **UnmapNotify** event is generated.

backing_store

Advises the X Server what to do with the contents of a window. Some implementations may choose to maintain the contents of **InputOutput** windows. If the X Server maintains the contents of a window, the pixels saved offscreen are known as the backing store. This field can be set to the following values:

NotUseful	Advises the X Server that maintaining contents is not necessary. Some X implementations can still maintain contents; therefore, exposure events are not generated. This is the default value.
WhenMapped	Advises the X Server that maintaining contents of obscured regions when the window is mapped would be beneficial. The X Server can generate an **Expose** event when the window is created.
Always	Advises the X Server that maintaining contents even when the window is unmapped would be beneficial. Even if the window is larger than the parent window, this requests that the X Server maintain the complete contents of the window, not just the contents of the region within the boundaries of the parent window. While the X Server maintains the contents of the window, **Expose** events are not normally generated. The X Server can stop maintaining contents at any time.

When the contents of obscured regions of a window are being maintained, the regions obscured by noninferior windows are included in the destination of graphics requests (and source, when the window is the source). Regions obscured by inferior windows, however, are not included.

backing_planes

Indicates (with one bits) which bit planes of the **InputOutput** window hold dynamic data that must be preserved in the backing store and during save unders. If you request backing store or save unders, the *backing_planes* field will minimize the amount of off–screen memory required to store your window. The default is all bits set to the value of **1**.

backing_pixel

Specifies the values to use in planes not covered by the *backing_planes* field. The X Server is free to save only the specified bit planes in the backing store or the save under and is free to regenerate the remaining planes with the specified

	pixel value. Any extraneous bits in these values, beyond the depth of the window, can be ignored. If you request backing store or save unders, the *backing_pixel* field will minimize the amount of off–screen memory required to store your window. The default is the value of **0**.
save_under	If the *save_under* field is the value of **True**, the X Server is advised that saving the contents of the windows that it obscures would be beneficial when this window is mapped. The default is the value of **False**. Some server implementations can preserve bits of **InputOutput** windows under other **InputOutput** windows. This is not the same as preserving the contents of a window. If transient windows, such as pop-up menus, request that the system preserve the bits under them, the temporarily obscured applications do not have to repaint.
event_mask	Defines events the client is interested in for this **InputOutput** or **InputOnly** window or, in some cases, for the inferiors of the window. This mask is the bitwise-inclusive OR of one or more of the valid event mask bits. If the **NoEventMask** constant is specified, no maskable events are reported. The default is the empty set.
do_not_propagate_mask	Defines events that should not be propagated to ancestor windows when no client has the event type selected in this window. These masks are the bitwise-inclusive OR of one or more of the valid event mask bits. If the **NoEventMask** constant is specified, no maskable events are reported. The default is the empty set.
override_redirect	Specifies if a map or configure request on a window should override a **SubstructureRedirectMask** request on the parent window. The default is the value of **False**. To control window placement or to add decoration, a window manager may need to intercept or redirect a map or configure request. Pop-up windows, however, need to be mapped so that a window manager does not interfere with the response they receive. To do this, the *override_redirect* field must be used.
colormap	Specifies the colormap, if any, that best reflects the true colors of an **InputOutput** window. The colormap must have the same visual type as the window, or a **BadMatch** error is returned. The *colormap* field can be set to a specific colormap or to the value of **CopyFromParent**, which is the default. If the *colormap* field is set to the value of **CopyFromParent**, the colormap of the parent window is copied and used by the child window. However, the child window must have the same visual type as the parent, or a **BadMatch** error is returned. A **BadMatch** error is also returned if the parent window has the *colormap* field is set to the value of **None**. The colormap is copied by sharing the colormap object between the child and parent windows, not by making a complete copy of the

colormap contents. Subsequent changes to the parent window do not affect the child window.

cursor — If a cursor is specified, it is used whenever the pointer is in the specified window. The default is the value of **None**.

If the value of **None** is specified, the cursor of the parent is used when the pointer is in the **InputOutput** or **InputOnly** window. Any change in the parent cursor will change the displayed cursor immediately. Use the **XFreeCursor** subroutine to free the cursor immediately if no further explicit reference to it is made.

XWindowChanges Data Structure

```
#define   CWX            (1<<0)
#define   CWY            (1<<1)
#define   CWWidth        (1<<2)
#define   CWHeight       (1<<3)
#define   CWBorderWidth  (1<<4)
#define   CWSibling      (1<<5)
#define   CWStackMode    (1<<6)
```

```
typedef struct {
int x, y;
int width, height;
int border_width;
Window sibling;
int stack_mode;
} XWindowChanges
```

The fields of the **XWindowChanges** data structure are as follows:

x — Specifies the position of the x coordinate of the upper-left outer corner of the window. This coordinate is relative to the origin of the parent window.

y — Specifies the position of the y coordinate of the upper-left outer corner of the window. This coordinate is relative to the origin of the parent window.

width — Specifies the width of the inside of the window, excluding the border. This field should be a nonzero value or a **BadValue** error is returned. Attempts to configure a root window have no effect.

height — Specifies the width of the inside of the window, excluding the border. This field should be a nonzero value or a **BadValue** error is returned. Attempts to configure a root window have no effect.

border_width — Specifies the width of the border in pixels. Changing only the *border_width* field leaves the outer-left corner of the window in a fixed position, but moves the absolute position of the window origin. Attempts to change the *border_width* field on an **InputOnly** window will result in a **BadMatch** error.

sibling — Specifies the sibling window for stacking operations. If the *sibling* field is specified without the *stack_mode* field, a **BadMatch** error will result.

stack_mode Specifies how the window is to be restacked. The *stack_mode* field can be the **Above**, **Below**, **TopIf**, **BottomIf**, or **Opposite** value.

If a *sibling* and a *stack_mode* field are specified, the window is restacked as follows:

StackMode	**Sibling Specified.**
Above	The window is placed just above the sibling window.
Below	The window is placed just below the sibling window.
TopIf	If the sibling window occludes the window, the window is placed at the top of the stack.
BottomIf	If the window occludes the sibling window, the window is placed at the bottom of the stack.
Opposite	If the sibling window occludes the window, the window is placed at the top of the stack. Otherwise, if the window occludes the sibling window, the window is placed at the bottom of the stack.

If the *stack_mode* field is specified without a sibling window, the window is restacked as follows:

StackMode	**Sibling Not Specified.**
Above	The window is placed at the top of the stack.
Below	The window is placed at the bottom of the stack.
TopIf	If any sibling window occludes the window, the window is placed at the top of the stack.
BottomIf	If the window occludes any sibling window, the window is placed at the bottom of the stack.
Opposite	If any sibling window occludes the window, the window is placed at the top of the stack. Otherwise, if the window occludes any sibling window, the window is placed at the bottom of the stack.

If the *override_redirect* field of the window is the value of **False**, and if some other client has selected the **SubstructureRedirectMask** on the parent window, then the X Server generates a **ConfigureRequest** event, and no further processing is performed.

Otherwise, if some other client has selected the **ResizeRedirectMask** value on the window and the inside width or height of the window is being changed, a **ResizeRequest** event is generated, and the current inside width and height are used instead.

The *override_redirect* field of the window does not affect the **ResizeRedirectMask** value, and the **SubstructureRedirectMask** value on the parent window has precedence over the **ResizeRedirectMask** value on the window.

When the geometry of the window is changed as specified, the window is restacked among sibling windows, and a **ConfigureNotify** event is generated if the state of the window actually changes. The X Server generates **GravityNotify** events after generating **ConfigureNotify** events. If the inside width or height of the window has changed, the children of the window are affected as follows:

- If the size of the window actually changes, the subwindow may move according to the window gravity. Depending on the window's bit gravity, the contents of the window may also be moved.
- If regions of the window were obscured but are not now, exposure processing is performed on formerly obscured windows, including the window itself and its inferiors.
- By increasing the width or height, exposure processing is also performed on any new regions of the window and on any regions where window contents are lost.
- The restack check, specifically the computation for the **BottomIf**, **TopIf**, and **Opposite** values, is performed with respect to the final size and position of the window as controlled by the other fields of the request, not the initial position of the window. A *sibling* field should be specified with a *stack_mode* field.

Related Information

The **XChangeWindowAttributes** subroutine, **XCirculateSubwindows** subroutine, **XCirculateSubwindowsDown** subroutine, **XCirculateSubwindowsUp** subroutine, **XConfigureWindow** subroutine, **XLowerWindow** subroutine, **XMoveResizeWindow** subroutine, **XMoveWindow** subroutine, **XRaiseWindow** subroutine, **XResizeWindow** subroutine, **XRestackWindows** subroutine, **XSetWindowBackground** subroutine, **XSetWindowBackgroundPixmap** subroutine, **XSetWindowBorder** subroutine, **XSetWindowBorderPixmap** subroutine, **XSetWindowBorderWidth** subroutine, **XTranslateCoordinates** subroutine.

The **ChangeWindowAttributes** protocol request, **CirculateWindow** protocol request, **ConfigureWindow** protocol request.

XWindowsAttributes Data Structure

```
typedef struct {
   int x, y;                 /* location of window */
    int width, height;        /* width and height of window */
   int border_width;         /* border width of window */
   int depth;                /* depth of window */
    Visual *visual;           /* the associated visual structure */
    Window root;              /* root of screen containing window */
   int class;                /* InputOutput, InputOnly */
   int bit_gravity           /* one of the bit gravity values */
   int window_gravity;       /* one of the window gravity values */
   int backing_store;        /* NotUseful, WhenMapped, Always */
   unsigned long backing_planes;/* planes to be preserved if
                                    possible */
    unsigned long backing_pixel; /* value to be used when restoring
                                    planes */
   Bool save_under;          /* boolean, should bits under be
                                saved? */
    Colormap colormap;        /* colormap to be associated with
                                window */
    Bool map_installed;       /* boolean, is colormap currently
                                installed? */
   int map_state;            /* IsUnmapped, IsUnviewable,
                                IsViewable */
   long all_event_masks;     /* set of events all people have
                                interest in */
   long your_event_mask;     /* my event mask*/
   long do_not_propagate_mask;  /* set of events that should not
                                    propagate */
    Bool override_redirect;   /* boolean value for
                                override-redirect */
    Screen *screen;           /* back pointer to correct screen */
} XWindowAttributes;
```

The fields of the **XWindowAttributes** data structure are as follows:

x	Defines the x coordinate of the drawable. If the drawable is a window, this coordinate specifies the upper-left outer-corner of the window relative to the origin of the parent window. If the drawable is a pixmap, this coordinate is set to a value of 0.
y	Defines the x coordinate of the drawable. If the drawable is a window, this coordinate specifies the upper-left outer-corner of the window relative to the origin of the parent window. If the drawable is a pixmap, this coordinate is set to the value of **0**.
width	Specifies the width of the inside of the window, excluding the border, for a window.
height	Specifies the height of the inside of the window, excluding the border, for a window.
border_width	Specifies the width of the border in pixels. If the drawable is a pixmap, this field is set to the value of **0**.

depth	Specifies the depth of the pixmap or window in bits per pixel for the object. The depth must be supported by the root window of the specified drawable.
visual	Specifies a pointer to the **Visual** structure associated with the screen.
root	Specifies the root ID of the screen containing the window.
class	Specifies the window class, which is either the **InputOutput** or **InputOnly** value.
bit_gravity	Specifies the bit gravity of the window. The *bit_gravity* field can be set to one of the following values: **CenterGravity** **SouthGravity** **EastGravity** **SouthEastGravity** **ForgetGravity** **SouthWestGravity** **NorthGravity** **StaticGravity** **NorthEastGravity** **WestGravity** **NorthWestGravity**
win_gravity	Specifies the gravity of the window. The *win_gravity* field can be set to one of the following values: **CenterGravity** **SouthGravity** **EastGravity** **SouthEastGravity** **ForgetGravity** **SouthWestGravity** **NorthGravity** **StaticGravity** **NorthEastGravity** **WestGravity** **NorthWestGravity**
backing_store	Indicates how the X Server should maintain the contents of a window. It can be set to the **NotUseful**, **WhenMapped**, or **Always** value.
backing_planes	Indicates which bit planes of the window that hold dynamic data must be preserved in *backing_store* and during *save_under*.
backing_pixel	Indicates the values to use when restoring planes from a partial backing store.
save_under	Indicates if the area under the newly mapped windows should be saved and restored. It can be either the value of **True** or **False**.
colormap	Indicates colormap for the window specified. It can be set to a colormap ID or to the value of **None**.
map_installed	Indicates if the colormap is currently installed. It can be either the value of **True** or **False**.

map_state	Indicates the state of the window. It can be **IsUnmapped, IsUnviewable**, or **IsViewable** value. If the window is mapped, but an ancestor is unmapped, the *map_state* field is set to the value of **IsUnviewable**.
all_event_masks	Specifies the bitwise inclusive-OR of all event masks selected on the window by interested clients.
your_event_mask	Specifies the bitwise inclusive-OR of all event masks selected by the querying client.
do_not_propagate_mask	Specifies the bitwise inclusive-OR gate or operation of the set of events that should not propagate.
override_redirect	Indicates if this window overrides structure control facilities. It can be either the value of **True** or **False**. If the *override_redirect* field is the value of **True**, window manager clients usually should ignore the window. Transient windows should mark the windows associated with them.
screen	Returns a pointer to the correct screen.

Related Information

The **XGetGeometry** subroutine, **XGetWindowAttributes** subroutine, **XQueryPointer** subroutine, **XQueryTree** subroutine.

XColor Data Structure

```
typedef struct {
    unsigned long pixel;               /* pixel value */
   unsigned short red, green, blue;  /* RGB values */
    char flags;                        /* DoRed, DoGreen, DoBlue */
   char pad;
} XColor;
```

The fields of the **XColor** data structure are as follows:

blue	Specifies a scale between 0 and 65535. That is, on full, the color value would be 65535 independent of the number of bit planes of the display. For half brightness in a color, the value would be 32767. The color value would be 0 for off. This value gives uniform results for color values across displays with different numbers of bit planes.
flags	Specifies which colors are to be set. The *flags* field can be one or more of the **DoRed**, **DoGreen**, or **DoBlue** values.
green	Specifies a scale between 0 and 65535. That is, on full, the color value would be 65535 independent of the number of bit planes of the display. For half brightness in a color, the value would be 32767. The color value would be 0 for off. This value gives uniform results for color values across displays with different numbers of bit planes.

pad	Indicates to fill unused positions in a field with dummy data, usually zeros or blanks.
pixel	Specifies the value of the pixel.
red	Specifies a scale between 0 and 65535. That is, on full, the color value would be 65535 independent of the number of bit planes of the display. For half brightness in a color, the value would be 32767. The color value would be 0 for off. This value gives uniform results for color values across displays with different numbers of bit planes.

Related Information

The **XCopyColormapAndFree** subroutine, **XCreateColormap** subroutine, **XFreeColormap** subroutine, **XSetWindowColormap** subroutine.

The **ChangeWindowAttributes** protocol request, **CopyColormapAndFree**, **CreateColormap** protocol request, **FreeColormap** protocol request.

XGCValues Data Structure

```
#define   GCFunction             (1L<<0)
#define   GCPlaneMask            (1L<<1)
#define   GCForeground           (1L<<2)
#define   GCBackground           (1L<<3)
#define   GCLineWidth            (1L<<4)
#define   GCLineStyle            (1L<<5)
#define   GCCapStyle             (1L<<6)
#define   GCJoinStyle            (1L<<7)
#define   GCFillStyle            (1L<<8)
#define   GCFillRule             (1L<<9)
#define   GCTile                 (1L<<10)
#define   GCStipple              (1L<<11)
#define   GCTileStipXOrigin      (1L<<12)
#define   GCTileStipYOrigin      (1L<<13)
#define   GCFont                 (1L<<14)
#define   GCSubwindowMode        (1L<<15)
#define   GCGraphicsExposures    (1L<<16)
#define   GCClipXOrigin          (1L<<17)
#define   GCClipYOrigin          (1L<<18)
#define   GCClipMask             (1L<<19)
#define   GCDashOffset           (1L<<20)
#define   GCDashList             (1L<<21)
#define   GCArcMode              (1L<<22)


typedef struct {
   int function;             /* logical operation */
   unsigned long plane_mask; /* plane mask */
   Unsigned long foreground; /* foreground pixel */
   unsigned long background; /* background pixel */
   int line_width;           /* line width (in pixels) */
   int line_style;           /* LineSolid, LineOnOffDash,
                                LineDoubleDash */
   int cap_style;            /* CapNotLast, CapButt, CapRound,
                                CapProjecting */
```

```
    int join_style;          /* JoinMiter, JoinRound,
                                JoinBevel */
    int fill_style;          /* FillSolid, FillTiled,
                                FillStippled, or
                                FillOpaqueStippled */
    int fill_rule;           /* EvenOddRule, WindingRule */
    int arc_mode;            /* ArcChord, ArcPieSlice */
   Pixmap tile;             /* tile pixmap for tiling
                                operations */
   Pixmap stipple;          /* stipple 1 plane pixmap for
                                stippling */
    int ts_x_origin;         /* x offset for tile or stipple
                                operations */
    int ts_y_origin;         /* y offset for tile or stipple
                                operations */
   Font font;               /* default text font for text
                                operations */
    int subwindow_mode;      /* ClipByChildren,
                                IncludeInferiors */
   Bool graphics_exposures; /* Boolean, should exposures be
                                generated */
    int clip_x_origin;       /* x origin for clipping */
    int clip_y_origin;       /* y origin for clipping */
   Pixmap clip_mask;        /* bitmap clipping; other calls
                                for rects */
    int dash_offset;         /* patterned or dashed line
                                information */

    char dashes;
} XGCValues;
```

In graphics operations, given a source and destination pixel, the result is computed bitwise on corresponding bits of the pixels. A Boolean operation is performed in each bit plane.

For a line with coincident endpoints (x1=x2, y1=y2), when the *cap_style* field is applied to both endpoints, the semantics depends on the *line_width* field and the *cap_style* field:

CapNotLast	thin	Nothing is drawn.
CapButt	thin	A single pixel is drawn.
CapButt	wide	Nothing is drawn.
CapRound	wide	The closed path is a circle, centered at the endpoint, with diameter equal to the *line_width* field.
CapRound	thin	A single pixel is drawn.
CapProjecting	wide	The closed path is a square 4, aligned with the coordinate axes, centered at the endpoint, with sides equal to the *line_width* field.
CapProjecting	thin	A single pixel is drawn.

For a line with concident endpoints (x1=x2, y1=y2), when the *join_style* field is applied at one or both endpoints, the effect is as if the line was removed from the overall path. However, if the total path consists of or is reduced to a single point joined with itself, the effect is the same as when the *cap_style* field is applied at both endpoints.

The fields of the **XGCValues** data structure are as follows:

function Specifies the logical operation to be performed.

plane_mask Specifies the operations to a subset of planes that need to be restricted. The result is computed by the following:

```
((src FUNC dst) AND plane-mask) OR (dst AND (NOT plane-mask))
```

Range checking is not performed on the values for the *foreground*, *background*, or *plane_mask* fields. These values are truncated to the appropriate number of bits.

line_width Specifies the width of the line, measured in pixels. It can be greater than or equal to 1 (wide line) or can be the special value 0 (thin line).

Wide lines are drawn centered on the path described by the graphics request.

Unless otherwise specified by the *join_style* or the *cap_style* fields, the bounding box of a wide line with endpoints [x1, y1] to [x2, y2] and width w is a rectangle with vertices at the following real coordinates:

```
[x1-(w*sn/2), y1+(w*cs/2)], [x1+(w*sn/2), y1-(w*cs/2)]
,
[x1-(w*sn/2), y1+(w*cs/2)], [x1+(w*sn/2), y1-(w*cs/2)]
```

In this example, sn is the sine of the angle of the line, and cs is the cosine of the angle of the line. A pixel is part of the line and is drawn that way only if the center of the pixel is fully inside the bounding box, which has infinitely thin edges. If the center of the pixel is exactly on the bounding box, it is part of the line only if the interior is immediately to its right (the x increasing direction). Pixels with centers on a horizontal edge are a special case and are part of the line only if the interior or the boundary is immediately below (the y increasing direction) and the interior or the boundary is immediately to the right (the x increasing direction).

Thin lines (zero line width) are one pixel wide lines drawn using an unspecified, device-dependent algorithm. There are only the following two constraints on this algorithm:

- If a line is drawn unclipped from `[x1, y1]` to `[x2, y2]` and if another line is drawn unclipped from `[x1+dx, y1+dy]` to `[x2+dx, y2+dy]`, a point `[x, y]` is touched by drawing the first line only if the point `[x+dx, y+dy]` is touched by drawing the second line.
- The effective set of points comprising a line cannot be affected by clipping. That is, a point is touched in a clipped line only if the point lies inside the clipping region and the point would be touched by the line when drawn unclipped.

A wide line drawn from `[x1, y1]` to `[x2, y2]` always draws the same pixels as a wide line drawn from `[x2, y2]` to `[x1, y1]`, not counting the *cap_style* and the *join_style* fields. A *line_width* field of 0 may differ from a *line_width* field of 1 in which pixels are drawn.

In general, drawing a thin line is faster than drawing a wide line of width 1. However, because of their different drawing algorithms, thin lines may not mix well, aesthetically speaking, with wide lines. If it is desirable to obtain precise and uniform results across all displays, a client should always use a *line_width* of 1, rather than a *line_width* of 0.

line_style Defines which sections of a line are drawn. The *line_style* field can be one of the following values:

LineSolid	The full path of the line is drawn.
LineDoubleDash	The full path of the line is drawn, but the even dashes are filled differently than the odd dashes (see fill-style) with the **CapButt** value used where even and odd dashes meet.
LineOnOffDash	Only the even dashes are drawn, and the *cap_style* field applies to all internal ends of the individual dashes, except the **CapNotLast** value is treated as the **CapButt** value.

cap_style Defines how the endpoints of a path are drawn. The *cap_style* field can be one of the following values:

CapNotLast	Equivalent to the **CapButt** value, except that for a *line_width* field of 0 or 1, the final endpoint is not drawn.
CapButt	Square at the endpoint, perpendicular to the slope of the line, with no projection beyond.
CapRound	A circular arc with the diameter equal to the *line_width* field, centered on the endpoint (equivalent to the **CapButt** value for a *line_width* field 0 or 1).
CapProjecting	Square at the end, but the path continues beyond the endpoint for a distance equal to half the *line_width* field (equivalent to the **CapButt** value for *line_width* field 0 or 1).

join_style Defines how corners are drawn for wide lines. The *join_style* field can be one of the following values:

JoinMiter	The outer edges of two lines extend to meet at an angle.
JoinRound	A circular arc with diameter equal to the *line_width* field, centered on the join point.

JoinBevel	**CapButt** endpoint styles, and then the triangular notch filled.

tile
: Specifies the tile to be used. The tile pixmap must have the same root window and depth as the graphics context, or a **BadMatch** error results.. The *tile* field is interpreted relative to the origin of whatever destination drawable is specified in a graphics request.

stipple
: Specifies the stipple to be used. The stipple pixmap must have depth one and must have the same root window as the GC, or a **BadMatch** error results. For stipple operations where the *fill_style* field is **FillStippled**, but not **FillOpaqueStippled**, the stipple pattern is tiled in a single plane and acts as an additional clip mask to be ANDed with the *clip_mask* field. Any size pixmap can be used for tiling or stippling although some sizes may be faster to use than others.The *stipple* field is interpreted relative to the origin of whatever destination drawable is specified in a graphics request.

clip_x_origin
: Specifies that the field is interpreted relative to the origin of whatever destination drawable is specified in a graphics request.

clip_y_origin
: Specifies that the field is interpreted relative to the origin of whatever destination drawable is specified in a graphics request.

fill_style
: Specifies that the field defines the contents of the source for line, text, and fill requests. For all text and fill requests, for line requests with *line_style* field of **LineSolid**, and for the even dashes for line requests with *line_style* field of **LineOnOffDash** or **LineDoubleDash** the following applies:

FillSolid	Foreground.
FillTiled	Tile.
FillOpaqueStippled	A tile with the same width and height as stipple, but with stipple background has a 0 and with stipple foreground has a 1.
FillStippled	Foreground masked by stipple.

When drawing lines with the *line_style* field value of **LineDoubleDash**, the odd dashes are controlled by the *fill_style* field in the following manner:

FillSolid	Background.
FillTiled	Same as even dashes.
FillOpaqueStippled	Same as even dashes.
FillStippled	Background masked by stipple.

Storing a pixmap in a **GC** may result in a copy being made. If the pixmap is later used as the destination for a graphics request, the change may be reflected in the graphics context. If the pixmap is used simultaneously in a graphics request both as a destination and as a tile or stipple, the results are not defined.

dashes

Specifies the dash list for the dashed line style to be set for the specified **GC**.

The value allowed by the *dashes* field is a simplified form of the more general patterns that can be set with the **XSetDashes** subroutine. Specifying a value of `N` is equivalent to specifying the two-element list `[N, N]` in the **XSetDashes** subroutine. `N` specifies the length of the dash list. This value must be nonzero, or a **BadValue** error results. The default dash list in a newly created **GC** is equivalent to [4,4].

dash_offset

Specifies the phase of the pattern for the dashed line style to be set for the graphics context specifying how many elements into the dash list the pattern should actually begin in any single graphics request.

Dashing is continuous through path elements combined with *join_style*, but is reset to the *dash_offset* each time a *cap_style* is applied at a line endpoint.

The unit of measure for dashes is the same as in the ordinary coordinate system. Ideally, a dash length is measured along the slope of the line, but implementations are required to match only for horizointal and vertical lines. It is suggested that you measure the length along the major axis of the line. The major axis is defined as the x axis for lines drawn at an angle of between -45 and +45 degrees or between 315 and 225 degrees from the x axis. For all other lines, the major axis is the y axis. The default value for the dash list in a newly created **GC** is equivalent to [4, 4].

clip_mask

Restricts writes to the destination drawable. If a pixmap is specified as the *clip_mask* field, it must have a depth of one and have the same root window as the **GC**. If the *clip_mask* field is the value of **None**, the pixels are always drawn, regardless of the clip origin. The *clip_mask* field can also be set with the **XSetClipRectangles** or **XSetRegion** subroutines. Only pixels where the *clip_mask* field has a 1-bit are drawn. Pixels are not drawn outside the area covered by the *clip_mask* field or where the *clip_mask* field has a 0 bit. It affects all graphics requests.The *clip_mask* field does not clip sources.The *clip_mask* field origin is interpreted relative to the origin of the specified destination drawable in the graphics request.

subwindow_mode

Specifies how the subwindow should be clipped. The *subwindow_mode* field can be one of the following values:

ClipByChildren	Indicates that both source and destination windows are clipped additionally by all viewable **InputOutput** children.
IncludeInferiors	Indicates that neither the source window nor the destination window is clipped by inferiors. This results in drawing through the boundaries of subwindows. Using the **IncludeInferiors** value on a window

with the depth of one with mapped inferiors of differing depth is allowed, but the semantics are undefined by the core protocol.

fill_rule

Defines which pixels are inside (drawn) for paths given in the **XFillPolygon** subroutine. The *fill_rule* field can be set to one of the following values:

EvenOddRule	Specifies that a point is inside if an infinite ray with the point as origin crosses the path an odd number of times.
WindingRule	Specifies that a point is inside if an infinite ray with the point as origin crosses an unequal number of clockwise and counterclockwise directed path segments.A clockwise directed path segment is one that crosses the ray from left to right as observed from the point.A counterclockwise segment is one that crosses the ray from right to left as observed from the point.

For **EvenOddRule** and **WindingRule**, a point is infinitely small, and the path is an infinitely thin line.

arc_mode

Controls filling in the **XFillArcs** subroutine. The *arc_mode* field can be one of the following values:

ArcPieSlice	Specifies that arcs are pie-sliced filled.
ArcChord	Specifies that arcs are chord-filled.

graphics_exposures

Controls the **GraphicsExpose** event generation for the **XCopyArea** and **XCopyPlane** subroutines and any similar requests defined by extension subroutines. The **GraphicsExpose** events are sent even when they are not explicitly requested. To suppress them, set the *graphics_exposures* field to the value of **False**.

Related Information

The **AllPlanes** macro.

The **XChangeGC** subroutine, **XCopyGC** subroutine, **XCreateGC** subroutine, **XFreeGC** subroutine, **XQueryBestSize** subroutine, **XQueryBestStipple** subroutine, **XQueryBestTile** subroutine, **XSetArcMode** subroutine, **XSetBackground** subroutine, **XSetClipMask** subroutine, **XSetClipOrigin** subroutine, **XSetClipRectangles** subroutine, **XSetDashes** subroutine, **XSetFillRule** subroutine, **XSetFillStyle** subroutine, **XSetFont** subroutine, **XSetForeground** subroutine, **XSetFunction** subroutine, **XSetGraphicsExposures** subroutine, **XSetLineAttributes** subroutine, **XSetPlaneMask** subroutine, **XSetState** subroutine, **XSetStipple** subroutine, **XSetSubwindowMode** subroutine, **XSetTile** subroutine, **XSetTSOrigin** subroutine.

XStandardColormap Data Structure

```
typedef struct {
    Colormap colormap;
   unsigned long red_max;
   unsigned long red_mult;
   unsigned long green_max;
   unsigned long green_mult;
    unsigned long blue_max;
   unsigned long blue_mult;
    unsigned long base_pixel;
} XStandardColormap;
```

The properties containing the **XStandardColormap** data structure have the type RGB_COLOR_MAP.

The fields in the **XStandardColormap** data structure include the following:

colormap	Specifies the ID of a colormap created by the **XCreateColormap** subroutine.
red_max	Specifies the maximum red values.
green_max	Specifies the maximum green values.
blue_max	Specifies the maximum blue values. Each color coefficient ranges from zero (0) to its maximum, inclusive.

An example of a common colormap allocation is `3/3/2` (3 planes for red, 3 planes for green, and 2 planes for blue). This colormap would have *red_max* = `7`, *green_max* = `7`, and *blue_max* = `3`.

An alternate allocation that uses only 216 colors is *red_max* = `5`, *green_max* = `5`, and *blue_max* = `5`.

red_mult	Specifies the scale factors used to compose a full pixel value.
green_mult	Specifies the scale factors used to compose a full pixel value.
blue_mult	Specifies the scale factors used to compose a full pixel value.

For a `3/3/2` allocation *red_mult* might be `32`, *green_mult* might be `4`, and *blue_mult* might be `1`.

For a 6-colors-each allocation, *red_mult* might be `36`, *green_mult* might be `6`, and *blue_mult* might be `1`.

base_pixel	Specifies the base pixel value used to compose a full pixel value. The *base_pixel* field value is usually obtained from the **XAllocColorPlanes** subroutine.

Given integer red, green, and blue coefficients in the appropriate ranges, you can compute a corresponding pixel value by using the following expression:

```
r * red_mult + g * green_mult + b * blue_mult + base_pixel
```

For **GrayScale** colormaps, only the *colormap*, *red_max*, *red_mult*, and *base_pixel* fields are defined. The other fields are ignored. Compute a gray-scale pixel value by using the following expression:

```
gray * red_mult + base_pixel
```

XSegment Data Structure

```
typedef struct {
   short x1, x2, y1, y2;
} XSegment;
```

All *x* and *y* fields are 16-bit signed integers. Do not generate coordinates and sizes out of the 16-bit ranges because the protocol has only 16-bit fields for these values. For example, the rectangle {0,0,50000,1} references the coordinates x > = 49,999, and y = 0. This cannot be represented in 16 bits and the results are not defined.

Related Information

The **XDrawLine** subroutine, **XDrawLines** subroutine, **XDrawRectangle** subroutine, **XDrawRectangles** subroutine, **XDrawSegments** subroutine.

The **PolySegment** protocol request.

XRectangle Data Structure

```
typedef struct {
   short x, y;
   unsigned short width, height;
} XRectangle;
```

All *x* and *y* fields are 16-bit signed integers. The *width* and *height* fields are 16-bit unsigned integers. Do not generate coordinates and sizes out of the 16-bit ranges because the protocol has only 16-bit fields for these values. For example, the rectangle {0,0,50000,1} references the coordinates X > = 49,999, and Y = 0. This cannot be represented in 16 bits and the results are not defined.

Related Information

The **XDrawLine** subroutine, **XDrawLines** subroutine, **XDrawRectangle** subroutine, **XDrawRectangles** subroutine, **XDrawSegments** subroutine.

The **PolyRectangle** protocol request

XPoint Data Structure

```
typedef struct {
   short x, y,
} XPoint;
```

All *x* and *y* fields are 16-bit signed integers. Do not generate coordinates and sizes out of the 16-bit range because the protocol has only 16-bit fields for these values.

Related Information

The **XDrawPoint** subroutine, **XDrawPoints** subroutine.

XArc Data Structure

```
typedef struct {
    short x, y;
    unsigned short width, height;
    short angle1, angle2;           /* Degrees multiplied by 64 */
} XArc;
```

All *x* and *y* fields are 16-bit signed integers. The *width* and *height* fields are 16-bit unsigned integers. Your application should not generate coordinates and sizes out of the 16-bit ranges because the protocol has only 16-bit fields for these values.

Related Information

The **XDrawArc** subroutine, **XDrawArcs** subroutine.

XCharStruct Data Structure

```
typedef struct {
   short lbearing;              /* origin to left edge of raster */
   short rbearing;              /* origin to right edge of raster */
   short width;                 /* advance to next character's
                                   original */
   short ascent;                /* baseline to top edge of raster */
   short descent;               /* baseline to bottom edge of
                                   raster */
   unsigned short attributes;  /* per character flags (not
                                   predefined) */
} XCharStruct;
```

The **XCharStruct** data structure defines the bounding box of a single character, a string, or the overall characteristics of a font. A nonexistent character is represented with all fields of the **XCharStruct** data structure set to the value of **0**. Any of the fields of the **XCharStruct** data structure can be negative.

The fields of the **XCharStruct** data structure are defined as follows:

lbearing	Specifies the extent of the left edge of the character ink from the origin.
rbearing	Specifies the extent of the right edge of the character ink from the origin.
width	Specifies the logical width of the character. If the *width* field is negative, the next character is placed to the left of the current origin.
ascent	Specifies the extent of the top edge of the character ink from the origin.
descent	Specifies the extent of the bottom edge of the character ink from the origin. If the baseline is at the y coordinate *y*, the logical extent of the font is inclusive between the y coordinate values (`y – font.ascent`) and (`y + font.descent – 1`). Typically, the minimum interline spacing between rows of text is given by ascent + descent.

For a character origin at [x, y], the bounding box of a character in terms of **XCharStruct** components, is a rectangle:

The upper-left corner is:

```
    [x + lbearing, y – ascent]
```

The width is:

```
    rbearing – lbearing
```

The height is:

```
    ascent + descent
```

The origin for the next character is defined as:

```
    [x + width, y]
```

The baseline is logically viewed as being immediately below non-descending characters. When the *descent* field is zero, only pixels with y coordinates less than *y* are drawn. The origin is logically viewed as being concident with the left edge of a non-kerned character.

When `lbearing` is the value of **0**, no pixels with the X-coordinate less than *x* are drawn. Any of these values can be negative.

attributes Specifies the per character flags. The X protocol does not define the interpretation of the *attributes* field.

The baseline (the y position of the character origin) is logically viewed as being the scan line just below nondescending characters. When *descent* is 0, only pixels with Y coordinates less than y are drawn, and the origin is logically viewed as being coincident with the left edge of a nonkerned character. When *lbearing* is zero, no pixels with X coordinates less than x are drawn. Any of the **XCharStruct** metric members could be negative. If the *width* is negative, the next character will be placed to the left of the current origin. The X protocol does not define the interpretation of the *attributes* member in the **XCharStruct** structures. A nonexistent character is represented with all members of its **XCharStruct** structure set to zero.

XFontProp Data Structure

```
typedef struct {
    Atom name;
   unsigned long card32;
} XFontProp;
```

The **XFontProp** data structure describes a font property. A pointer to a list of these properties is included in the **XFontStruct** data structure.

XChar2b Data Structure

```
typedef struct {         /* normal 16-bit characters are 2 bytes */
   unsigned char byte1;
   unsigned char byte2;
} XChar2b;
```

The **XChar2b** data structure is used in the **Xlib** library subroutines that use 2-byte matrix fonts.

Enhanced X-Windows supports both single byte per character and two bytes per character text operations. Either form can be used with a font, but a single byte per character text request can specify a single byte only, that is, the first row of a two-byte font. A two-byte font is similar in concept to a two-dimensional matrix of defined characters.

byte1	Specifies the range of defined rows.
byte2	Defines the range of defined columns of the font.

Single byte per character fonts have no rows defined. The *byte2* range specified in the structure defines a range of characters.

XFontStruct Data Structure

The **XFontStruct** data structure contains all the information for the font, including font specific information and a pointer to an array of **XCharStruct** data structures for the characters contained in the font. If characters are undefined or nonexistent, the *default_char* field is used. If the font has characters all the same size, only the information in the *min_bounds* and *max_bounds* fields is used.

```
typedef struct {
    XExtData *Ext_data;          /* hook for extension to hang
                                    data*/
    Font fid;                    /* Font ID for this font */
    unsigned direction;          /* hint about the direction font
                                    is painted */
    unsigned min_char_or_byte2;  /* first character */
    unsigned max_char_or_byte2;  /* last character */
    unsigned min_byte1;          /* first row that exists */
    unsigned max_byte1;          /* last row that exists */
    Bool all_chars_exist;        /* flag if all characters have
                                    nonzero size */
    unsigned default_char;       /* char to print for undefined
                                    character */
    int n_properties;            /* how many properties there are */
    XFontProp *properties;       /* pointer to array of additional
                                    properties */
    XCharStruct min_bounds;      /* minimum bounds over all
                                    existing char */
    XCharStruct max_bounds;      /* maximum bounds over all
                                    existing char */
    XCharStruct *per_char;       /* first char to last char
                                    information */
    int ascent;                  /* logical extent above baseline
                                    for spacing */
    int descent;                 /* logical decent below baseline
                                    for spacing */
} XFontStruct;
```

The fields of the **XFontStruct** data structure include the following:

direction	Provides a hint as to what most **XCharStruct** elements have in terms of character-width metric. The Core protocol does not support vertical text. The *direction* field can be set to one of the following:

FontLeftToRight	Specifies a positive character-width metric.
FontRightToLeft	Specifies a negative character-width metric.

min_byte1	Indicates the first row that exists.
max_byte1	Indicates the last row that exists.
min_char_or_byte2	Specifies the linear character index of the last element. If the *min_byte1* and *max_byte1* fields are both the value of **0**, then the *max_char_or_byte2* field specifies the linear character index corresponding to the first element of the *per_char* field array. If either the *min_byte1* or *max_byte1* field is nonzero, then both the *min_char_or_byte2* and *max_char_or_byte2* fields are less than 256, and the two-byte character index values corresponding to the *per_char* field array element *N* (counting from 0) are:

```
byte1 = N/D + min_byte1
byte2 = N\D + min_char_or_byte2
```

where:

```
D = max_char_or_byte2 – min_char_or_byte2 + 1
/ = integer division
\ = integer modulus
```

per_char	Specifies information about each character. If the *per_char* field is the value of **NULL**, all glyphs between the first and last character, indexes inclusive, have the same information as given by both the *min_bounds* and *max_bounds* fields.
all_chars_exist	If the *all_chars_exist* field is the value of **True**, all characters in the *per_char* field array have nonzero bounding boxes.
default_char	Specifies the character to use when an undefined or nonexistent character is specified by the client. The *default_char* is a 16-bit character. For a font using 2-byte matrix format, the *default_char* has *byte1* in the most-significant byte and *byte2* in the least-significant byte. If the *default_char* specifies an undefined or nonexistent character, no printing is performed.
min_bounds	Specifies the smallest rectangle enclosing the shape obtained by superimposing all of the characters at the same origin [x,y].
max_bounds	Specifies the most extreme values of each individual **XCharStruct** component over all elements of this array which ignores nonexistent characters. The bounding box of the font is defined as follows: The upper-left coordinate is:

```
[x + min_bounds.lbearing, y – max_bounds.ascent]
```

The x and y coordinates are the baseline coordinates of the box, relative to the origin.

The width is:

```
max_bounds.rbearing – min_bounds.lbearing
```

The height is:

```
max_bounds.ascent + max_bounds.descent
```

ascent Specifies the logical extent of the font above the baseline that is used for determining line spacing. Specific characters can extend beyond this.

descent Specifies the logical extent of the font at or below the baseline that is used for determining line spacing. Specific characters may extend beyond this.

The interpretation of the *attributes* field in the **XCharStruct** data structure is not defined by the core protocol. A nonexistent character is represented with members of the **XCharStruct** data structure set to the value of **0**.

A font is not guaranteed to have any properties. The interpretation of the property value, for example, **INT32, CARD32**, must be derived from a prior knowledge of the property. When possible, fonts should have the properties listed in the following table. (Atom names are case-sensitive.) The following built-in property atoms are in **<X11/Xatom.h>**.

Property Name	**Type**	**Description**
MIN_SPACE	unsigned	The minimum interword spacing, specified in pixels.
NORM_SPACE	unsigned	The normal interword spacing, specified in pixels.
MAX_SPACE	unsigned	The maximum interword spacing, specified in pixels.
END_SPACE	unsigned	The additional spacing at the end of sentences, specified in pixels.
SUPERSCRIPT_X SUPERSCRIPT_Y	integers	The offset, specified in pixels from character origin where superscripts should begin. If the origin is at `[x,y]`, then superscripts should begin at `[x + SUPERSCRIPT_X, y – SUPERSCRIPT_Y]`.
SUBSCRIPT_X SUBSCRIPT_Y	integers	Offset, specified in pixels, from character origin where subscripts should begin. If origin is at [x,y], then subscripts should begin at `[x + SUPERSCRIPT_X, y + SUPERSCRIPT_Y]`.
UNDERLINE_POSITION	integer	The y offset, specified in pixels from the baseline to the top of an underline. If the baseline is the y coordinate *y*, then the top of the underline is at `(y + UNDERLINE_POSITION)`.

UNDERLINE_THICKNESS	unsigned	Thickness of the underline, specified in pixels.
STRIKEOUT_ASCENT STRIKEOUT_DESCENT	integers	Vertical extents, specified in pixels, for boxing or voiding characters. If the baseline is at Y-coordinate *y*, then the top of the strikeout box is at (`y – STRIKEOUT_ASCENT`), and the height of the box is (`STRIKEOUT_AS-CENT + STRIKEOUT_DESCENT`).
ITALIC_ANGLE	integer	The angle of the dominant staffs of characters in the font, in degrees scaled by 64, relative to the three o'clock position from the character origin, with positive indicating counterclock-wise motion.
X_HEIGHT	integer	1 *ex* as in TeX, but expressed in pixels. Often the height of lowercase x.
QUAD_WIDTH	integer	1 *em* as in TeX, but expressed in pixels. Often the width of the digits 0-9.
CAP_HEIGHT	integer	The y offset, specified in pixels from the base-line to the top of the capital letters, ignoring accents. If the baseline is at the y coordinate *y*, then the top of the uppercase letters is at (`y – CAP_HEIGHT`).
WEIGHT	unsigned	The weight or boldness of the font, expressed as a value between 0 and 1000.
POINT_SIZE	unsigned	The point size of the font at the ideal resolu-tion, expressed in 1/10ths of points. There are 72.27 points to the inch.
RESOLUTION	unsigned	The number of pixels per point, expressed in 1/100ths, at which the font was created.

Related Information

The **XFreeFont** subroutine, **XFreeFontInfo** subroutine, **XFreeFontNames** subroutine, **XFreeFontPath** subroutine, **XGContextFromGC** subroutine, **XGetFontPath** subroutine, **XGetFontProperty** subroutine, **XListFonts** subroutine, **XListFontsWithInfo** subroutine, **XLoadFont** subroutine, **XLoadQueryFont** subroutine, **XQueryTextExtents** subroutine, **XQueryTextExtents16** subroutine, **XSetFontPath** subroutine, **XTextExtents** subroutine, **XTextExtents16** subroutine, **XTextWidth** subroutine, **XTextWidth16** subroutine, **XUnloadFont** subroutine.

XTextItem Data Structure

```
typedef struct {
    char *chars; /* pointer to string */
   int nchars;  /* number of characters */
   int delta;   /* delta between strings along the x axis */
    Font font;   /* Font to print it in, None does not change */
} XTextItem;
```

If the *font* field is the value of **None**, the font is changed before printing and is stored in the GC. If an error is generated during text drawing, the font in the GC is undefined.

Related Information

The **XDrawText** subroutine

XTextItem16 Data Structure

```
typedef struct {
    XChar2b *chars; /* pointer to two byte characters */
   int nchars;      /* number of characters */
   int delta;       /* delta between strings along the x axis */
    Font font;      /* font to print it in, None does not change */
} XTextItem16;
```

The fields of the **XTextItem16** data structure are as follows:

chars	Specifies a pointer to two-byte characters. The *chars* field of the **XTextItem16** data structure is of type **XChar2b**. The X Server interprets each member of the **XChar2b** structure as a 16-bit number that has been transmitted by most-significant byte first. The *byte1* field of the **XChar2b** structure is taken as the most-significant byte.
nchars	Specifies the number of characters.
delta	Specifies the delta between strings along the x axis.
font	Specifies the font to print it in. If the *font* field is the value of **None**, the font is changed before printing and stored in the **GC**. If an error is generated during text drawing, the font in the **GC** is undefined.

Related Information

The **XDrawText16** subroutine.

XImage Data Structure

```
typedef struct _XImage {
   int width, height;          /* size of image */
   int xoffset;                /* number of pixels offset in X
                                  direction */
   int format;                 /* XYBitmap, XYPixmap, ZPixmap */
   char *data;                 /* pointer to image data */
   int byte_order;             /* data byte order, MSBFirst or
                                  LSBFirst */
   int bitmap_unit;            /* quant. of scanline 8, 16, 32 */
   int bitmap_bit_order;       /* MSBFirst or LSBFirst */
   int bitmap_pad;             /* 8, 16, 32 either XYPixmap or
                                  ZPixmap */
   int depth;                  /* depth of image */
   int bytes_per_line;         /* accelerator to next line */
   int bits_per_pixel;         /* bits per pixel (ZPixmap) */
   unsigned long red_mask;     /* bits in z arrangement */
   unsigned long green_mask;   /* bits in z arrangement */
   unsigned long blue_mask;    /* bits in z arrangement */
   char *obdata;               /* hook for the object routines to
                               /* hang on */
    struct funcs {              /* image manipulation routines */
      struct _XImage *(*create_image)();
      int (*destroy_image)();
      unsigned long (*get_pixel)();
      int (*put_pixel)();
      struct _XImage *(*sub_image)();
      int (*add_pixel)();
   } f;

} XImage;
```

The **XImage** data structure describes an image as it exists in client memory. You can request changes to some fields in this data structure, for example, *height*, *width*, and *xoffset*. These changes create a subset of the image. Other fields of this structure, for example, byte_order and bitmap_unit, are characteristic of both the image and the server. If these fields differ between the image and the server, **XPutImage** makes the appropriate conversions.

If the image is formatted as an **XYPixmap**, the first byte of the first line of plane *n* must be located at the address of the client as follows:

```
(data + (n * height * bytes_per_line)).
```

Related Information

The **XAddPixel** subroutine, **XCreateImage** subroutine, **XDestroyImage** subroutine, **XGetImage** subroutine, **XGetPixel** subroutine, **XGetSubImage** subroutine, **XPutImage** subroutine, **XPutPixel** subroutine, **XSubImage** subroutine.

XKeyboardControl Data Structure

```
#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)


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 fields of the **XKeyboardControl** data structure include the following values:

key_click_percent	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 an error.
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 an error.
bell_pitch	Sets the pitch, specified in hertz (Hz) of the bell, if possible. A setting of -1 restores the default. Other negative values generate an error.
bell_duration	Sets the duration of the bell (in milliseconds), if possible. A setting of -1 restores the default. Other negative values generate an error.
led	Specifies the LED member. If the *led_mode* and the *led* fields are specified, the state of those LEDs is changed, if possible. This occurs where the *led* field is the ordinal number of the LED to be changed and not a mask.
led_mode	The state of all LEDs is changed, if possible. At most 32 LEDs, numbered from 1, are supported. If an LED is specified without *led_mode*, a **BadMatch** error is generated. No standard interpretation of LEDs is defined.
key	Specifies a key on the keyboard.
auto_repeat_mode	Specifies the auto repeat mode. If the *auto_repeat_mode* and the *key* fields are specified, the *auto_repeat_mode* of that key is changed, if possible. If only the *auto_repeat_mode* field 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 the *auto_repeat_mode* field, a **BadMatch** error is generated.

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.

XKeyboardState Data Structure

```
typedef struct {
    int key_click_percent;
   int bell_percent;
   unsigned int bell_pitch, bell_duration;
   unsigned long led_mask;
   int global_auto_repeat;
   char auto_repeats[32];
} XKeyboardState;
```

For the LEDs, the least-significant bit of the *led_mask* field corresponds to LED 1, and each bit in *led_mask* that is set to 1 indicates an LED that is lit.

The *auto_repeats* field is a bit vector. Each bit indicates that auto-repeat is enabled for the corresponding key. The vector is represented as 32 bytes. Byte N (from 0) contains the bits for keys 8N to 8N+7, with the least-significant bit in the byte representing key 8N.

The *global_auto_repeat* field can be set to the value of **AutoRepeatModeOn** or **AutoRepeatModeOff**.

Related Information

The **XAutoRepeatOn** subroutine, **XBell** subroutine, **XChangeKeyboardControl** subroutine, **XGetKeyboardControl** subroutine, **XGetPointerMapping** subroutine, **XQueryKeymap** subroutine, **XSetPointerMapping** subroutine.

XModifierKeymap Data Structure

```
typedef struct {
   int max_keypermod;              /* Max number of keys per
                                      modifier of this server */
   KeyCode *modifiermap;           /* An 8 by max_keypermod array
                                    of the modifiers */
} XModifierKeymap;
```

Related Information

The **<X11/keysym.h>** header file, **<X11/keysymdef.h>** header file

The **XChangeKeyboardMapping** subroutine, **XDeleteModifiermapEntry** subroutine, **XFreeModifiermap** subroutine, **XGetKeyboardMapping** subroutine, **XGetModifierMapping** subroutine, **XInsertmodifiermapEntry** subroutine, **XLookupString** subroutine, **XNewModifiermap** subroutine, **XSetModifierMapping** subroutine.

XHostAddress Data Structure

```
typedef struct {
   int family;        /* for example FamilyInternet */
   int length;        /* length of address, in bytes */
   char *address;     /* pointer to where to find the address */
} XHostAddress;
```

The fields of the **XHostAddress** data structure are:

family	Specifies which protocol address family to use (for example, the TCP/IP or UNIX domain). The family symbols are defined in the **<X11/X.h>** header file.
length	Specifies the length of the address in bytes.
address	Specifies a pointer to the address.

Related Information

The **<X11/X.h>** header file, **/etc/X?.hosts** file.

The **XAddHost** subroutine, **XAddHosts** subroutine, **XListHosts** subroutine, **XRemoveHost** subroutine, **XRemoveHosts** subroutine.

XAnyEvent Data Structure

For each event type, a corresponding structure is declared in the **<X11/Xlib.h>** header file.

```
typedef struct {
   int type;
   unsigned long serial;      /* Number of last request processed
                                 by the server */
   Bool send_event;           /* True if this came from a SendEvent
                                 request */
   Display *display;          /* Display the event was read from */
   Window window;
} XAnyEvent:
```

All the event structures have the following common fields.

type	Set to the event type constant name that uniquely identifies the event type. For example, when the X Server reports a **GraphicsExpose** event to a client application, the event sends an **XGraphicsExposeEvent** structure with the *type* member set to **GraphicsExpose**.
display	Set to a pointer to the display the event was read on.
send_event	Set to the value of **TRUE** if the event was generated by an **XSendEvent** request.
serial	Set to the serial number reported in the protocol but expanded from the 16-bit least significant bits to a full 32-bit value.

Related Information

The **<X11/Xlib.h>** header file.

XEvent Data Structure

```
typedef union _XEvent {
   int type                    /* Must not be changed */
   XAnyEvent xany;
   XKeyEvent xkey;
   XButtonEvent xbutton;
   XMotionEvent xmotion;
   XCrossingEvent xcrossing;
   XFocusChangeEvent xfocus;
   XExposeEvent xexpose;
   XGraphicsExposeEvent xgraphicsexpose;
   XNoExposeEvent xnoexpose;
   XVisibilityEvent xvisibility;
   XCreateWindowEvent xcreatewindow;
   XDestroyWindowEvent xdestroywindow;
   XUnmapEvent xunmap;
   XMapEvent xmap;
   XMapRequestEvent xmaprequest;
   XReparentEvent xreparent;
   XConfigureEvent xconfigure;
   XGravityEvent xgravity;
   XResizeRequestEvent xresizerequest;
   XConfigureRequestEvent xconfigurerequest;
   XCirculateEvent xcirculate;
   XCirculateRequestEvent xcirculaterequest;
   XPropertyEvent xproperty;
   XSelectionClearEvent xselectionclear;
   XSelectionRequestEvent xselectionrequest;
   XSelectionEvent xselection;
   XColormapEvent xcolormap;
   XClientMessageEvent xclient;
   XMappingEvent xmapping;
   XErrorEvent xerror;
   XKeymapEvent xkeymap;
   long pad[24];
} XEvent;
```

Related Information

The **<X11/Xlib.h>** header file.

XButtonPressedEvent or **XButtonReleasedEvent** Data Structure

```
typedef struct {
   int type;                  /* ButtonPress or ButtonRelease */
   unsigned long serial;      /* Number of the last request
                                 processed by the server */
   Bool send_event;           /* True if this came from a
                                 SendEvent request */
   Display *display;          /* The display the event was read
                                 from */
   Window window;             /* The event window it is reported
                                 relative to */
   Window root;               /* Root window that the event
                                 occurred on */
   Window subwindow;          /* The child window */
   Time time;                 /* Milliseconds */
   int x, y;                  /* Pointer x, y coordinates in the
                                 event window */
   int x_root, y_root;        /* Coordinates relative to the root
                                 window */
   unsigned int state;        /* Key or button mask */
   unsigned int button;       /* Detail */
   Bool same_screen;          /* Same screen flag */
} XButtonEvent;
typedef XButtonEvent XButtonPressedEvent;
typedef XButtonEvent XButtonReleasedEvent;
```

The fields for these structures are defined as follows:

type	Set to the event type constant name that uniquely identifies the event type. For example, when the X Server reports a **GraphicsExpose** event to a client application, the event sends an **XGraphicsExposeEvent** structure with the *type* field set to **GraphicsExpose**.
display	Set to a pointer to the display the event was read on.
send_event	Set to the value of **TRUE** if the event was generated by an **XSendEvent** request.
serial	Set to the serial number reported in the protocol but expanded from the 16-bit least significant bits to a full 32-bit value.
window	The window ID of the window on which the event was generated. This is the event window. The X Server uses this window to report the event.
root	The window ID of the root window of the source.
x_root	This is set to the x pointer coordinate relative to the origin of the root window at the time of the event.

y_root

This is set to the y pointer coordinate relative to the origin of the root window at the time of the event.

same_screen

Indicates if the event window is on the same screen as the root window. This parameter can be:

TRUE	If the event and root windows are on the same screen.
FALSE	If the event and root windows are not on the same screen.

subwindow

Can be one of the following:

- The child of the event window that is an ancestor of or is the source member, if the event window is on the same screen as the root window.
- Otherwise, the *subwindow* is the value of **None.**

x

Can be one of the following:

- If the event window is on the same screen as the root window, the x coordinate is set to the coordinate relative to the event window's origin
- Otherwise, *X* is the value of **0**.

y

Can be one of the following:

- If the event window is on the same screen as the root window, the y coordinate is set to the coordinate relative to the event window's origin
- Otherwise, *Y* is the value of **0**.

time

The time that the event was generated. The time is expressed in milliseconds since the server reset.

state

Indicates the state of the pointer buttons and modifier keys just prior to the event. The X Server can set this member to the bitwise-inclusive OR of one or more of the following button or modifier key masks:

Button1Mask	**ShiftMask**	**Mod1Mask**
Button2Mask	**LockMask**	**Mod2Mask**
Button3Mask	**ControlMask**	**Mod3Mask**
Button4Mask		**Mod4Mask**
Button5Mask		**Mod5Mask**

button

This represents the pointer buttons that changed state for the **XButtonPressedEvent** and **XButtonReleasedEvent** data structures, and can be set to one or more of the following button names: **Button1, Button2, Button3, Button4, Button5.**

Related Information

The **ButtonPress** protocol event, **ButtonRelease** protocol event.

XKeyPressedEvent or XKeyReleasedEvent Data Structure

```
typedef struct {
   int type;                 /* KeyPress or KeyRelease */
   unsigned long serial;     /* Number of the last request
                                processed by the server */
   Bool send_event;          /* True if this came from a SendEvent
                                request */
   Display *display;         /* The display the event was read
                                from */
   Window window;            /* The event window it is reported
                                relative to */
   Window root;              /* Root window that the event
                                occurred on */
   Window subwindow;         /* The child window */
   Time time;                /* Milliseconds */
   int x, y;                 /* Pointer x, y coordinates in the
                                event window */
   int x_root, y_root;       /* Coordinates relative to the root
                                window */
   unsigned int state;       /* Key or button mask */
   unsigned int keycode;     /* Detail */
   Bool same_screen;         /* Same screen flag */
} XKeyEvent;
typedef XKeyEvent XKeyPressedEvent;
typedef XKeyEvent XKeyReleasedEvent;
```

The fields for these structures are defined as follows:

type	Set to the event type constant name that uniquely identifies the event type. For example, when the X Server reports a **GraphicsExpose** event to a client application, the event sends an **XGraphicsExposeEvent** data structure with the *type* field set to **GraphicsExpose**.
display	Set to a pointer to the display the event was read on.
send_event	Set to the value of **TRUE** if the event was generated by an **XSendEvent** request.
serial	Set to the serial number reported in the protocol but expanded from the 16-bit least significant bits to a full 32-bit value.
window	The window ID of the window on which the event was generated. This is the event window. The X Server uses this window to report the event.
root	The window ID of the root window of the source.
x_root	This is set to the x pointer coordinate relative to the origin of the root window at the time of the event.
y_root	This is set to the y pointer coordinate relative to the origin of the root window at the time of the event.

same_screen	Indicates if the event window is on the same screen as the root window. This field can be either of the following values: **TRUE** If the event and root windows are on the same screen. **FALSE** If the event and root windows are not on the same screen.
subwindow	Can be one of the following: • The child of the event window that is an ancestor of or is the source member, if the event window is on the same screen as the root window. • Otherwise, the *subwindow* field has the value of **None**.
x	Can be one of the following: • The x coordinate relative to the origin of the event window if the root window is on the same screen as the event window. • Otherwise, the *x* field has the value of **0**.
y	Can be one of the following: • The y coordinate relative to the origin of the event window if the root window is on the same screen as the event window. • Otherwise, the *y* field has the value of **0**.
time	The time that the event was generated. The time is expressed in milliseconds since the server reset.
state	Indicates the state of the pointer buttons and modifier keys just prior to the event. The X Server can set this member to the bitwise include OR of one or more of the following button or modifier key masks:

Button1Mask	**ShiftMask**	**Mod1Mask**
Button2Mask	**LockMask**	**Mod2Mask**
Button3Mask	**ControlMask**	**Mod3Mask**
Button4Mask		**Mod4Mask**
Button5Mask		**Mod5Mask**

keycode	This is set to a number that represents a physical key on the keyboard for the **XKeyPressedEvent** and **XKeyReleasedEvent** data structures.

Related Information

The **KeyPress** event, **KeyRelease** event.

XPointerMovedEvent Data Structure

```
typedef struct {
   int type;                    /* MotionNotify */
   unsigned long serial;        /* Number of the last request
                                   processed by the server */
   Bool send_event;             /* True if this came from a
                                   SendEvent request */
   Display *display;            /* The display the event was read
                                   from */
   Window window;               /* The event window it is reported
                                   relative to */
   Window root;                 /* Root window that the event
                                   occurred on */
   Window subwindow;            /* The child window */
   Time time;                   /* Milliseconds */
   int x, y;                    /* Pointer x, y coordinates in the
                                   event window */
   int x_root, y_root;          /* Coordinates relative to the root
                                   window */
   unsigned int state;          /* Key or button mask */
   unsigned int keycode;        /* Detail */
   char is_hint;                /* Detail */
   Bool same_screen;            /* Same screen flag */
} XMotionEvent;
typedef XMotionEvent XPointerMovedEvent;
```

The fields for the **XPointerMovedEvent** data structure are defined as follows:

type	Set to the event type constant name that uniquely identifies the event type. For example, when the X Server reports a **GraphicsExpose** event to a client application, the event sends an **XGraphicsExposeEvent** structure with the *type* field set to the value of **GraphicsExpose**.
display	Set to a pointer to the display the event was read on.
send_event	Set to the value of **TRUE** if the event was generated by an **XSendEvent** request.
serial	Set to the serial number reported in the protocol but expanded from the 16-bit least significant bits to a full 32-bit value.
window	The window ID of the window on which the event was generated. This is the event window. The X Server uses this window to report the event.
root	The window ID of the root window of the source.
x_root	This is set to the x pointer coordinate relative to the origin of the root window at the time of the event.
y_root	This is set to the y pointer coordinate relative to the origin of the root window at the time of the event.

same_screen	Indicates if the event window is on the same screen as the root window. This field can be either of the following values: **TRUE** If the event and root windows are on the same screen. **FALSE** If the event and root windows are not on the same screen.
subwindow	Can be one of the following: • The child of the event window that is an ancestor of or is the source member, if the event window is on the same screen as the root window. • Otherwise, the *subwindow* field has the value of **None**.
x	Can be one of the following: • The x coordinate relative to the origin of the event window if the source window and the event window are on the same screen. • Otherwise, the *x* field has the value of **0**.
y	Can be one of the following: • The y coordinate relative to the origin of the event window if the source window and the event window are on the same screen. • Otherwise, the *y* field has the value of **0**.
time	The time that the event was generated. The time is expressed in milliseconds since the server reset.
state	Indicates the state of the pointer buttons and modifier keys just prior to the event. The X Server can set this member to the bitwise-inclusive OR of one or more of the following button or modifier key masks:

Button1Mask	**ShiftMask**	**Mod1Mask**
Button2Mask	**LockMask**	**Mod2Mask**
Button3Mask	**ControlMask**	**Mod3Mask**
Button4Mask		**Mod4Mask**
Button5Mask		**Mod5Mask**

is_hint This can be set to the value of **NotifyNormal** or **NotifyHint** for the **XPointerMovedEvent** data structure.

Related Information

The **MotionNotify** event.

XCrossingEvent or XEnterWindowEvent or XLeaveWindowEvent Data Structures

```
typedef struct {
   int type;                    /* EnterNotify or LeaveNotify */
   unsigned long serial;        /* Number of the last request
                                   processed by the server */
    Bool send_event;             /* True if this came from a
                                   SendEvent request */
    Display *display;            /* The display the event was read
                                   from */
    Window window;               /* The event window it is
                                   reported relative to */
   Window root;                 /* Root window that the event
                                   occurred on */
    Window subwindow;            /* The child window */
    Time time;                   /* Milliseconds */
   int x, y;                    /* Pointer x, y coordinates in
                                   the event window */
   int x_root, y_root;          /* Coordinates relative to the
                                   root window */
   int mode;                    /* NotifyNormal, NotifyGrab,
                                   NotifyUngrab */
   int detail;                  /* NotifyAncestor, NotifyVirtual,
                                   NotifyInferior,
                                   NotifyNonlinear,
                                   NotifyNonlinearVirtual */
    Bool same_screen;            /* Same screen flag */
    Bool focus;                  /* Boolean focus */
   unsigned int state;          /* Key or button mask */
} XCrossingEvent;
typedef XCrossingEvent XEnterWindowEvent;
typedef XCrossingEvent XLeaveWindowEvent;
```

The fields of the **XCrossingEvent**, **XEnterWindowEvent**, and **XLeaveWindowEvent** data structures are as follows:

type	Set to the event type constant name that uniquely identifies the event type. For example, when the X Server reports a **GraphicsExpose** event to a client application, the event sends an **XGraphicsExposeEvent** structure with the *type* field set to **GraphicsExpose**.
display	Set to a pointer to the display the event was read on.
send_event	Set to the value of **TRUE** if the event was generated by an **XSendEvent** request.
serial	Set to the serial number reported in the protocol but expanded from the 16-bit least significant bits to a full 32-bit value.
window	The window ID of the window on which the **EnterNotify** or **LeaveNotify** event was generated. This window is the event window. The X Server uses this window to report the events.

root	The ID of the root window on which the event occurred.
subwindow	In a **LeaveNotify** event, if a child of the event window contains the initial position of the pointer, the *subwindow* field is set to that child. Otherwise, the X Server sets the *subwindow* field to the value of **None**.
subwindow	In an **EnterNotify** event, if a child of the event window contains the final pointer position, the *subwindow* is set to that child. Otherwise, it is set to the value of **None**.
time	The time (in milliseconds) the event was generated.
x	The x pointer position in the event window. This position is always the final position of the pointer, not the initial position of the pointer.
y	The y pointer position in the event window. This position is always the final position of the pointer, not the initial position of the pointer. If the event window is on the same screen as the root window, x and y are the pointer coordinates relative to the origin of the event window. Otherwise, the *x* and *y* fields are set to the value of **0**.
x_root	Set to the x pointer coordinate relative to the origin of the root window at the time of the event.
y_root	Set to the y pointer coordinate relative to the origin of the root window at the time of the event.
same_screen	Indicates if the event window is on the same screen as the root window. The *same_screen* field can be either of the following values:

True	The event and root windows are on the same screen.
False	The event and root windows are not on the same screen.

focus	Indicates whether the event window is the focus window or an inferior of the focus window. The *focus* field can be either of the following values:

True	The event window is the focus window or an inferior of the focus window.
False	The event window is neither the focus window nor an inferior of the focus window.

state	Indicates the state of the pointer buttons and modifier keys immediately prior the event. The X Server can set this field to the bitwise-inclusive OR of one or more of the following button or modifier key mask values.

Button1Mask	**ShiftMask**	**Mod1Mask**
Button2Mask	**LockMask**	**Mod2Mask**
Button3Mask	**ControlMask**	**Mod3Mask**
Button4Mask		**Mod4Mask**
Button5Mask		**Mod5Mask**

mode	Indicates if the events are normal events or pseudo motion events when a grab activates, or when a grab deactivates. The X Server can set the *mode* field to one of the following values: **NotifyNormal** **NotifyGrab** **NotifyUngrab**.
detail	Indicates the notify detail can be set to one of the following values: **NotifyAncestor** **NotifyVirtual** **NotifyInferior** **NotifyNonlinear** **NotifyNonlinearVirtual**

Related Information

The **EnterNotify** event, **LeaveNotify** event.

The **XChangeActivePointerGrab** subroutine, **XGrabKeyboard** subroutine, **XGrabPointer** subroutine, **XUngrabPointer** subroutine.

XFocusChangeEvent or XFocusInEvent or XFocusOutEvent Data Structures

```
typedef struct {
   int type;                  /* FocusIn or FocusOut */
   unsigned long serial;      /* Number of the last request
                                 processed by the server */
   Bool send_event;           /* True if this came from a
                                 SendEvent request */
    Display *display;          /* The display the event was read
                                 from */
    Window window;             /* The window of the event */
   int mode;                  /* NotifyNormal, NotifyGrab,
                                 NotifyUngrab */
   int detail;                /* NotifyAncestor, NotifyVirtual,
                                 NotifyInferior,
                                 NotifyNonlinear,
                                 NotifyNonlinearVirtual,
                                 NotifyPointer,
                                 NotifyPointerRoot,
                                 NotifyDetailNone */
} XFocusChangeEvent;
typedef XFocusChangeEvent XFocusInEvent;
typedef XFocusChangeEvent XFocusOutEvent;
```

The fields of the **XFocusChange**, **XFocusInEvent** and **XFocusOutEvent** data structures include the following definitions:

window	Specifies the window ID of the window on which the **FocusIn** or **FocusOut** event was generated. The X Server uses this window to report the event.

mode	Specifies the type of focus event. The *mode* field can be set to one of the following values:

NotifyNormal	Specifies a normal focus event.
NotifyWhileGrabbed	Specifies a focus event while grabbed.
NotifyGrab	Specifies a focus event when a grab activates.
NotifyUngrab	Specifies a focus event when a grab deactivates.

detail	Indicates the notify detail depending on the event mode. The *detail* field can be one of the following values:

NotifyAncestor	**NotifyVirtual**
NotifyInferior	**NotifyNonlinear**
NotifyNonlinearVirtual	**NotifyPointer**
NotifyPointerRoot	**NotifyDetailNone**

All **FocusOut** events caused by a window unmap are generated after any **UnmapNotify** event, but the ordering of **FocusOut** events with respect to generated **EnterNotify**, **LeaveNotify**, **VisibilityNotify**, and **Expose** events is not constrained by the X protocol.

Related Information

The **FocusIn** event, **FocusOut** event.

The **XGrabKeyboard** subroutine, **XUngrabKeyboard** subroutine.

XKeymapEvent Data Structure

```
typedef struct {
    int type;                    /* KeymapNotify */
    unsigned long serial;        /* Number of the last request
                                    processed by the server */
    Bool send_event;             /* True if this came from a
                                    SendEvent request */
    Display *display;            /* The display the event was
                                    read from */
    Window window;
    char key_vector[32];
} XKeymapEvent;
```

The fields of the **XKeymapEvent** data structure associated with this event include the following:

window	This is not used, but present for use with toolkit operations.
key_vector	Specifies the bit vector of the keyboard. Each bit indicates that the corresponding key is currently pressed. The vector is represented as 32 bytes. `Byte N` (from 0) contains the bits for keys `8N to 8N+7` with the least significant bit in the byte representing key `8N`.

Related Information

The **KeymapNotify** event.

XExposeEvent Data Structure

```
typedef struct {
   int type;                     /* Expose */
   unsigned long serial;      /* Number of the last request
                                 processed by the server */
   Bool send_event;           /* True if this came from a
                                 SendEvent request */
   Display *display;          /* The display the event was read
                                 from */
   Window window;
   int x, y;
   int width, height:
   int count;                 /* If nonzero, at least this many
                                 more */
} XExposeEvent;
```

The fields of the **XExposeEvent** data structure include the following:

window	The window ID of the exposed (damaged) window.
x	Indicates the x coordinate of the rectangle. This coordinate is set relative to the origin of the drawable.
y	Indicates the y coordinate of the rectangle. This coordinate is set relative to the origin of the drawable.
width	Specifies the width extent of the rectangle.
height	Specifies the height extent of the rectangle.
count	Specifies the number of **Expose** events that should follow. The *count* field can be:

0	No **Expose** events will follow. (Applications not designed to optimize redisplay by distinguishing between subareas of a window redisplay entirely if the *count* field is the value of **0**.)
nonzero	At least that number, and possibly more, **Expose** events will follow. (Applications not designed to optimize redisplay by distinguishing between subareas of a window, do not respond if the *count* field is a nonzero value.)

Related Information

The **Expose** event.

XGraphicsExposeEvent Data Structure

```
typedef struct {
   int type;                    /* GraphicsExpose */
   unsigned long serial;        /* Number of the last request
                                   processed by the server */
    Bool send_event;             /* True if this came from a
                                    SendEvent request */
    Display *display;            /* The display the event was read
                                   from */
    Drawable drawable;
   int x, y;
    int width, height:
    int count;                   /* If nonzero, at least this many
                                   more */
    int major_code;              /* core is CopyArea or CopyPlane */
    int minor_code;              /* Not defined in the core */
} XGraphicsExposeEvent;
```

The **XGraphicsExposeEvent** data structure included the following fields:

drawable	Specifies the drawable ID of the destination region on which the graphics request is to be performed.
major_code	Specifies the graphics request initiated by the client This field can have one of the following values:

X_CopyArea	Indicates that a call to the **XCopyArea** subroutine initiated the request.
X_CopyPlane	Indicates that a call to the **XCopyPlane** subroutine initiated the request.

These constants are defined in the **<X11/Xproto.h>** file.

minor_code	Specifies the graphics request initiated by the client. This field, however, is not defined by the core X protocol and will have the value of **0** in these cases, although it may be used as an extension.
x	Specifies the x coordinate of the upper left corner of the rectangle. This coordinate is relative to the origin of the drawable.
y	Specifies the y coordinate of the upper left corner of the rectangle. This coordinate is relative to the origin of the drawable.
width	Specifies the size (extent) of the rectangle.
height	Specifies the size (extent) of the rectangle.
count	Specifies the number of **GraphicsExpose** events to follow for the specified window. This field can have the following values:

0	Indicates that no **GraphicsExpose** events will follow.

nonzero	Indicates that at least that number, and possibly more, **GraphicsExpose** events will follow.

Related Information

The **GraphicsExposure** event, **NoExpose** event.

The **XCopyArea** subroutine, **XCopyPlane** subroutine, **XCreateGC** subroutine, **XSetGraphicsExposures** subroutine.

XNoExposeEvent Data Structure

```
typedef struct {
   int type;                 /* NoExpose */
   unsigned long serial;     /* Number of the last request
                                processed by the server */
    Bool send_event;          /* True if this came from a SendEvent
                                request */
    Display *display;         /* The display the event was read
                              from */
   Drawable drawable;
   int major_code;           /* core is CopyArea or CopyPlane */
   int minor_code;           /* Not defined in the core */
} XNoExposeEvent;
```

The **XGraphicsExposeEvent** and **XNoExposeEvent** data structures have the following common fields:

drawable — Specifies the drawable ID of the destination region on which the graphics request is to be performed.

major_code — Specifies the graphics request initiated by the client The *major_code* field can have either of the following values:

X_CopyArea	Indicates that a call to the **XCopyArea** subroutine initiated the request.
X_CopyPlane	Indicates that a call to the **XCopyPlane** subroutine initiated the request.

minor_code — Specifies the graphics request initiated by the client. The *minor_code* field, however, is not defined by the core X protocol and will have the value of **0** in these cases, although it may be used as an extension.

Related Information

The **<X11/Xproto.h>** file.

The **GraphicsExpose** event, **NoExpose** event.

The **XCopyArea** subroutine, **XCopyPlane** subroutine, **XCreateGC** subroutine, **XSetGraphicsExposures** subroutine.

XCirculateEvent Data Structure

```
typedef struct {
   int type;               /* CirculateNotify */
   unsigned long serial;   /* Number of last request processed
                              by the server */
   Bool send_event;        /* True if this came from a SendEvent
                              request */
    Display *display;       /* The display the event was read
                              from */
    Window event;
    Window window;
   int place;              /* PlaceOnTop, PlaceOnBottom */
} XCirculateEvent;
```

The **XCirculateEvent** data structure includes the following fields:

type	Specifies the event type, **CirculateNotify.**
serial	Specifies the number of the last request processed by the server.
send_event	Specifies **True** if this came from a SendEvent request.
display	Specifies the display that the event was read from.
event	Specifies the window on which the event was generated. This field is set to either the restacked window or its parent, depending on whether **StructureNotify** or **SubstructureNotify** was selected.
window	Specifies the window that was restacked.
place	Specifies the window position after the restack occurs. The *place* field can have either of the following values:

PlaceOnTop	Indicates that the window is now on top of all siblings.
PlaceOnBottom	Indicates that the window is now below all siblings.

Related Information

The **CirculateNotify** event.

The **XCirculateSubwindows** subroutine, **XCirculateSubwindowsDown** subroutine, **XCirculateSubwindowsUp** subroutine.

XConfigureEvent Data Structure

```
typedef struct {
   int type;                    /* ConfigureNotify */
   unsigned long serial;        /* Number of the last request
                                   processed by the server */
    Bool send_event;             /* True if this came from a
                                   SendEvent request */
    Display *display;            /* The display the event was read
                                   from */
    Window event:
    Window window;
   int x, y;
   int width, height;
    int border_width;
    Window above;
    Bool override_redirect;
} XConfigureEvent;
```

The **XConfigureEvent** data structure includes the following fields:

event	Specifies the window on which the event was genereated. This field is set to either the reconfigured window or its parent, depending on whether **StructureNotify** or **SubstructureNotify** was selected.
window	Specifies the window whose size, position, border, or stacking order was changed.
x	Specifies the x coordinate of the upper left corner of the window. This coordinate is relative to the origin of the new parent window.
y	Specifies the y coordinate of the upper left corner of the window. This coordinate is relative to the origin of the new parent window
width	Specifies the size of the window, excluding the border.
height	Specifies the size of the window, excluding the border.
border_width	Specifies the width of the window border, in pixels.
above	Specifies the window ID of the sibling window. This field is used for stacking operations. If the *above* field is set to the value of **None**, the reconfigured window is on the bottom of the stack with respect to sibling windows. If this field is set to a sibling window, the reconfigured window is placed on top of this sibling window.
override_redirect	Specifies the value set in the *override_redirect* field of the window. If this field is the value of **True**, window manager clients should ignore the window.

Related Information

The **ConfigureNotify** event.

The **XConfigureWindow** subroutine, **XLowerWindow** subroutine, **XRaiseWindow** subroutine, **XRestackWindows** subroutine, **XMoveWindow** subroutine, **XResizeWindow** subroutine, **XMoveResizeWindow** subroutine, **XMapRaised** subroutine, **XSetWindowBorderWidth** subroutine.

XCreateWindowEvent Data Structure

The **XCreateWindowEvent** data structure includes the following fields:

parent	Specifies the parent of the created window.
window	Specifies the created window.
x	Specifies the x coordinate of the upper left outside corner of the created window. This coordinate is relative to the inside of the borders of the parent window.
y	Specifies the y coordinate of the upper left outside corner of the created window. This coordinate is relative to the inside of the borders of the parent window.
width	Specifies the inside size of the created window, excluding the border. This field is always a nonzero value.
height	Specifies the inside size of the created window, excluding the border. This field is always a nonzero value.
border_width	Specifies the width of the border of the created window, in pixels.
override_redirect	Specifies the value set in the *override_redirect* field of the window. If this field is set to the value of **True**, window manager clients should ignore the window.

Related Information

The **CreateNotify** event.

The **XCreateSimpleWindow** subroutine, **XCreateWindow** subroutine.

XDestroyWindowEvent Data Structure

```
typedef struct {
   int type;                /* DestroyNotify */
   unsigned long serial;    /* Number of the last request
                               processed by the server */
    Bool send_event;         /* True if this came from a SendEvent
                               request */
    Display *display;        /* The display the event was read
                               from */
    Window event;
    Window window;
} XDestroyWindowEvent;
```

The **XDestroyWindowEvent** data structure includes the following fields:

event	Specifies the window on which the event was generated. This field is set to either the destroyed window or its parent, depending on whether **StructureNotify** or **SubstructureNotify** was selected.
window	Specifies the window that is destroyed.

Related Information

The **DestroyNotify** event.

The **XDestroySubwindows** subroutine, **XDestroyWindow** subroutine.

XGravityEvent Data Structure

```
typedef struct {
   int type;                  /* GravityNotify */
   unsigned long serial;      /* Number of the last request
                                 processed by the server */
    Bool send_event;           /* True if this came from a
                                 SendEvent request */
    Display *display;          /* The display the event was read
                                 from */
    Window event;
    Window window;
   int x, y;
} XGravityEvent;
```

The **XGravityEvent** data structure includes the following fields:

event	Specifies the window on which the event was generated. This field can be set to either the window that was moved or its parent, depending on whether **StructureNotify** or **SubstructureNotify** was selected.
window	Specifies the window that was moved.
x	Specifies the x coordinate of the upper left outside corner of the window. This coordinate is relative to the origin of the new parent window.
y	Specifies the y coordinate of the upper left outside corner of the window. This coordinate is relative to the origin of the new parent window.

Related Information

The **GravityNotify** event.

The **XConfigureWindow** subroutine, **XMoveResizeWindow** subroutine, **XResizeWindow** subroutine.

XMapEvent Data Structure

```
typedef struct {
   int type;                   /* MapNotify*/
   unsigned long serial;       /* Number of the last request
                                  processed by the server */
    Bool send_event;            /* True if this came from a
                                  SendEvent request */
    Display *display;           /* The display the event was read
                                  from */
    Window event;
    Window window;
    Bool override_redirect;     /* Boolean, is override set... */
} XMapEvent;
```

The **XMapEvent** data structure includes the following fields:

event	Specifies the window on which the event was generated. This field is set to either the mapped window or its parent, depending on whether **StructureNotify** or **SubstructureNotify** was selected.
window	Specifies the window that was mapped.
override_redirect	Specifies the value set in the *override_redirect* field of the window. If this field is the value of **True**, window manager clients should ignore this window, because these events usually are generated from pop ups, which override structure control.

Related Information

The **MapNotify** event.

The **XMapRaised** subroutine, **XMapSubwindows** subroutine, **XMapWindow** subroutine.

XMappingEvent Data Structure

```
typedef struct {
   int type;                 /* MappingNotify */
   unsigned long serial;     /* Number of the last request
                                processed by the server */
    Bool send_event;          /* True if this came from a SendEvent
                                request */
    Display *display;         /* The display the event was read
                                from */
    Window window;            /* Unused */
    int request;              /* MappingModifier, MappingKeyboard,
                                      MappingPointer */
    int first_keycode;        /* The first_keycode */
    int count;                /* Defines the range of change with
                                first_keycode*/
} XMappingEvent;
```

The **XMappingEvent** data structure includes the following fields:

request	Specifies the kind of mapping change that occurred. The *request* field can have the following values:

MappingModifier	Indicates that the modifier mapping was changed.
MappingKeyboard	Indicates that the keyboard mapping was changed.
MappingPointer	Indicates that the pointer button mapping was changed.

first_keycode	Specifies the first number in the range of the altered mapping. This field is set only if the *request* field is **MappingKeyboard.**
count	Specifies the number of keycodes altered. This field is set only if the *request* field is **MappingKeyboard**.

To update the client application's knowledge of the keyboard, use the **XRefreshKeyboardMapping** subroutine.

Related Information

The **MappingNotify** event.

The **XChangeKeyboardMapping** subroutine, **XRefreshKeyboardMapping** subroutine, **XSetModifierMapping** subroutine, **XSetPointerMapping** subroutine.

XReparentEvent Data Structure

```
typedef struct {
    int type;                   /* ReparentNotify */
    unsigned long serial;       /* Number of the last request
                                   processed by the server */
    Bool send_event;            /* True if this came from a
                                   SendEvent request*/
    Display *display;           /* The display the event was read
                                   from */
    Window event;
    Window window;
    Window parent;
    int x, y;
    Bool override_redirect;
} XReparentEvent;
```

The **XReparentEvent** data structure includes the following fields:

event	Specifies the window on which the event was generated. This field is set to either the reparented window or the old or new parent, depending on whether **StructureNotify** or **SubstructureNotify** was selected.
window	Specifies the window that was reparented.
parent	Specifies the new parent window.

x	Specifies the x coordinate of the upper left outer corner of the reparented window. This coordinate is relative to the origin of the new parent window.
y	Specifies the y coordinate of the upper left outer corner of the reparented window. This coordinate is relative to the origin of the new parent window
override_redirect	Specifies the value set in the *override_redirect* field of the reparented window. If this field is the value of **True**, window manager clients should ignore the window.

Related Information

The **ReparentNotify** event.

The **XReparentWindow** subroutine.

XUnmapEvent Data Structure

```
typedef struct {
   int type;                  /* UnmapNotify */
   unsigned long serial;      /* Number of the last request
                                 processed by the server */
    Bool sendevent;            /* True if this came from a
                                 SendEvent request */
    Display *display;          /* The display the event was read
                                 from */
    Window event;
    Window window;
    Bool from_configure;
} XUnmapEvent;
```

The **XUnmapEvent** data structure includes the following fields:

event	Specifies the window on which this event was generated. This paramer may be set to either the unmapped window or its parent, depending on whether **StructureNotify** or **SubstructureNotify** was selected.
window	Specifies the window that was unmapped.
from_configure	Specifies the value of **True** if the event was generated as a result of resizing the parent window when the window itself had a *win_gravity* field of **UnmapGravity**.

Related Information

The **UnmapNotify** event.

The **XUnmapSubwindows** subroutine, **XUnmapWindow** subroutine.

XVisibilityEvent Data Structure

```
typedef struct {
   int type;                  /* VisibilityNotify */
   unsigned long serial;      /* Number of the last request
                                 processed by the server */
   Bool send_event;           /* True if this came from a
                                 SendEvent request */
   Display *display;          /* The display the event was read
                                 from */
   Window window;
   int state;
} XVisibilityEvent;
```

The **XVisibilityEvent** data structure includes the following fields:

window	Specifies the window whose visibility state changes.
state	Specifies the visibility state of the window. This field can have one of the following values:

VisibilityUnobscured

VisibilityPartiallyObscured

VisibilityFullyObscured.

Related Information

The **VisibilityNotify** event.

XCirculateRequestEvent Data Structure

```
typedef struct {
   int type;                /* CirculateRequest */
   unsigned long serial;    /* Number of the last request
                               processed by the server */
   Bool send_event;         /* True if this came from a
                               SendEvent request */
   Display *display;        /* The display the event was read
                               from */
   Window parent;
   Window window;
   int place;               /* PlaceOnTop, PlaceOnBottom */
} XCirculateRequestEvent;
```

The **XCirculateRequestEvent** data structure includes the following fields:

parent	Specifies the parent window.
window	Specifies the subwindow to be restacked.
place	Specifies the new position of the window in the stacking order. This field can have either of the following values:

PlaceOnTop	Indicates that the window will be placed on top of all siblings.
PlaceOnBottom	Indicates that the window will be placed below all siblings.

Related Information

The **CirculateRequest** event.

The **XCirculateSubwindows** subroutine, **XCirculateSubwindowsDown** subroutine, **XCirculateSubwindowsUp** subroutine.

XConfigureRequestEvent Data Structure

```
typedef struct {
   int type;                  /* ConfigureRequest */
   unsigned long serial;      /* Number of the last request
                                 processed by the server */
   Bool send_event;           /* True if this came from a
                                 SendEvent request */
   Display *display;          /* The display the event was read
                                 from */
   Window parent;
   Window window;
   int x, y;
   int width, height;
   int border_width;
   Window above;
   int detail;                /* Above, Below, TopIf, BottomIf,
                                       Opposite */
   unsigned long value_mask;
} XConfigureRequestEvent;
```

The **XConfigureRequestEvent** data structure includes the following fields:

parent	Specifies the parent window.
window	Specifies the window to be reconfigured.
x	Specifies the x coordinate of the upper left outer corner of the reconfigured window. The value for this field is set according to the current geometry of the window.
y	Specifies the y coordinate of the upper left outer corner of the reconfigured window. The value for this field is set according to the current geometry of the window.
width	Specifies the size of the reconfigured window, excluding the border. The value for this field is set according to the current geometry of the window.
height	Specifies the size of the reconfigured window, excluding the border. The value for this field is set according to the current geometry of the window.

border_width	Specifies the width of the border of the reconfigured window, in pixels. The value for this field is set according to the current geometry of the window.
above	Specifies the sibling window. This field can have one of the following values:

None	Indicates that the reconfigured window is placed on the bottom of the stack with respect to sibling windows. This is the default value for this field.
SiblingWindow IDs	The reconfigured window is placed on top of these sibling windows.

detail	Specifies the notify detail. This member can be set to **Below, TopIf, BottomIf**, or **Opposite**. The default value for this field is **Above.**
value_mask	Specifies which components were indicated in the configure window request.

Related Information

The **ConfigureRequest** event.

The **XConfigureWindow** subroutine, **XLowerWindow** subroutine, **XMapRaised** subroutine, **XMoveResizeWindow** subroutine, **XMoveWindow** subroutine, **XRaiseWindow** subroutine, **XResizeWindow** subroutine, **XRestackWindows** subroutine, **XSetWindowBorderWidth** subroutine.

XMapRequestEvent Data Structure

```
typedef struct {
   int type;                /* MapRequest */
   unsigned long serial;    /* Number of the last request
                               processed by the server */
    Bool send_event;         /* True if this came from a
                               SendEvent request */
    Display *display;        /* The display the event was read
                               from */
   Window parent;
    Window window;
} XMapRequestEvent;
```

The **XMapRequestEvent** data structure includes the following fields:

parent	Specifies the parent window.
window	Specifies the window to be mapped.

Related Information

The **MapRequest** event.

The **XMapRaised** subroutine, **XMapSubwindows** subroutine, **XMapWindow** subroutine.

XResizeRequestEvent Data Structure

```
typedef struct {
    int type;                    /* ResizeRequest */
    unsigned long serial;        /* Number of the last request
                                    processed by the server */
    Bool send_event;             /* True if this came from a
                                    SendEvent request */
    Display *display;            /* The display the event was read
                                    from */
    Window parent;
    int width, height;
} XResizeRequestEvent;
```

The **XResizeRequestEvent** data structure includes the following fields:

window	Specifies the window that another client attempted to resize.
width	Specifies the inside size of the window, excluding the border.
height	Specifies the inside size of the window, excluding the border.

Related Information

The **ResizeRequest** event.

The **XConfigureWindow** subroutine, **XMoveResizeWindow** subroutine, **XResizeWindow** subroutine.

XColormapEvent Data Structure

```
typedef struct {
    int type;                      /* ColormapNotify */
    unsigned long serial;          /* Number of the last request
                                      processed by the server */
    Bool send_event;               /* True if this came from a
                                      SendEvent request */
    Display *display;              /* The display the event was read
                                      from */
    Window window;
    Colormap colormap;             /* The colormap or None */
    Bool new;
   int state;                     /* ColormapInstalled,
                                      ColormapUninstalled*/
} XColormapEvent;
```

The **XColormapEvent** data structure includes the following fields:

window	Specifies the window whose associated colormap is changed, installed, or uninstalled.
colormap	Specifies the colormap associated with the window for a colormap changed by a call to the **XChangeWindowAttributes** subroutine. For a colormap

changed by a call to the **XFreeColormap** subroutine, this field is the value of **None**.

new	Specifies if the colormap for the specified window was changed or installed or uninstalled. This field can have either of the following values:

True	Indicates that the colormap was changed.
False	Indicates that the colormap was installed or uninstalled.

state	Specifies if the colormap is installed or uninstalled. This field can have either of the following values:

ColormapInstalled

ColormapUninstalled

Related Information

The **ColormapNotify** event.

The **XChangeWindowAttributes** subroutine, **XFreeColormap** subroutine, **XInstallColormap** subroutine, **XSetWindowColormap** subroutine, **XUninstallColormap** subroutine.

XClientMessageEvent Data Structure

```
typedef struct {
   int type;                    /* ClientMessage */
   unsigned long serial;        /* Number of the last request
                                   processed by the server */
   Bool send_event;             /* True if this came from a
                                   SendEvent request */
   Display *display;            /* The display the event was read
                                   from */
   Window window;
   Atom message_type;
   int format;
   union {
      char b[20];
      short s[10];
      long l[5];
      } data;
} XClientMessageEvent;
```

The **XClientMessageEvent** data structure includes the following fields:

window	Specifies the window to which the event was sent.
message_type	Specifies an atom which indicates how the data is to be interpreted by the receiving client. This field is not interpreted by the X Server.
format	Specifies whetner the data should be viewed as a list of bytes, shorts, or longs. This field should be set to 8 bits, 16 bits, or 32 bits.
data	Specifies a union which contains the members b (bytes), s (shorts), and l (longs). These members represent data of 20 8-bit values,10

16-bit values, and 5 32-bit values. Some message types may not use all these values. This field is not interpreted by the X Server.

Related Information

The **XAnyEvent** data structure.

The **ClientMessage** event.

The **XSendEvent** subroutine.

XPropertyEvent Data Structure

```
typedef struct {
   int type;                    /* PropertyNotify */
   unsigned long serial;        /* Number of the last request
                                   processed by the server */
   Bool send_event;             /* True if this came from a
                                   SendEvent request */
   Display *display;            /* The display the event was
                                   read from */
   Window window;
   Atom atom;
   Time time;
   int state;                   /* PropertyNewValue,
                                          PropertyDeleted*/
} XPropertyEvent;
```

The **XPropertyEvent** data structure includes the following fields:

window	Specifies the window whose associated property is changed.
atom	Specifies the atom of the property that is changed or requested.
time	Specifies the server time when the property is changed.
state	Specifies whether the property is changed to a new value or deleted. This field can have the following values:

PropertyNewValue	Indicates that the property is changed or that it is replaced with identical data using the **XChangeProperty** or **XRotateWindowProperties** subroutines.
PropertyDeleted	Indicates that the property is deleted using the **XDeleteProperty** or **XGetWindowProperty** subroutines.

Related Information

The **PropertyNotify** event.

The **XChangeProperty** subroutine, **XDeleteProperty** subroutine, **XGetWindowProperty** subroutine, **XRotateWindowProperties** subroutine.

XSelectionClearEvent Data Structure

```
typedef struct {
   int type;                  /* SelectionClear */
   unsigned long serial;      /* Number of the last request
                                 processed by the server */
    Bool send_event;          /* True if this came from a
                                 SendEvent request */
   Display *display;          /* The display the event was read
                                 from */
   Window window;
   Atom selection;
   Time time;
} XSelectionClearEvent;
```

The **XSelectionClearEvent** data structure includes the following fields:

window	Specifies the window losing ownership of the selection.
selection	Specifies the selection atom.
time	Specifies the last change time recorded for the selection.

Related Information

The **SelectionClear** event.

The **XSetSelectionOwner** subroutine.

XSelectionRequestEvent Data Structure

```
typedef struct {
   int type;                  /* SelectionRequest */
   unsigned long serial;      /* Number of the last request
                                 processed by the server */
    Bool send_event;          /* True if this came from a
                                 SendEvent request */
    Display *display;         /* The display the event was read
                                 from */
    Window owner;
    Window requestor;
    Atom selection;
    Atom target;
    Atom property;
    Time time;
} XSelectionRequestEvent;
```

The **XSelectionRequestEvent** data structure includes the following fields:

owner	Specifies the window owning the selection. This is the window specified by the current owner in the **XSetSelectionOwner** subroutine.
requestor	Specifies the window requesting the selection.

selection	Specifies the atom that names the selection.
target	Specifies the atom which indicates the type the selection is requested in.
property	Specifies a property name or the value of **None**.
time	Specifies the time, either in a timestamp (milliseconds) or in **CurrentTime,** taken from the **XConvertSelection** request.

Related Information

The **SelectionRequest** event.

The **XConvertSelection** subroutine, **XSetSelectionOwner** subroutine.

XSelectionEvent Data Structure

```
typedef struct {
    int type;                        /* SelectionNotify */
    unsigned long serial;            /* Number of the last request
                                       processed by the server */
    Bool send_event;                 /* True if this came from a
                                       SendEvent request */
    Display *display;                /* The display the event was
                                       read from */
    Window requestor;
    Atom selection;
    Atom target;
    Atom property;                   /* The atom or None */
    Time time;
} XSelectionEvent;
```

The **XSelectionEvent** data structure includes the following fields:

requestor	Specifies the window associated with the requestor of the selection.
selection	Specifies the atom that indicates the selection.
target	Specifies the atom that indicates the converted type.
property	Specifies the atom that indicates the property the result is stored on. This field is set to the value of **None** if the conversion fails.
time	Specifies the time when the conversion took place. This can be a timestamp (in milliseconds) or **CurrentTime**.

Related Information

The **SelectionNotify** event.

The **XConvertSelection** subroutine, **XSendEvent** subroutine.

XErrorEvent Data Structure

```
typedef struct {
   int type;
    Display *display;              /* The display the event was
                                      read from */
   unsigned long serial;          /* The serial number of the
                                      failed request */
   unsigned char error_code;      /* The error code of the failed
                                      request */
   unsigned char request_code;    /* The major op code of the
                                      failed request */
   unsigned char minor_code;      /* The minor op code of the
                                      failed request */
    XID resourceid;                /* The resource id */
} XErrorEvent;
```

The **XErrorEvent** data structure includes the following fields:

display	Specifies the display that the event was read from.
serial	Specifies the number of requests, starting with the one sent over the network connection when it was opened. This value is the value of **NextRequest** immediately before the failing call was made.
error_code	Specifies the error code of the failed request.
request_code	Specifies a protocol request of the procedure that failed. The *request_code* field values are defined in the **<X11/Xproto.h>** file.
minor_code	Specifies the minor op code of the failed request.
resourceid	Specifies the resource ID.

Related Information

The **<X11/Xproto.h>** header file.

The **XDisplayName** subroutine, **XGetErrorDatabaseText** subroutine, **XGetErrorText** subroutine, **XSetErrorHandler** subroutine, **XSetIOErrorHandler** subroutine.

XWMHints Data Structure

```
#define   InputHint          (1L<<0)
#define   StateHint          (1L<<1)
#define   IconPixmapHint     (1L<<2)
#define   IconWindowHint     (1L<<3)
#define   IconPositionHint   (1L<<4)
#define   IconMaskHint       (1L<<5)
#define   WindowGroupHint    (1L<<6)
#define   AllHints           (InputHint/StateHint/IconPixmapHint/
                              IconWindowHint/IconPositionHint/
                              IconMaskHint/WindowGroupHint)


typedef struct {
   long flags;                /* Marks which fields in this
                                 structure are defined */
   Bool input;                /* Indicates whether this application
                                 relies on the window manager to get
                                 keyboard input */
   int initial_state;         /* The initial state of the
                                 application */
   Pixmap icon_pixmap;        /* The pixmap to be used as the icon */
   Window icon_window;        /* The window to be used as the icon */
   int icon_x, icon_y;        /* The initial position of the icon */
   Pixmap icon_mask;          /* The pixmap to be used as the mask
                                 for the icon_pixmap field */
   XID window_group;          /* The id of the related window
                                 group */
} XWMHints ;
```

The **XWMHints** data structure includes the following fields:

flags Specifies which fields are defined in the **XWMHints** data structure. The values for this field are as follows:

InputHint

StateHint

IconPixmapHint

IconWindowHint

IconPositionHint

IconMaskHint

WindowGroupHint

AllHints

input Specifies the input focus model used by the application. This field communicates the input focus model to the window manager and can have the following values:

True	Indicates that the application accepts input, but never explicitly sets focus to any of the subwindows. These applications use the push model of focus management. The *input* field also has this value if the application sets input focus to its subwindows only when it is given to its top level window by a window manager.
False	Indicates that the application manages its input focus by explicitly setting focus to one of its subwindows whenever keyboard input is requested. These applications use the pull model of focus management. The *input* field also has this value if the application never expects any keyboard input.

Pull model window managers should make it possible for push model application to get input by setting input focus to the top level windows of applications with the *input* field set to the value of **True**. Push model window managers should ensure that pull model applications do not break them by resetting the input focus to **PointerRoot** when it is appropriate.

initial_state Specifies the initial state of the application. The values for this field are:

DontCareState	Don't know or care
NormalState	Most applications start this way
ZoomState	The application wants to start zoomed
IconicState	The application wants to start as an icon
InactiveState	The application believes it is seldom used; some window managers may put it on inactive menu

icon_mask Specifies which pixels of the *icon_pixmap* field should be used as the icon. The *icon_mask* field allows for nonrectangular pixmaps. Both fields must be bit maps.

icon_window Specifies the window to be used as an icon for window managers that support such use.

window_group Specifies if this window belongs to a group of other windows. For example, if a single application manipulates multiple top level windows, this field provides the window manager with enough information to iconify all of the windows instead of only one window.

Related Information

The **XGetWMHints** subroutine, **XSetWMHints** subroutine.

XSizeHints Data Structure

```
#define   USPosition    (1L<<0)    /* user specified x, y */
#define   USSize        (1L<<1)    /* user specified width,
                                      height */
#define   PPosition     (1L<<2)    /* program specified position */
#define   PSize         (1L<<3)    /* program specified size */
#define   PMinSize      (1L<<4)    /* program specified minimum
                                      size */
#define   PMaxSize      (1L<<5)    /* program specified maximum
                                      size */
#define   PResizeInc    (1L<<6)    /* program specified resize
                                      increments */
#define   PAspect       (1L<<7)    /* program specified min and
                                      max aspect ratios */
#define   PAllHints                (PPosition/PSize/PMinSize/
                                   PMaxSize/PResizeInc/PAspect)


typedef struct {
   long flags;                     /* Marks which fields in this
                                      structure are defined */
   int x, y;
   int width, height;
   int min_width, min_height;
   int max_width, max_height;
   int width_inc, height_inc;
   struct {
      int x;                       /* The numerator */
      int y;                       /* The denominator */
   } min_aspect, max_aspect;
   int base_width, base_height;
   int win_gravity;
} XSizeHints
```

The **XSizeHints** data structure includes the following fields:

flags Specifies how the position and size of the window is set. The values for this field are as follows:

USPosition	User specified x, y
USSize	User specified width, height
PPosition	Program specified position
PSize	Program specified size
PMinSize	Program specified minimum size
PMaxSize	Program specified maximum size
PResizeInc	Program specified resize increments
PAspect	Program specified minimum and maximum aspect ratios

	PAllHints	(PPosition|PSize|PMinSize|PMaxSize|PResizeInc|PAspect)
x	Specifies the x coordinate for the upper left corner of the window.	
y	Specifies the y coordinate for the upper left corner of the window.	
width	Specifies the width of the window.	
height	Specifies the height of the window.	
min_width	Specifies the minimum width of the window for the application.	
min_height	Specifies the minimum height of the window for the application.	
max_width	Specifies the maximum width of the window.	
max_height	Specifies the maximum height of the window.	
width_inc	Specifies an arithmetic progression of sizes, from minimum size to maximum size, for the window resize requests.	
height_inc	Specifies an arithmetic progression of sizes, from minimum size to maximum size, for the window resize requests.	
min_aspect	Specifies the minimum of the range of aspect ratios the application prefers. This field is expressed as a ratio of the *x* and *y* fields.	
max_aspect	Specifies the maximum of the range of aspect ratios the application prefers. This field is expressed as a ratio of the *x* and *y* fields.	
base_width	Defines an arithmetic progression, when used with the *width_inc* field, of the preferred window width.	
base_height	Defines an arithmetic progression, when used with the *height_inc* field, of the preferred window height.	

Related Information

The **XGetNormalHints** subroutine, **XGetSizeHints** subroutine, **XGetZoomHints** subroutine, **XSetNormalHints** subroutine, **XSetSizeHints** subroutine, **XSetZoomHints** subroutine.

XIconSize Data Structure

```
typedef struct {
   int min_width, min_height;
   int max_width, max_height;
   int width_inc, height_inc;
} XIconSize;
```

The **XIconSize** data structure includes the following fields:

min_width	Specifies the minimum icon width.
min_height	Specifies the minimum icon height.
max_width	Specifies the maximum icon width.

max_height	Specifies the maximum icon height.
width_inc	Specifies an arithmetic progression of sizes, from minimum to maximum, that represent the supported icon sizes.
height_inc	Specifies an arithmetic progression of sizes, from minimum to maximum, that represent the supported icon sizes.

Related Information

The **XGetIconSizes** subroutine, **XSetIconSizes** subroutine.

XClassHint Data Structure

```
typedef struct {
   char *res_name;
    char *res_class;
} XClassHint;
```

The **XClassHint** data structure includes the following fields:

res_name	Specifies the application name.
res_class	Specifies the application class.

Related Information

The **XGetClassHint** subroutine, **XSetClassHint** subroutine.

XrmValue Data Structure

```
typedef struct {
   unsigned int size;
   caddr_t addr;
} XrmValue, *XrmValuePtr;
```

A resource database is an opaque type used by the lookup routines.

```
    typedef struct _XrmHashBucketRec *XrmDatabase;
```

Database values consist of a size, an address, and a representation type. The representation type allows storage of data tagged by some application defined type (for example, font or color). It has nothing to do with the C language data type or with its class.

The **XrmValue** data structure has the following fields:

size	Specifies the size of the resource database, specified in bytes.
addr	Specifies the location of the resource database.

XrmOptionDescList Data Structure

```
typedef enum {
   XrmoptionNoArg,            /* Value is specified in
                                 OptionDescRec.value */
   XrmoptionIsArg,            /* Value is the option string
                                 itself */
   XrmoptionStickyArg,        /* Value is characters immediately
                                 following option */
   XrmoptionSepArg,           /* Value is next argument in argv */
   XrmoptionResArg,           /* Resource and value in next
                                 argument in argv */
   XrmoptionSkipArg,          /* Ignore this option and the next
                                 argument in argv */
   XrmoptionSkipLine,         /* Ignore this option and the rest
                                 of argv */
} XrmOptionKind;


typedef struct {
   char *option;              /* Option specification string in
                                 argv */
   char *resourceName;        /* Binding and resource name
                                 (sans application name)
   XrmOptionKind argKind;     /* Which style of option it is */
   caddr_t value;             /* Value to provide if
                                 XrmoptionNoArg */
} XrmOptionDescRec, *XrmOptionDescList;
```

The **XrmOptionDescList** data structure includes the following fields:

option	Specifies the option specification string in the *argv* field.
resourceName	Specifies the binding and resource name (without the application name).
argKind	Specifies the style of option. This field can be one of the following values:

XrmoptionNoArg	The value is specified in the *value* field.
XrmoptionIsArg	The value is the option string itself.
XrmoptionStickyArg	The value is found in the characters immediately following the option.
XrmoptionSepArg	The value is the next argument in the *argv* field.
XrmoptionResArg	The resource and value in the next argument in the *argv* field.
XrmoptionSkipArg	Ignore this option and the next argument in the *argv* field.

XrmoptionSkipLine	Ignore this option and the rest of the *argv* field.

value	The value to provide if the *argKind* field is **XrmoptionNoArg.**

Related Information

The **XrmParseCommand** subroutine.

XAIXDeviceMappingEvent Data Structure

```
typedef struct {
   int type;                  /* Event type */
   unsigned long serial       /* Number of last request processed by
                               server */
   Bool send_event;           /* True if from SendEvent request */
   Display *display;          /* Display event was read from */
   Window window;             /* unused */
   int request;               /* AIXMappingDial or AIXMappingLpfk */
   int lpfkmask;              /* lpfk input */
   int lightmask;             /* lpfk output */
   int dialmask;              /* dial mask */
} XAIXDeviceMappingEvent;
```

type	Specifies the event type, which is **AIXDeviceMappingNotify**.
serial	Specifies the serial number of last event processed in the server.
send_event	Specifies if the event was generated by a **SendEvent** protocol request. If it was, the *send_event* field is set to the value of **True.**
display	Specifies the connection to the X Server.
window	Unused in this request.
lpfkmask	Set to new lpfkmask value if the request is **AIXMappingLpfk.**
lightmask	Set to the new lightmask value if the request is **AIXMappingLpfk.**
dialmask	Set to the new dialmask value if the request is **AIXMappingDial.**

Enhanced X-Windows Extensions Overview

This chapter describes techniques for writing extensions to the Xlib library that have the same performance rate as Core protocol requests. The Using AIX Extensions section describes Enhanced X-Windows extensions.

Note: Because an Enhanced X-Windows 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.

Enhanced X-Windows Basic Extension Subroutines

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

Hooking Into the Enhanced X-Windows 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 Enhanced X-Windows protocol and the Enhanced X-Windows library interface. Hooking routines, which generate protocol requests for Enhanced X-Windows, 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** subroutine.

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 subroutine. This structure is public to extension and cannot be changed.

The **XInitExtension** subroutine calls the **XQueryExtension** subroutine 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 the stub implementor needs to access the extension.

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

The types of functions and associated subroutines that hook into the Enhanced X-Windows library are the following:

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

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

Graphics Context (GC) Caching With Enhanced X-Windows

GC's 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 subroutine whose behavior depends on the contents of a **GC** must flush 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 flushed.

Using Enhanced X-Windows for Graphics Batching

If you extend Enhanced X-Windows 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, here 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.

Using Enhanced X-Windows to Define Extension Stubs, Requests and Replies

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

- Request names
- Request structures
- Reply structures

X Server requests contain the length, expressed in 16-bit quantity of 32-bits, of the request. Therefore, a single request can be no more than 256 kilobytes in length. 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
```

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

Using Enhanced X-Windows to 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 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 subroutine 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, therefore 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.

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>**. 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.

Using Enhanced X-Windows to 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 extensions that have reply structures longer than 32 bytes are the following:

- The **GetWindowAttributes** protocol request
- The **QueryFont** protocol request
- The **QueryKeymap** protocol request
- The **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).

Using Enhanced X-Windows to Lock Data Structures

To support asynchronous input and multithreaded access to a single display connection, the display must be locked so that each stub can 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.

Using Enhanced X-Windows to Send Protocol Request and Arguments

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

- The **GetReq** macro
- The **GetReqExtra** macro
- The **GetResReq** macro
- The **GetEmptyReq** macro

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);
```

If the protocol request has a single 32-bit argument (such as a **Pixmap**, **Window**, **Drawable**, **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);
```

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);

/* fill in arguments here */

req—>arg1 = arg1;
req—>arg2 = arg2;
```

A few stub routines, such as the **XCreateGC** subroutine and the **XCreatePixmap** subroutine, 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** subroutine and the **XSetBackground** subroutine, are special cases of more general routines like the **XMoveResizeWindow** subroutine and the **XChangeGC** subroutine. 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.

Using Variable Length Enhanced X-Windows 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 longwords. 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** macro. 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
- the number of bytes to be sent

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

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

If the data is 16-bit entities, use the **PackData** macro. It performs correctly on machines where a short is 32-bits instead of the usual 16.

Both the **Data** and the **PackData** macros can use their last argument more than once, so that the *argument* field should be a variable rather than an expression, such as `nitems*sizeof(item)`. This sort of computation should be done in a separate statement before calling the **Data** macro.

If the protocol request requires a reply, use the **_XSend** subroutine. The **_XSend** subroutine is faster than the **Data** macro, because it sends the data immediately instead of copying it into the output buffer.

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

Using Enhanced X-Windows For 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** subroutine, the request is sent immediately. The library, however, waits until any error generated has been handled.

Using Enhanced X-Windows to Allocate and Deallocate 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** extension subroutine to return a scratch buffer. This storage must only be used inside the critical section of your stub.

Using Enhanced X-Windows to Derive the Correct Extension Opcode

When writing an extension stub routine map from the call to the proper major and minor opcodes. While there are a number of strategies, the simplest and fastest is outlined here:

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** subroutine. 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 will return a pointer to the **XExtCodes** data structure for this extension, which you would 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.

Using Enhanced X-Windows Extension Event Types

An extension event is data generated asynchronously by the X Server as a result of some input device activity or as side effects of an extension request. Device-related events propagate from the source window to ancestor windows until some client application has selected that event type or until the event is explicitly discarded. The X Server never sends an event to a client application unless the client has specifically asked to be informed of that event type, usually by calling the **XSelectDeviceInput** extension subroutine from the **Xlib** library. However, the **AIXDeviceMappingNotify** events are always sent.

The event type describes a specific event generated by the X Server. For each event type, a corresponding name is defined in the **<X11/AIX.h>** header file. The following table lists the event category and its associated event type or types.

Event Category	Event Type
Lpfk events	**LPFKeyPress**
Dial events	**DialRotate**
Focus and mapping change events	**AIXFocusIn** **AIXFocusOut** **AIXDeviceMappingNotify**

Using Enhanced X-Windows to Define Event Structures

Each event type has a corresponding structure declared in the **<X11/AIX.h>** header file. Event structures have the following fields:

type	Specifies the event type constant name that uniquely identifies the type. For example, when the X Server reports a **DialRotate** event to a client application, it sends an **XDialRotatedEvent** structure with the *type* field set to **DialRotate.**
displayID	Specifies a pointer to the display the event was read on.
send_event	Set to the value of **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.

The X Server can send extension events at any time in the input stream, even while the client application sends a request and receives a reply. Events received, while waiting for a reply, can be stored by the **Xlib** library in the event queue.

Using Enhanced X-Windows Event Masks

Clients select extension event reporting of most events relative to a window by passing an extension event mask to an **Xlib** library event-handling function that takes the *ext_event_mask* argument. The bits of the event mask are defined in the **<X11/AIX.h>** header file. Each bit in the event mask maps to an event mask name. The event mask name describes the event or events to be returned to a client application by the server.

The following table lists the event mask that can be specified in the *ext_event_mask* argument and the circumstances under which to specify them.

Event Mask	Circumstances
NoEventMask	No events wanted
LPFKeyPressMask	Lpfk key-down events wanted
DialRotateMask	Dial rotate events wanted
DialRotateMask	Dail rotate events wanted
AIXFocusChangeMask	Dial or lpfk input focus events wanted
AIXDeviceMapChangeMask	Device state change events wanted.

Using Enhanced X-Windows Dial and Lpfk Extensions

The lpfk and dial (or valuator) devices operate in two modes, **AutoLoad** and **EventReport**. These modes are mutually exclusive. The X Server automatically installs the attributes of dial and lpfk when in the **AutoLoad** mode. Under the **EventReport** mode, the client is responsible for downloading the attributes of dial and lpfk into the X Server. The server starts up with the **EventReport** mode.

Xlib provides routines to change the dial control or get the current dial control parameter. The dial control parameter is dial granularity. Routines for changing the on/off lpfk keypress input and keylight output are also provided.

The X Server can report **LPFKeyPress** events to a client when a Lighted Programmable Function Key (LPFKey) is pressed. To receive **LPFKeyPress** events in a client application, pass a window ID and the **LPFKeyPressMask** value as the *EventMask* parameter to **XSelectLpfkInput**.

An **LPFKeyPress** event has an event type of **LPFKeyPress** and an associated structure name of **XLPFKeyPressedEvent**.

The X Sever can report **DialRotate** events to a client when a dial is rotated. To receive **DialRotate** events in a client application, pass a window ID and **DialRotateMask** as the *EventMask* parameter to **XSelectDialInput**.

DialRotate events are generated like **KeyPress** events. They have the **DialRotate** event type and an associated **XDialRotatedEvent** data structure.

Processing Enhanced X-Windows Input Extension Events

The event types reported to a client application during event processing depend on the event masks in the *EventMask* parameter of the **XSelectDeviceInput** extension subroutine. Processing descriptions include explanations of the structure or structures associated with the event. All the event structures contain *type* and *display* fields.

The following table lists the event mask, the associated event type or types, and the structure name associated with the event type.

Event Mask	Event Type	Structure
LPFKeyPressMask	LPFKeyPress	XLPFKeyPressedEvent
DialRotateMask	DialRotate	XDialRotatedEvent
AIXFocusChangeMask	AIXFocusIn AIXFocusOut	AIXFocusInEvent AIXFocusOutEvent
AIXDeviceMapChangeMask	AIXDeviceMappingNotify	XAIXDeviceMappingFormat

Processing Enhanced X-Windows Dial and Lpfk Events

The X Server reports **LPFKeyPress** events to clients that need to know when an **LPFKey** is pressed. It also reports **DialRotate** events to clients that need to know when a dial is rotated.

To receive **LPFKeyPress** events in a client application, pass a window ID and **LPFKeyPressMask** as the *event_mask* field to the **XSelectLpfkInput** extension subroutine. To select input for a specific key, use the **XSelectLPFK** subroutine.

To receive **DialRotate** events in a client application, pass a window ID and **DialRotateMask** as the *event_mask* field to the **XSelectDialInput** extension subroutine. To select the specific dial, use the **XSelectDial** subroutine.

The source of the event is the smallest window containing the pointer. The window used by the X Server to report these events depends on its position in the window hierarchy and whether any intervening window prohibits the generation of these events.

The X Server searches the window hierarchy, starting with the source window until it locates the first window specified by a client window specified by a client as having an interest in these events. If one of the intervening windows has its *do_not_propogate_mask* field set to prohibit generation of the event type, the event of those types are suppressed. Clients can modify the window used for reporting with the **XSetDeviceInputFocus** extension subroutine.

The structures associated with these events are the **XLFPKeyPressedEvent** data structure and the **XDialRotatedEvent** data structure. These structures have the following fields:

window	Specifies the ID of the window on which the event was generated. It is referred to as the *Event Window*. The X Server uses this window to report the event.
root	Specifies the window ID of the source window or the root window.
x_root	Specifies the x coordinate, which is relative to the origin of the root window at the time of the event, and which is set to the pointer.
y_root	Specifies the y coordinate, which is relative to the origin of the root window at the time of the event, and which is set to the pointer.
same_screen	Indicates whether the event window is on the same screen as the root window. This can be:
True	Indicates that the event window and the root window are on the same screen.
False	Indicates that the event window and the root window are not on the same screen.
subwindow	Specifies the child of the event window that is an ancestor of or is the source field if the source window is an inferior of the event window. Otherwise, the X Server sets *subwindow* to the value of **None**.
time	Specifies the time in milliseconds when the event was generated since the server reset.
x	Specifies the x coordinate, which is relative to the origin of the event window if the event window is on the same screen as the root window. Otherwise, this coordinate is the value of **0**.
y	Specifies the x coordinate, which is relative to the origin of the event window if the event window is on the same screen as the root window. Otherwise, this coordinate is the value of **0**.
state	Indicates the state of the pointer buttons and modifier keys just prior to the event. The pointer buttons can be the bitwise inclusive OR of one or more of the following button or modifier key masks:

Button1Mask	**Mod1Mask**	**ShiftMask**
Button2Mask	**Mod2Mask**	**LockMask**
Button3Mask	**Mod3Mask**	**ControlMask**
Button4Mask	**Mod4Mask**	
Button5Mask	**Mod5Mask**	

The **XLPFKeyPressedEvent** data structure contains the following unique component:

keycode	Specifies a number that represents a physical key on the lpfk that ranges from 0 to 31.

The **XDialRotatedEvent** data structure contains the following unique components:

dialnum	Specifies the dial that has been rotated.

dialval	Specifies the difference between the current dial value and the last dial value. For clockwise rotation, this value is positive. For counterclockwise rotation, this value is negative.

Processing Enhanced X-Windows Dial and Lpfk Input Focus Events

This section describes the processing that occurs for the input focus events **AIXFocusIn** and **AIXFocusOut**. The X Server reports **AIXFocusIn** or **AIXFocusOut** events to clients requiring information about when the dial or lpfk input focus changes. The dial or lpfk is always attached to a window, which is usually the root window or a top-level window called the focus window. The focus window and the position of the pointer determines the window that receives dial of lpfk input.

To receive **AIXFocusIn** and **AIXFocusOut** events in a client application, pass a window ID and **AIXFocusChangeMask** as the *ext_event_mask* field to **XSelectDeviceInput.**

The fields of the **AIXFocusInEvent** data structure and the **XAIXFocusOutEvent** data structure associated with these events include the following:

window	Specifies the ID of the window on which the **AIXFocusIn** or **AIXFocusOut** event was generated. The X Server uses this window to report the event
mode	Specifies the mode, which is **NotifyNormal**
devtype	Specifies the focus device
detail	Indicates the notify detail. It can be one of the following:

NotifyAncestor	**NotifyVirtual**
NotifyInferior	**NotifyNonlinear**
NotifyNonlinearVirtual	**NotifyPointer**
NotifyPointerRoot	**NotifyDetailNone**

Processing Enhanced X-Windows AIXFocus Events

Focus events are identified by the **XAIXFocusInEvent** data structure or the **XAIXFocusOutEvent** data structure whose *mode* field is **NotifyNormal**. The X Server processes normal focus events according to the following scenarios:

1. When the focus moves from window A to window B and A is an inferior of B with the pointer in window P, the X Server generates the following:
   - An **AIXFocusOut** event on window A that has the **XAIXFocusOutEvent** data structure with **NotifyAncestor** as the *detail* field.
   - An **AIXFocusOut** event on each window between window A and window B exclusive that has the **XAIXFocusOutEvent** data structure with **NotifyVirtual** as the *detail* field.
   - An **AIXFocusIn** event on window B that has the **XAIXFocusOutEvent** data structure with **NotifyInferior** as the *detail* field.
   - An **AIXFocusIn** event on each window below window B down to and including window P if window P is an inferior of window B, but is not window A, and is not an inferior of window A that has the **XAIXFocusInEvent** data structure with **NotifyInferior** as the *detail* field.
2. When the focus moves from window A to window B and B is an inferior of A with the pointer in window P, the X Server generates:
   - An **AIXFocusOut** event on each window from window P up to, but not including window A (in that order), if window P is an inferior of window A, but is not window A,

and window P is not an inferior of window B nor an ancestor of window B that has the **XAIXFocusOutEvent** data structure with **NotifyPointer** as the *detail* field.

- An **AIXFocusOut** event on window A that has the **XAIXFocusOutEvent** data structure with **NotifyInferior** as the *detail* field.
- An **AIXFocusIn** event on each window between window A and window B exclusive that has the **XAIXFocusInEvent** data structure with **NotifyVirtual** as the *detail* field.
- An **AIXFocusIn** event on window B that has the **XAIXFocusInEvent** data structure with **NotifyAncestor** as the *detail* field.

3. When the focus moves from window A to window B and window C is their least common ancestor, and with the pointer in window P, the X Server generates the following:

   - An **AIXFocusOut** event on each window from window P up to, but not including window A if window P is an inferior of window A that has the **XAIXFocusOutEvent** data structure with **NotifyPointer** as the *detail* field.
   - An **AIXFocusOut** event on window A that has the **XAIXFocusOutEvent** data structure with **NotifyNonlinear** as the *detail* field.
   - An **AIXFocusOut** event on each window between window A and window C exclusive that has the **XAIXFocusOutEvent** data structure with **NotifyNonlinearVirtual** as the *detail* field.
   - An **AIXFocusIn** event on each window between C and B exclusive that has the **XAIXFocusInEvent** data structure with **NotifyNonlinearVirtual** as the *detail* field.
   - An **AIXFocusIn** event on window B that has the **XAIXFocusInEvent** structure with **NotifyNonlinear** as the *detail* field.
   - An **AIXFocusIn** event on each window below window B down to and including window P and window P is an inferior of window B that has the **XAIXFocusInEvent** data structure with **NotifyPointer** as the *detail* field.

4. If the focus window is **PointerRoot** (events sent to the window under the pointer) or **None** in **XSetDeviceInputFocus**, when the focus moves from window A to **PointerRoot** or **None** with the pointer in window P, the X Server generates the following:

   - An **AIXFocusOut** event on each window from window P up to, but not including window A if window P is an inferior of window A that has the **XAIXFocusOutEvent** data structure with **NotifyPointer** as the *detail* field.
   - An **AIXFocusOut** event on window A that has the **XAIXFocusOutEvent** data structure with **NotifyInferior** as the *detail* field.
   - An **AIXFocusOut** event on each window above window A up to, and including its root window, and window A and is not a root window that has the **XAIXFocusOutEvent** data structure with **NotifyNonlinearVirtual** as the *detail* field.
   - An **AIXFocusIn** event on the root window of all screens that has the **XAIXFocusInEvent** data structure with **NotifyPointerRoot** or **NotifyDetailNone** as the *detail* field.
   - An **AIXFocusIn** event on each window from the root of window P down to, and including window P, (new focus **PointerRoot**) that has the **XAIXFocusInEvent** data structure with **NotifyPointerRoot** as the *detail* field.

5. When the focus moves from **PointerRoot** (events sent to the window under the pointer) or **None** to window A, with the pointer in window P, the X Server generates the following:
   - An **AIXFocusOut** event on each window from window P up to and including the root of window P, (the old focus **PointerRoot**) that has the **XAIXFocusOutEvent** data structure with **NotifyPointerRoot** as the *detail* field.
   - An **AIXFocusOut** event on all root windows that have the **XAIXFocusEvent** data structure with **NotifyPointerRoot** or **NotifyDetailNone** as the *detail* field.
   - An **AIXFocusIn** event on each window from the root of window A down to, but not including window A, and window A is not a root window, that has the **XAIXFocusInEvent** data structure with **NotifyNonlinearVirtual** as the *detail* field.
   - An **AIXFocusIn** event on window A that has the **XAIXFocusInEvent** data structure with **NotifyNonlinear** as the *detail* field.
   - An **AIXFocusIn** event on each window below window A down to, and including window P, and window P is an inferior of window A that has the **XAIXFocusInEvent** data structure with **NotifyPointer** as the *detail* field.
6. When the focus moves from **PointerRoot** (events sent to the window under the pointer) to **None** (or vice versa), with the pointer in window P, the X Server generates the following:
   - An **AIXFocusOut** event on each window from window P up to and including the root of window P, (the old focus **PointerRoot**) that has the **XAIXFocusOutEvent** data structure with **NotifyPointerRoot** as the *detail* field.
   - An **AIXFocusOut** event on all root windows that have the **XAIXFocusOutEvent** data structure with **NotifyPointerRoot** or **NotifyDetailNone** as the *detail* field.
   - An **AIXFocusIn** event on all root windows that have the **XAIXFocusInEvent** data structure with **NotifyDetailNone** or **NotifyPointerRoot** as the *detail* field.
   - An **AIXFocusIn** event on each window below window P down to, and including window P (new focus is **PointerRoot**) that has the **XAIXFocusInEvent** data structure with **NotifyPointerRoot** as the *detail* field.

Processing Enhanced X-Windows AIXDeviceMappingNotify Events

The X Server reports **AIXDeviceMappingNotify** events to all clients. This event type is generated when a client application uses the following:

- The **XSetDialControl** extension subroutine, which indicates that new dial granularity has been set.
- The **XSetLpfkControl** extension subroutine, which indicates that new lpfk input/output settings.

The fields of the **XAIXMappingEvent** data structure associated with the **XAIXDeviceMappingNotify** event include *request*, *dialmask*, *lpfkmask*, and *lightmask*.

- The *request* field indicates the kind of mapping change that occurred. It can be **AIXMappingDial** or **AIXMappingLpfk**.
- If the *request* field is **AIXMappingDial**, the dial granularity is changed. If the *request* field is **AIXMappingLpfk**, the lpfk input/output setting is changed.

The *dialmask*, *lpfkmask*, and *lightmask* fields indicate the setting value.

Related Information

The **<X11/Xproto.h>** header file.

The **_xDoSomethingReply** data structure, **xResourceReq** data structure, **xDoSomethingReq** data structure , **XExtCodes** data structure.

The **_XAllocScratch** extension subroutine, **_XReply** extension subroutine, **XESetCloseDisplay** extension subroutine, **XESetCopyGC** extension subroutine, **XESetCreateFont** extension subroutine, **XESetCreateGC** extension subroutine, **XESetError** extension subroutine, **XESetErrorString** extension subroutine, **XESetEventToWire** extension subroutine, **XESetFlushGC** extension subroutine, **XESetFreeFont** extension subroutine, **XESetFreeGC** extension subroutine, **XESetWireToEvent** extension subroutine, **XFreeExtensionList** extension subroutine, **XGetErrorText** subroutine, **XInitExtension** subroutine, **XListExtensions** extension subroutine, **XMaxRequestSize** extension subroutine, **XQueryExtension** extension subroutine, **XSelectDeviceInput** extension subroutine, **XSelectDialInput** extension subroutine, **XSelectLpfkInput** extension subroutine.

The **XCreateGC** subroutine, **XCreatePixmap** subroutine, **XMapWindow** subroutine, **XMoveWindow** subroutine, **XSynchronize** subroutine.

The **GetKeyboardControl** protocol request, **GetWindowAttributes** protocol request, **GrabServer** protocol request, **QueryFont** protocol request, **QueryKeymap** protocol request.

List of Enhanced X-Windows Extension Data Structures

The **_XExtCodes** data structure

The **xDoSomethingReq** data structure

The **xResourceReq** data structure

The **XLPFKeyPressedEvent** data structure

The **XDialRotatedEvent** data structure

The **XAIXFocusChangeEvent** data structure

_XExtCodes Data Structure

```
typedef struct _XExtCodes {
   int extension;       /* extension number                     */
   int major_opcode;    /* major opcode assigned by server      */
   int first_event;     /* first event number for the extension */
   int first_error;     /* first error number for the extension */

} XExtCodes;
```

XLPFKeyPressedEvent Data Structure

```
typedef struct {
   int type;                  /* of event */
   unsigned long serial;      /* number of last request processed by
                               the server */
   Bool send_event;           /* true if this came from a SendEvent
                               request */
   Display *display;          /* display the ivent was read from */
   Window window;             /* "event" window it is reported
                                 relative to */
   Window root;               /* root window that the event occurred
                               on */
   Window subwindow;          /* child window */
   Time time;                 /* milliseconds */
   int x, y;                  /* pointer x, y coordinates in the
                                 event window */
   int x_root, y_root;        /* coordinates relative to root */
   unsigned int state;        /* key or button mask */
   unsigned int keycode;      /* detail */
   Bool same_screen;          /* same screen flag */

} XLPFKeyEvent;

typedef XLPFKeyEvent XLPFKeyPressedEvent;
```

XDialRotatedEvent Data Structure

```
typedef struct {
   int type;                  /* of event */
   unsigned long serial;      /* number of last request processed by
                               the server */
   Bool send_event;           /* true if this came from a SendEvent
                               request */
   Display *display;          /* display the ivent was read from */
   Window window;             /* "event" window it is reported
                                 relative to */
   Window root;               /* root window that the event occurred
                               on */
   Window subwindow;          /* child window */
   Time time;                 /* milliseconds */
   int x, y;                  /* pointer x, y coordinates in the
                                 event window */
   int x_root, y_root;        /* coordinates relative to root */
   unsigned int state;        /* key or button mask */
   short int dialval;         /* dial value */
   short int dialnum;         /* dial number */
   Bool same_screen;          /* same screen flag */

} XRotateEvent;

typedef XRotateEvent XDialRotatedEvent;
```

XAIXFocusChangeEvent Data Structure

```
typedef struct {
   int type;                 /* AIXFocusIn or AIXFocusOut */
   unsigned long serial;     /* number of last request processed
                                by the server */
   Bool send_event;          /* true if this came from a SendEvent
                                request */
   Display *display;         /* display the ivent was read from */
   Window window;            /* "event" window it is reported
                                relative to */
   short mode;               /* NotifyNormal */
   short devtype;            /* dial or lpfk */
   int detail;               /* NotifyAncestor, NotifyVirtual,
                                NotifyInferior, NotifyNonLinear,
                                NotifyNonLinearVirtual,
                                NotifyPointer, NotifyPointerRoot,
                                NotifyDetailNone, */

} XAIXFocusChangeEvent;

typedef XAIXFocusChangeEvent XAIXFocusInEvent;
typedef XAIXFocusChangeEvent XAIXFocusOutEvent;
```

Enhanced X-Windows Toolkit Overview

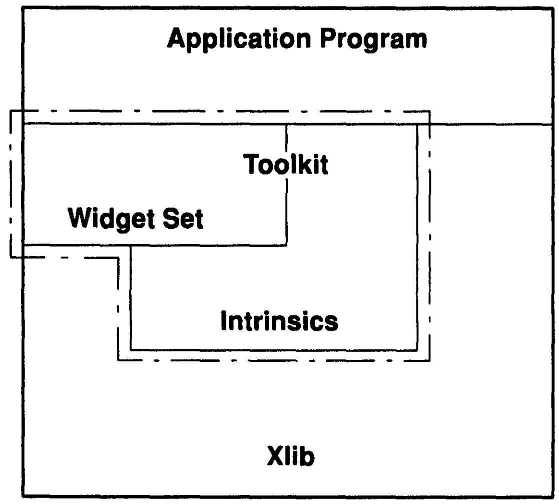
The Enhanced X-Windows Toolkit provides basic subroutines for building a wide variety of application environments and gives programmers a common set of underlying user-interface subroutines. These tools simplify the design of application user interfaces in the Enhanced X-Windows programming environment.

Enhanced X-Windows Intrinsics and Widgets Overview

The Intrinsics library and a widget set make up the Enhanced X-Windows Toolkit. The Intrinsics provide the base mechanisms necessary to build a wide variety of widget sets and application environments. Because the Intrinsics mask implementation details from the widget and the application programmer, the widgets and the application environments built with these widgets are fully extensible and support user-developed or extended components. By following a small set of conventions, widget programmers can extend their widget sets in new ways and can have these extensions function with the existing facilities.

The Intrinsics library is a library package layered on the **Xlib** library. As such, the Intrinsics library provide subroutines and structures for extending the basic programming abstractions of the Enhanced X-Windows. By providing these subroutines and structures for intercomponent and intracomponent interactions, the Intrinsics library provide the next layer of functionality from which the widget sets are built.

This X programming environment can be illustrated as three tiers, all sitting beneath the application program. The basic support (bottom tier) is the **C** language Enhanced X-Windows subroutines (**Xlib**). The Intrinsics is on the **Xlib** library. The widget set is on the Intrinsics. The combination of the Intrinsics and the widget set compose the Enhanced X-WindowsToolkit.



The Enhanced X-Windows toolkit application is usually a client of a given widget set, a subset of the Intrinsics subroutines, and a smaller set of **Xlib** subroutines. At the same time, a widget set is a client of both the Intrinsics library and the **Xlib** library. And, the Intrinsics library is a client of the **Xlib** library only.

For the application programmer, the Enhanced X-Windows Toolkit provides the following:

- A consistent interface (widget set) for writing applications.
- A set of Intrinsics mechanisms (subroutines and structures) used also for writing applications.

For the widget programmer, the Enhanced X-Windows Toolkit provides the following:

- A set of Intrinsics mechanisms for building widgets.
- An architectural model for constructing and composing widgets.

Applications that use the Enhanced X-Windows Toolkit must include the following header files:

- **<X11/Xlib.h>**
- **<X11/Intrinsic.h>**
- **<X11/StringDefs.h>**

and possibly **<X11/Shell.h>**

Widget implementations should include the **<X11/IntrinsicP.h>** header file instead of the **<X11/Intrinsic.h>** header file.

The applications should also include the additional headers for each widget class to be used, such as the **<X11/Label.h>** or **<X11/Scroll.h>** header file. The Intrinsics object library file is named the **libXt.a** file.

Using Enhanced X-Windows to Define Widgets

The fundamental data type of the Enhanced X-Windows Toolkit is the *widget*, a combination of a window and its associated semantics. A widget is dynamically allocated and contains state information. Every widget belongs to one *widget class* that is allocated statically and initialized. The widget class contains the operations allowed on widgets of that class.

Logically, a widget is a rectangle with associated input and output semantics. Some widgets display information, such as text or graphics, while others are containers for other widgets, such as a menu box. Some widgets are output only and do not react to pointer or keyboard input, while others change their display in response to input and can call functions attached to them by an application. The user can alter much of the input and output of a widget, such as fonts, colors, sizes, and border widths.

A *widget instance* is composed of two parts:

- A data structure that contains instance-specific values.
- A class structure that contains information that is applicable to all widgets of that class.

Each widget class is logically composed of the procedures and data that is associated with all widgets belonging to that class. These procedures and data can be inherited by subclasses.

Physically, a widget class is a pointer to a structure. The contents of this structure are constant for all widgets of the widget class, even though the values can vary from widget class to widget class. (Here, *constant* means the class structure is initialized at compile-time and never changed, except for a one-time class initialization and in-place compilation of resource lists. Compilation occurs when the first widget of the class or subclass is created.) A widget instance is allocated and initialized by the **XtCreateWidget** subroutine.

The organization of the declarations and code for a new widget class between a public **.h** file, a private **.h** file, and the implementation **.c** file is described in Defining Widget Classes. The predefined widget classes adhere to the conventions found in Defining the Core Widget.

There are three predefined widget classes:

Core widget class	Defines the fields common to all widgets. All widget classes are subclasses of Core, which is defined by the **CoreClassPart** and **CorePart** structures.
Composite widget class	Exists as a subclass of the Core class and defines widgets that act as containers of other widgets. Composite widgets are defined by the **CompositeClassPart** and **CompositePart** structures.
Constraint widget class	Exists as a subclass of the Composite class and defines widgets that maintain additional state data for each child widget. This data can include client-defined contraints on the child widget's geometry. Constraint widgets are defined by the **ConstraintClassPart** and **ConstraintPart** structures.

Using Enhanced X-Windows to Name Widgets

The Intrinsics library provide the ability to create new widgets and organize a set of widgets into an application. Use the following guidelines for naming new widgets:

- A *record component* name is written in lowercase letters. If a name contains more than one word, the words are connected by _ (underscores). For example, the *background_pixmap* field is a compound record component name.
- The first letter of *procedure* and *type* names is capitalized. Compound type and procedure names are capitalized at the beginning of each part of the name. For example, **XtArgList** is a *compound procedure* name and **XtInputCallbackProc** is a *compound type* name.
- A *resource* name string is spelled the same as the *record component* name, except that capitalization is used for compound names. For the compiler to find spelling errors, each resource name should have a macro definition prefixed with **XtN**. For example, the

background_pixmap field has the corresponding resource name identifier **XtNbackgroundPixmap**, which is defined as the **backgroundPixmap** string. Since many predefined names are listed in the **<X11/StringDefs.h>** file, make sure any new name you create is not in this file.

- A *resource class* string starts with a capital letter and uses capitalization in compound names. For example, "BorderWidth" is a resource class compound name. Each resource class string should have a macro definition prefixed with **XtC**, for example, **XtCBorderWidth**.
- A *resource representation* string is spelled the same as the type name, such as **TranslationTable.** Each representation string should have a macro definition prefixed with **XtR**, for example, **XtRTranslationTable**.
- New widget class names begin with a capital letter. Capitalization is used to form compound words. For example, the following names can be derived from the **AbcXyz** new class name:

Example	**Name of**
AbcXyzPart	A partial widget instance structure.
AbcXyzRec and **_AbcXyzRec**	A complete widget instance structure.
AbcXyzWidget	A widget instance pointer type.
AbcXyzClassPart	A partial class structure.
AbcXyzClassRec and **_AbcXyzClassRec**	A complete class structure.
abcXyzClassRec	A class structure variable.
abcXyzWidgetClass	A class pointer variable.

Action procedures available to translation specifications should follow the same naming conventions as procedures. Capitalize the first letter and use capitalization for compound names, for example, **Highlight** and **NotifyClient**.

Using Enhanced X-Windows with the Core Widget Class

The Core widget class defines the fields common to all widgets. All widget classes are subclasses of Core, which is defined by the **CoreClassPart** and **CorePart** data structures.

All widget classes have the Core class fields as their first component. The prototypical type **WidgetClass** is defined with the Core class fields only. Various routines can cast widget class pointers, as needed, to specific widget class types. For example:

```
typedef struct {
   CoreClassPart core_class;
} WidgetClassRec, *WidgetClass;
```

The predefined class record and pointer for **WidgetClassRec** are the following:

extern WidgetClassRec *widgetClassRec*;
extern WidgetClass *widgetClass*;

The opaque types, **Widget** and **WidgetClass**, and the opaque variable, *widgetClass*, are defined for generic actions on widgets.

All widget instances have the core fields as their first component. The prototypical type **Widget** is defined with the core set of fields only. Various routines can cast widget pointers, as needed, to specific widget types. For example:

```
typedef struct {
   CorePart core;
} WidgetRec, *Widget;
```

Using Enhanced X-Windows with the Composite Widget Class

The Composite widget class exists as a subclass of the Core class and defines widgets that act as containers of other widgets. Composite widgets are defined by the **CompositeClassPart** and **CompositePart** data structures.

Composite widget classes have the composite fields immediately after the core fields:

```
typedef struct {
   CoreClassPart core_class;
   CompositeClassPart composite_class;
} CompositeClassRec, *CompositeWidgetClass;
```

The predefined class record and pointer for **CompositeClassRec** are the following:

extern CompositeClassRec *compositeClassRec*;
extern WidgetClass *compositeWidgetClass;*

The opaque types, **CompositeWidget** and **CompositeWidgetClass**, and the opaque variable, *compositeWidgetClass*, are defined for generic operations on widgets that are a subclass of **CompositeWidget**.

Composite widgets have the following *composite* fields immediately after the core fields:

```
typedef struct {
   CorePart core;
   CompositePart composite;
} CompositeRec, *CompositeWidget;
```

Using Enhanced X-Windows with the Constraint Widget Class

The Constraint widget class exists as a subclass of the Composite class and defines widgets that maintain additional state data for each child widget. This data can include client-defined contraints on the geometry of the child widget. Constraint widgets are defined by the **ConstraintClassPart** and **ConstraintPart** structures.

Constraint widget classes have the following constraint fields immediately after the composite fields:

```
typedef struct {
   CoreClassPart core_class;
   CompositeClassPart composite_class;
   ConstraintClassPart constraint_class;
} ConstraintClassRec, *ConstraintWidgetClass;
```

The predefined class record and pointer for **ConstraintClassRec** are the following:

extern ConstraintClassRec *constraintClassRec*;
extern WidgetClass *constraintWidgetClass*;

The opaque types **ConstraintWidget** and **ConstraintWidgetClass**, and the opaque variable, *constraintWidgetClass*, are defined for generic operations on widgets that are a subclass of **ConstraintWidgetClass**

Constraint widgets have the following *constraint* fields immediately after the composite fields:

```
typedef struct {
   CorePart core;
   CompositePart composite;
   ConstraintPart constraint;
} ConstraintRec, *ConstraintWidget
```

Related Information

The **<X11/Xlib.h>** header file, **<X11/Intrinsic.h>** header file, **<X11/IntrinsicP.h>** header file, **<X11/StringDefs.h>** header file, **<X11/Shell.h>** header file.

The **CompositeClassPart** data structure, **CompositePart** data structure, **ConstraintClassPart** data structure, **ConstraintPart** data structure, **CoreClassPart** data structure, **CorePart** data structure.

Enhanced X-Windows Widget Classes Overview

The *widget_class* field of a widget points to its widget class structure that contains information that is constant for all widgets of that class.

With this class-oriented structure, widget classes do not usually implement procedures directly. Rather, widgets implement procedures available through their widget class structure. These class procedures are invoked by generic procedures that envelop common actions around the procedures implemented by the widget class. Such procedures are applicable to all widgets of that class and to widgets that are subclasses of that class.

All widget classes are a subclass of the Core class and can be subclassed further. Subclassing reduces the amount of code and declarations you write to make a new widget class that is similar to an existing class.

You do not have to describe every resource your widget uses in an **XtResourceList** data structure. Instead, describe the resources your widget has that its superclass does not. Subclasses usually inherit many of the procedures of their superclasses, such as the expose procedure or the geometry handler. On the other hand, subclassing too extensively creates a subclass that does not inherit the procedures of its superclass.

To make good use of subclassing, widget declarations and naming conventions are highly stylized. A widget consists of three files:

- A public **.h** file used by client widgets or applications.
- A private **.h** file used by widgets that are subclasses of the widget.
- A **.c** file that implements the widget class.

Using Enhanced X-Windows Widget Subclassing in Public .h Files

The public **.h** file for a widget class is imported by clients. It contains the following:

- A reference to the public **.h** files for the superclass.
- The names and classes of the new resources to be added to its superclass.
- The class record pointer used to create widget instances.
- The **C** language type, which is used to declare widget instances of this class.
- Entry points for new class methods.

The following example shows the public **.h** file for a possible implementation of the Label widget:

```
#ifndef LABEL_H
#define LABEL_H

/* New resources */
#define XtNjustify        "justify"
#define XtNforeground     "foreground"
#define XtNlabel          "label"
#define XtNfont           "font"
#define XtNinternalWidth  "internalWidth"
#define XtNinternalHeight "internalHeight"

/* Class record pointer */
extern WidgetClass labelWidgetClass;

/* C widget type definition */
typedef struct _LabelRec  *LabelWidget;

/* New class method entry points */
extern void Label SetText();
   /* Widget widget */
   /* String text */

extern String Label GetText();
   /* Widget widget */

#endif LABEL_H
```

Conditionally including the text allows the application to include header files for different widgets without determining if they have already been included as a superclass of another widget.

To accommodate an operating system with file name length restrictions, the name of the public **.h** file is the first 10 characters of the widget class. For example, the public **.h** file for the Constraint widget is the **Constraint.h** file.

Using Enhanced X-Windows Widget Subclassing in Private .h Files

The private **.h** file for a widget is imported by widget classes that are subclasses of the widget. It contains:

- A reference to the public **.h** file for the class.
- A reference to the private **.h** file for the superclass.
- The new fields this widget adds to the widget structure of its superclass.
- The complete widget instance structure for this widget.
- The new fields added to the **Constraint** structure of its superclass if the widget is a subclass of **Constraint.**
- The complete **Constraint** structure if the widget is a subclass of **Constraint**.
- The new fields added to the widget class structure of its superclass.
- The complete widget class structure for this widget.
- The name of the constant for the generic widget class structure.
- An inherit procedure for subclasses where each new procedure in the widget class structure should inherit a superclass operation.

The following example shows the private **.h** file for the Label widget:

```
#ifndef LABELP_H
#define LABELP_H

#include <X11/Label.h>
```

```
/* New fields for the Label widget record */

typedef struct {
/* Settable resources */
   Pixel foreground;
   XFontStruct *font;
   String label;            /* text to display */
   XtJustify justify;
   Dimension internal_width;  /* # of pixels in horizontal
                                 border*/
   Dimension internal_height; /* # of pixels in vertical border */

/* Data derived from resources */
   GC normal_GC;
   GC gray_GC;
   Pixmap gray_pixmap;
   Position label_x;
   Position label_y;
   Dimension label_width;
   Dimension label_height;
   Cardinal label_len;
   Boolean display_sensitive;
} LabelPart;

/* Full instance record declaration */
typedef struct_LabelRec {
   CorePart core;
   LabelPart label;
} LabelRec;

/* Types for label class methods */
typedef void (*LabelSetTextProc)();
   /* Widget widget */
   /* String text */

typedef String (*LabelGetTextProc)();
   /* Widget widget */

/* New fields for the Label widget class record */
typedef struct {
   LabelSetTextProc set_text;
   LabelGetTextProc get_text;
   caddr_t extension;
} LabelClassPart;

/* Full class record declaration */
typedef struct_LabelClassRec {
   CoreClassPart core_class;
   LabelClassPart label_class;
} LabelClassRec;

/* Class record variable */
extern LabelClassRec labelClassRec;

#define LabelInheritSetText((LabelSetTextProc)_XtInherit)
#define LabelInheritGetText((LabelGetTextProc)_XtInherit)
#endif LABELP_H
```

To accommodate operating systems with file name length restrictions, the name of the private **.h** file is the first nine characters of the widget class followed by a capital **P**. For example, the private **.h** file for the Constraint widget is the **ConstrainP.h** file.

Using Enhanced X-Windows Widget Subclassing in .c Files

The **.c** file for a widget contains the structure initializer for the class record variable. This initializer contains the following parts:

- Class information, such as the following:
  - *superclass, class_name, widget_size, class_initialize, class_inited.*
- Data constants, such as the following:
  - *resources* and *num_resources*, *actions* and *num_actions*, *visible_interest, compress_motion, compress_exposure, version.*
- Widget operations, such as the following:
  - *initialize, realize, destroy, resize, expose, set_values, accept_focus*, and any operations specific to the widget.

The superclass field points to the superclass **WidgetClass** record. For direct subclasses of the generic Core widget, superclass should be initialized to the address of the **widgetClassRec** structure. The superclass is used for class chaining operations and for inheriting or enveloping the operations of a superclass. Some of the fields are the following:

class_name	Specifies the text name for this class used by the resource manager. For example, the Label widget has the string "Label". More than one widget class can use the same text class name.
widget_size	Specifies the size of the corresponding widget structure, not the size of the class structure.
version	Specifies the toolkit version number. This field is used at run time to check consistency of the Toolkit and widgets in an application. It should be set to the value returned by the **XtVersion** in the widget class initialization.

To run widgets that are backwards compatible with previous Intrinsic versions, use the **XtVersionDontCheck** value. This value turns off version checking for those widgets.

extension	Used for upwards compatibility. If you add additional fields to class parts, all subclass structure layout change and complete recompilation is required. To avoid recompilation, an extension field at the end of each class part can point to a record that contains any additional class information that is required.

The following is a compressed version of the **.c** file for the Label widget.

```
/* Resources specific to Label */

#define XtRJustify "Justify"
static XtResource resources[] = {
   {XtNforeground, XtCForeground, XtRPixel, sizeof(Pixel),
     XtOffset(LabelWidget, label.foreground), XtRString,
     XtDefaultForeground,
   {XtNfont, XtCFont, XtRFontStruct, sizeof(XFontStruct*),
     XtOffset(LabelWidget, label.font), XtRString, XtDefaultFont},
   {XtNlabel, XtCLabel, XtRString, sizeof(String),
     XtOffset(LabelWidget, label.label), XtRString, NULL},
                 .
                 .
}

/* Forward declarations of procedures */

static void ClassInitialize();
static void Initialize();
static void Realize();
static void SetText();
static void GetText();
    .
    .
    .
/* Class record constant */
LabelClassRec labelClassRec = {
 {
  /* Core class fields */
   /* superclass              */ (WidgetClass) &widgetClassRec,
   /* class_name              */ "Label",
   /* widget_size             */ sizeof(LabelRec),
   /* class_initialize        */ ClassInitialize,
   /* class_part_initialize */ NULL,
   /* class_inited            */ False,
   /* initialize              */ Initialize,
   /* initialize_hook         */ NULL,
   /* realize                 */ Realize,
   /* actions                 */ NULL,
   /* num_actions             */ 0,
   /* resources               */ resources,
   /* num_resources           */ XtNumber(resources),
   /* xrm_class               */ NULLQUARK,
   /* compress_motion         */ True,
   /* compress_exposure       */ True,
   /* compress_enterleave     */ True,
   /* visible_interest        */ False,
   /* destroy                 */ NULL,
   /* resize                  */ Resize,
   /* expose                  */ Redisplay,
   /* set_values              */ SetValues,
   /* set_values_hook         */ NULL,
   /* set_values_almost       */ XtInheritSetValuesAlmost,
   /* get_values_hook         */ NULL,
   /* accept_focus            */ NULL,
   /* version                 */ XtVersion,
   /* callback_offsets        */ NULL,
   /* tm_table                */ NULL,
   /* query_geometry          */ XtInheritQueryGeometry,
   /* display_accelerator     */ NULL,
   /* extension               */ NULL,
 },
 {
 /* Label class fields */
   /* get_text                */ GetText,
   /* set_text                */ SetText,
   /* extension               */ NULL,
 }
};

/* Class record pointer */
```

```
WidgetClass labelWidgetClass = (WidgetClass) &labelClassRec;

/* New method access routines */
void Label SetText (widget, text)
   Widget widget;
   String text;
{
   Label WidgetClass lwc = (Label WidgetClass) XtClass(widget);
   XtCheckSubclass(widget, labelWidgetClass, NULL);
   *(lwc->label_class.set_text)(widget, text)
}

/*Private procedures*/
   .
   .
   .
```

Using Enhanced X-Windows to Chain Superclass Operations

Some fields defined in the widget class structure are self-contained and independent of the values for these fields defined in superclasses. Among these are the following:

- class_name
- accept_focus
- class_initialize
- compress_motion
- widget_size
- compress_exposure
- realize
- compress_enterleave
- visible_interest
- set_values_almost
- resize
- tm_table
- expose
- *version*.

Some fields defined in the widget class structure are accessed only after their corresponding superclass value has been accessed. This is called downward superclass chaining. In this case, the invocation of a single operation first accesses the Core class, then the subclass, and so on down the class chain to the widget class of the widget. These superclass-to-subclass fields are the following:

- class_part_initialize
- get_values_hook
- initialize_hook
- initialize
- set_values_hook
- set_values
- *resources*.

In addition, for subclasses of the **Constraint** class, the resources field of the **ConstraintClassPart** structure is chained from the **Constraint** class down to the subclass.

Some fields defined in the widget class structure are accessed before their corresponding superclass value has been accessed. This is called upward superclass chaining. In this case, the invocation of a single operation actually first accesses the widget class, then its superclass, and so on up the class chain to the Core class. The subclass-to-superclass fields are the following:

- destroy
- *actions.*

Using Enhanced X-Windows to Initialize a Widget Class

Many class records can be initialized completely at compile time. In some cases, however, a class may need to register type converters or perform other kinds of "one-time" initializations.

The C language does not have initialization procedures that are called automatically when a program starts up. Thus, a widget class can declare a *class_initialize* procedure that will be called (once) automatically by the X Toolkit. A class initialization procedure pointer is of the **XtProc** type.

Specify the value of **NULL** in the *class_initialize* field to indicate that a widget class does not have a class initialization procedure.

In addition to class initializations, some widget classes must must perform additional initializations for fields in their part of the class record. These initializations are performed in the *class_part_initialize* procedure, which is stored in the *class_part_initialize* field. The *class_part_initialize* procedure pointer is of the **XtWidgetClassProc** type.

During class intialization, the class part initialization procedure for the class and its superclasses are called in superclass-to-subclass order on the class record.

These procedures perform necessary dynamic initializations to the part of the record for their class. The most common procedure is the resolution of any inherited methods defined in the class. For example, if a widget class *C* has superclasses Core, Composite, *A*, and *B*, the class record for *C* is passed first to the *class_part_initialize* record of Core. This resolves any inherited Core methods and compiles the textual representations of the resource list and action table that are defined in the class record. Next, the Composite *class_part_initialize* procedure initializes the composite part of *C*'s class record. Finally, the *class_part_initialize* procedures for *A, B,* and *C* are called in order.

Specify the value of **NULL** in the *class_part_initialize* field for classes that do not define new class fields or that do not need extra processing for their class fields.

All widget classes must begin with the *class_inited* field set to the value of **False**, whether or not they have a class initialization procedure.

The first time a widget of a class is created, the **XtCreateWidget** subroutine ensures that the widget class and all superclasses are initialized in superclass-to-subclass order by checking that each *class_inited* field is the value of **False**, by calling the *class_initialize* field and the *class_part_initialize* field procedures for the class and all its superclasses.

Then, the Instrinsics library sets the *class_inited* field to the value of **True**. After the one-time initialization, a class structure is constant.

The following is the class initialization procedure for the Label widget.

```
static void ClassInitialize()
{
   XtQEleft = XrmStringToQuark("left");
   XtQEcenter = XrmStringToQuark("center");
   XtQEright = XrmStringToQuark("right");

   XtAddConverter(XtRString, XtRJustify, CvtStringToJustify,
    NULL, 0);

}
```

A class is initialized the first time a widget of that class, or any subclass, is created. If the class initialization procedure registers type converters, these type converters are not available until the first widget is created.

Using Enhanced X-Windows to Inherit Superclass Operations

A widget class can use any of the self-contained operations of its superclass rather than implementing its own code. The inherited operations of the superclass most frequently used include the following:

- expose
- realize
- insert_child
- delete_child
- geometry_manager
- *set_values_almost.*

To inherit an *xyz* operation, specify the **XtInherit***Xyz* constant in your class record. Every class that declares a new procedure in its widget class part must provide for inheriting the procedure in its *class_part_initialize* procedure.

The special chained operations *initialize*, *set_values* and *destroy*, which are declared in the Core record, do not have inherit procedures. Widget classes that do nothing beyond what their superclass does, specify the value of **NULL** for chained procedures in their class records.

Inheriting compares the value of the field with a known, special value. If a match occurs, it copies the superclass value for that field. This special value is usually the Intrinsics library **_XtInherit** internal value cast to the appropriate type. (The **_XtInherit** internal value issues an error message if it is called directly.)

For example, the Composite class private include file contains these definitions:

```
#define XtInheritGeometryManager ((XtGeometryHandler) _XtInherit)
#define XtInheritChangeManaged ((XtWidgetProc) _XtInherit)
#define XtInheritInsertChild ((XtArgsProc) _XtInherit)
#define XtInheritDeleteChild ((XtWidgetProc) _XtInherit)
```

The Composite *class_part_initialize* procedure begins as follows:

```
static void CompositeClassPartInitialize(widgetClass)
   WidgetClass widgetClass;
{
   register CompositeWidgetClass
wc=(CompositeWidgetClass)widgetClass;
   CompositeWidgetClass
super=(CompositeWidgetClass)wc—>core_class.superclass;

   if (wc—>composite_class.geometry_manager ==
      XtInheritGeometryManager) {
      wc—>composite_class.geometry_manager = super—>
      composite_class.geometry_manager;
   }

   if (wc—>composite_class.change_managed ==
      XtInheritChangeManaged) {
      wc—>composite_class.change_managed = super—>
      composite_class.change_managed;
   }
   .
   .
   .
```

The defined inherit constants for Core are the following:

- XtInheritRealize
- XtInheritResize
- XtInheritExpose
- XtInheritSetValuesAlmost
- XtInheritAcceptFocus
- **XtInheritDisplayAccelerator.**

The defined inherit constants for Composite are the following:

- XtInheritGeometryManager
- XtInheritChangeManaged
- XtInheritInsertChild
- **XtInheritDeleteChild.**

Using Enhanced X-Windows to Call Superclass Operations

A widget class sometimes needs to call a superclass operation that usually is not chained. For example, the expose procedure for a widget might call the expose procedure of its superclass and then perform more work of its own. Composite classes with fixed children can implement *insert_child* by calling their superclass *insert_child* procedure then, calling the **XtManageChild** subroutine to add the child to the managed list.

The class procedure should call its own superclass procedure, not the superclass of the widget. The class procedure should use its own class pointers and not the class pointers of the widget. This technique is referred to as enveloping the operation of the superclass.

Related Information

The **XtResourceList** data structure.

The **XtProc** data type, **XtWidgetClassProc** data type.

The **XtCheckSubclass** macro, **XtClass** macro, **XtSuperclass** macro.

The **XtIsSubclass** subroutine, **XtManageChild** subroutine.

Enhanced X-Windows Widget Creation Overview

The creation of widget instances is a three-phase process:

1. The widgets are allocated and initialized with resources and are optionally added to the managed subset of their parent.
2. All composite widgets are notified of their managed children in a bottom-up traversal of the widget tree.
3. The widgets create windows that then get mapped.

To start the first phase, the application calls the **XtCreateWidget** subroutine for all its widgets and adds them, as appropriate, to the managed set of their parent widget by calling the **XtManageChild** subroutine. To avoid an O(n**^{2}) creation process where each composite widget lays itself out each time a widget is created and managed, parent widgets are not notified of changes in their managed set during this phase.

After all widgets have been created, the application calls the **XtRealizeWidget** subroutine on the top-level widget to start the second and third phases. The **XtRealizeWidget** subroutine first recursively traverses the widget tree in a post-order (bottom-up) traversal, and then notifies each composite widget that has one or more managed children through its *change_managed* procedure.

Notifying a parent widget about its managed set involves geometry layout and, possibly, geometry negotiation. A parent widget must deal with constraints on its size that are imposed from above, as when a user specifies the application window size. A parent widget also deals with suggestions made from below, as when a primitive child widget computes its preferred size. Any clash between the two can cause geometry changes to ripple in both directions through the widget tree. The parent widget may force some of its children widgets to change size and position and may issue geometry requests to its own parent widget to accommodate all its children widgets better. Until this process is settled, placement on the screen is uncertain.

Consequently, to avoid unnecessary requests to the X Server to move windows after creation, windows are not actually created in the first and second phases.

Finally, the **XtRealizeWidget** subroutine starts the third phase by making a pre-order (top-down) traversal of the widget tree, allocates an Enhanced X-Windows window to each widget by means of its realize procedure, and maps the managed widgets.

Using Enhanced X-Windows to Create and Merge Argument Lists

Many of the Intrinsics subroutines need to be passed pairs of resource names and values, called an argument list. These are passed as an **ArgList** structure, which contains the **XtArgVal** data structure.

If the size of the resource is less than, or equal to, the size of an **XtArgVal** data structure, the resource value is stored directly in the *Value* parameter. Otherwise, the *Value* parameter is a pointer to the resource value.

Using Enhanced X-Windows to Create a Widget Instance

The **XtCreateWidget** subroutine creates an instance of a widget. This routine performs many of the boilerplate operations of widget creation.

Using Enhanced X-Windows to Create an Application Shell Instance

Applications can have multiple top-level widgets, which can be on different screens. To create several independent windows, applications use the **XtAppCreateShell** subroutine, which creates a top-level widget that is the root of a widget tree.

To create multiple top-level shells within a single logical application, use one of the following methods:

- Designate one shell as the real top-level shell and create the others as pop-up children of the first with the **XtCreatePopupShell** subroutine. Use this method when you need to designate a main window. It leads to resource specifications as follows:

xmail.geometry:...	***(main window)***
xmail.read.geometry:...	***(read window)***
xmail.compose.geometry:...	***(compose window).***

- Have all shells as pop-up children of an unrealized top-level shell.
  Use this method when you do not need to create a main window. It leads to resource specifications as follows:

xmail.headers.geometry:...	***(headers window)***
xmail.read.geometry:...	***(read window)***
xmail.compose.geometry:...	***(compose window).***

Using Enhanced X-Windows to Initialize a Widget Instance

The initialize procedure pointer in a widget class is of the **XtInitProc** type.

An initialization procedure performs the following:

- Allocates space and copies any resources that are referenced by address. For example, if a widget has a field that is a **String** (**char** *), it cannot depend upon the characters at that address to remain constant, but must dynamically allocate space for the string and copy it to the new space.

Note: Do not allocate space for or copy callback lists.

- Computes values for unspecified resource fields. For example, if width and height are the value of 0, the widget computes a width and height based on other resources. This is the only time a widget can directly assign its own width and height.
- Computes values for uninitialized non-resource fields that are derived from resource fields. For example, **GC**s that the widget uses are derived from resources like background, foreground, and font.

An initialization procedure can also check certain fields for internal consistency, such as a specification of a color map for a depth that does not support that color map.

Initialization procedures are called in superclass-to-subclass order. Most of the initialization code for a specific widget class deals with fields defined in that class and not with fields defined in its superclasses.

If a subclass does not need an initialization procedure becauseit does not need to perform nay of the above operations, it can specify the value of **NULL** for the initialize field in the class record.

Sometimes a subclass may want to overwrite values filled in by its superclass. In particular, size calculations of a superclass are often incorrect for a subclass. In this case, the subclass must modify or recalculate fields declared and computed by its superclass. For example, a subclass can visually surround its superclass display. In this case, the width and height calculated by the superclass initialize procedure are too small and need to be incremented by the size of the surround. The subclass needs to know if its superclass size was calculated by the superclass or was specified explicitly. All widgets must place themselves into whatever size is explicitly given, but widgets should compute a reasonable size if no size is requested.

In this example, the subclass with the visual surround can see if the width and height in the request widget are the value of 0. If so, the subclass adds its surround size to the width and height fields in the new widget. If not, the subclass must use the size originally specified.

The new widget becomes the actual widget instance record. Therefore, if the initialization procedure needs to call any routines that operate on a widget, it should specify "new" as the widget instance. The request widget should never be modified.

Using Enhanced X-Windows to Initialize a Constraint Widget Instance

The constraint initialize procedure is of the **XtInitProc** type. The values passed to the parent constraint initialization procedure are the same values passed to the class widget initialization procedure of the child widget.

The constraint initialize procedure should compute any constraint fields derived from constraint resources. It can make further changes to the widget to make the widget conform to the specified constraints, for example, changing the size or position of the widget.

Specify the value of **NULL** for the *initialize* field of the **ConstraintClassPart** in the class record if a constraint class does not need a constraint initialization procedure.

Using Enhanced X-Windows to Initialize Non-widget Data

The value of the *initialize_hook* field allows a widget instance to initialize non-widget data using information from the specified argument list. For example, the Text widget has subparts that are not widgets, but these subparts have resources that can be specified by the resource file or an argument list. The *initialize_hook* field is of the **XtArgsProc** type.

Using Enhanced X-Windows to Realize Widgets

All the Instrinsics library routines and all widget routines should work with realized or unrealized widgets. To realize a widget instance, use the **XtRealizeWidget** subroutine. The widget procedure must create a window for the widget. In addition, some widget procedures, such as **set_values**, may choose to operate differently after the widget has been realized.

The **XtCreateWidget**, **XtRealizeWidget**, **XtManageChildren**, **XtUnmanageChildren**, and **XtDestroyWidget** subroutines maintain the following invariants:

- If a widget is realized, then all managed children of the widget are realized.
- If a widget is realized, then all managed children of the widget that are *mapped_when_managed* are mapped.

To determine whether a widget has been realized, use the **XtIsRealized** subroutine.

Using Enhanced X-Windows to Create a Window for a Widget Instance

A widget class can inherit its realize procedure from its superclass during class initialization. The realize procedure defined for the Core widget class calls the **XtCreateWindow** subroutine, with the passed *value_mask* and *attributes* fields, and with the **windowClass** and the *visual* field set to **CopyFromParent**. Both **CompositeWidgetClass** and **ConstraintWidgetClass** inherit this realize procedure, and most new widget subclasses can do the same.

The most common noninherited realize procedures set the *bit_gravity* field in the mask, set attributes to the appropriate value, and then create the window. For example, Label sets the *bit_gravity* field to **WestGravity**, **CenterGravity** or **EastGravity**. Consequently, shrinking a Label moves the bits appropriately, and no **Expose** event is needed for repainting.

If a composite widget wants to realize its children in a particular order, typically to control the stacking order, it calls the **XtRealizeWidget** subroutine on its children in the appropriate order from within its own realize procedure.

Rather than call the **XCreateWindow** subroutine directly, a realize procedure should call the Toolkit analog **XtCreateWindow**. This routine, which simplifies the creation of windows for widgets, evaluates the following fields in the Core widget structure:

- x
- y
- width
- height
- depth
- *screenparent –> core.window*

Widgets that have children but are not a subclass of the *compositeWidgetClass* are responsible for calling the **XtRealizeWidget** subroutine for their children, usually from within the realize procedure.

Note: Because realize is not a chained operation, the widget classrealize procedure must update the **XSetWindowAttributes** data structure with all the appropriate fields from non-Core superclasses.

Using Enhanced X-Windows to Obtain Window Information

The Core widget definition contains the screen and window IDs. The window field can be the value of **NULL** for a while.

The display pointer, the parent widget, the screen pointer, and the window of a widget are returned by the following macros, which take a widget and return the specified value:

- The **XtDisplay** macro
- The **XtParent** macro
- The **XtScreen** macro
- The **XtWindow** macro.

Several window attributes are cached locally in the widget. Thus, the widgets can be set by the Resource Manager and the **XtSetValues** subroutine as well as the **XtCreateWindow** subroutine or subroutines that derive structures from these values. Examples of such structures are *depth* for deriving pixmaps and *background_pixel* for deriving **GCs**.

The *x, y, width, height*, and *border_width* window attributes are available to geometry managers. These fields are maintained synchronously inside the Toolkit. When an **XConfigureWindow** subroutine is issued on the widget window at the request of the parent, these values are updated immediately rather than whenever the server generates a **ConfigureNotify** event. In fact, most widgets do not have **SubstructureNotify** turned on. This ensures that all geometry calculations are based on the internally consistent Toolkit, rather than on either:

- Inconsistencies updated by asynchronous **ConfigureNotify** events.
- Wasteful consistencies, which slow the process because geometry managers ask the server for window sizes each time they need to layout their managed children widgets.

Using Enhanced X-Windows to Unrealize Widgets

After widgets have been realized, they can also be unrealized. To destroy the windows associated with a widget and its descendants, use the **XtUnrealizeWidget** subroutine.

Using Enhanced X-Windows to Destroy Widgets

To destroy widgets, the Toolkit can:

- Destroy all the pop-up children of the widget being destroyed and destroy all children of composite widgets.
- Remove and unmap the widget from its parent.
- Call the callback procedures that have been registered to trigger when the the widget is destroyed.
- Minimize the number of things a widget has to deallocate when destroyed.
- Minimize the number of **XDestroyWindow** calls.

To destroy a widget instance, use the **XtDestroyWidget** subroutine.

Using Enhanced X-Windows to Add and Delete Destroy Callbacks

When an application needs to perform additional processing during the destruction of a widget, it should register a destroy callback procedure for the widget. The destroy callback list is identified by the **XtNdestroyCallback** resource name.

The following example calls the **XtAddCallback** subroutine to add to a widget the application-supplied destroy callback procedure, **ClientDestroy**, which has the `client_data` data.

```
XtAddCallback (widget, XtNdestroyCallback, ClientDestroy,
client_data)
```

Similarly, the next example removes the application-supplied destroy callback procedure, **ClientDestroy** by calling the **XtRemoveCallback** subroutine.

```
XtRemoveCallback (widget, XtNdestroyCallback, ClientDestroy,
client_data)
```

The **ClientDestroy** procedure in both of the preceding examples is of the **XtCallbackProc** data type.

Using Enhanced X-Windows to Deallocate Dynamic Constraint Data

The destroy procedure pointer in the **CoreClassPart** data structure is the **XtWidgetProc** data type.

The destroy procedure is called for a widget whose parent is a subclass of the *constraintWidgetClass*. The destroy procedures are called in subclass-to-superclass order, starting at the parent of the widget and ending at the *constraintWidgetClass*. Therefore, a constraint destroy procedure of a parent should deallocate only storage that is specific to the constraint subclass and not the storage allocated by any of its superclasses. Use the value of **NULL** for the constraint destroy procedure for the parent not requiring deallocation of constraint storage.

The **destroy** procedures are called in subclass-to-superclass order. Therefore, a **destroy** procedure for a widget should only deallocate storage specific to the subclass and not to its superclasses. It should deallocate only those resources that are explicitly created by the subclass. If a widget does not need to deallocate any storage, the **destroy** procedure in its widget class record can be the value of **NULL**..

Deallocating storage includes, but is not limited to, the following:

- Calling the **XtFree** subroutine on dynamic storage allocated with the **XtMalloc, XtCalloc,** and other subroutines.
- Calling the **XFreePixmap** subroutine on pixmaps created with direct X calls.
- Calling the **XtDestroyGC** subroutine on graphics contexts allocated with the **XtGetGC** subroutine.
- Calling the **XFreeGC** subroutine on graphics contexts allocated with direct X calls.
- Calling the **XtRemoveEventHandler** subroutine on event handlers added with the **XtAddEventHandler** subroutine.
- Calling the **XtRemoveTimeout** subroutine on timers created with the **XtAppAddTimeout** subroutine.
- Calling the **XtDestroyWidget** subroutine for each child if the widget has children widgets and is not a subclass of the **compositeWidgetClass** class.

Using Enhanced X-Windows to Exit an Application

Use the **XtDestroyApplicationContext** subroutine to end all toolkit applications. Then, exit normally. Or, use the **XtUnmapWidget** subroutine on each top-level shell widget.

Using Enhanced X-Windows with Callbacks

Applications and other widgets (clients) often want to register a procedure with a widget that is called under certain conditions. For example, when a widget is destroyed, every procedure in its *destroy_callbacks* filed is called to notify clients of its impending destruction.

Every widget has a *destroy_callbacks* field. Additional callback lists can be defined as needed. For example, the Command widget has a callback list to notify clients when the button has been activated. Callback procedure fields for use in callback lists are of the **XtCallbackProc** type.

To pass a callback list as an argument in a call to the **XtCreateWidget**, the **XtSetValues,** or the **XtGetValues** subroutines, a client specifies the address of a null-terminated list of the **XtCallbackList** data type.

For example, the callback list for the `A` and `B` procedures with the `clientDataA` and `clientDataB` client data, respectively looks like the following:

```
static XtCallbackRec callbacks[] = {
   {A, (caddr_t) clientDataA},
   {B, (caddr_t) clientDataB},
   {(XtCallbackProc) NULL, (caddr_t) NULL}
};
```

Although callback lists are passed by address in argument lists, the Intrinsics library are aware of callback lists. The application-specific *initialize* and *set_values* procedures should not allocate memory for the callback list, as the Intrinsics library do this automatically by using a different structure for their internal representation.

Whenever a widget contains a callback list for use by clients, it also exports in its public **.h** file the resource name of the callback list. Applications and client widgets never access callback list fields directly. Instead, they identify the desired callback list using the exported resource name. All callback manipulation routines check that the requested callback list is implemented by the widget.

For the Intrinsics library to find and correctly handle callback lists, the lists should always be declared with the **XtRCallback** resource type.

The following subroutines perform operations involving callback lists:

XtAddCallback	Adds a callback procedure to a widget's callback list.
XtAddCallbacks	Adds a list of callback procedures to a widget's callback list.
XtRemoveCallback	Deletes a callback procedure from a widget's callback list.
XtRemoveCallbacks	Deletes a list of callback procedures from a widget's callback list.
XtRemoveAllCallbacks	Deletes all callback procedures from a widget's callback list.
XtCallCallbacks	Calls the procedures in a widget's callback list.
XtHasCallbacks	Determines the status of a widget's callback list.

Related Information

The **CoreClassPart** data structure, **XSetWindowAttributes** data structure, **XtArgVal** data structure, **XtCallbackList** data structure.

The **XtArgsProc** data type, **XtCallbackProc** data type, **XtInitProc** data type, **XtRealizeProc** data type, **XtWidgetProc** data type.

The **XtDisplay** macro, **XtIsRealized** macro **XtParent** macro, **XtScreen** macro, **XtWindow** macro.

The **XConfigureWindow** subroutine, **XFreeGC** subroutine, **XFreePixmap** subroutine, **XtAddCallback** subroutine, **XtAddCallbacks** subroutine, **XtAddEventHandler** subroutine, **XtAppAddTimeout** subroutine, **XtAppCreateShell** subroutine, **XtCallCallbacks** subroutine, **XtCalloc** subroutine, **XtCreatePopupShell** subroutine, **XtCreateWidget** subroutine, **XtCreateWindow** subroutine, **XtDestroyApplicationContext** subroutine, **XtDestroyGC** subroutine, **XtDestroyWidget** subroutine, **XDestroyWindow** subroutine, **XtFree** subroutine, **XtGetGC** subroutine, **XtGetSubresources** subroutine, **XtGetValues** subroutine, **XtHasCallbacks** subroutine, **XtMalloc** subroutine, **XtManageChild** subroutine, **XtManageChildren** subroutine, **XtMergeArgLists** subroutine, **XtRealizeWidget** subroutine. **XtRemoveAllCallbacks** subroutine, **XtRemoveCallback** subroutine, **XtRemoveCallbacks** subroutine, **XtRemoveEventHandler** subroutine, **XtRemoveTimeout** subroutine, **XtSetArg** subroutine, **XtSetValues** subroutine, **XtUnmanageChildren** subroutine, **XtUnmapWidget** subroutine, **XtUnrealizeWidget** subroutine.

Enhanced X-Windows Widget Instantiation Overview

A collection of widget instances constitutes a widget tree. The shell widget returned by the **XtAppCreateShell** subroutine is the root of the widget tree instance. The widgets with one or more children are the intermediate nodes of that tree, and the widgets with no children of any kind are the leaves of the widget tree. With the exception of pop-up children, this widget tree instance defines the associated Enhanced X-Windows tree.

Widgets are either primitive or composite. Either kind can have children widgets.

The Intrinsics library provides management mechanisms for constructing and interfacing between composite widgets, their children, and other clients. The Intrinsics library also recursively perform many operations, such as realization and destruction, on composite widgets and all of their children.

While a composite widget can, in unusual circumstances, have no children, they usually have at least one.

Pop-up children can be attached to any widget, regardless of class. Each pop-up child has a window that is a child of the root window so that the pop-up window is not clipped.

Widgets with no children of any kind are leaves of a widget tree. Widgets with one or more children are intermediate nodes of a widget tree. The shell widget returned by the **XtAppCreateShell** subroutine is the root window of a widget tree.

The normal children of the widget tree (including pop-ups) define the associated Enhanced X-Windows tree.

A widget tree is manipulated by several Intrinsics subroutines:

- The **XtRealizeWidget** subroutine, which traverses the tree downward and recursively realizes all pop-up widgets and children of composite widgets.
- The **XtDestroyWidget** subroutine, which traverses the tree downward and destroys all pop-up widgets and children of composite widgets.
- The **XtMakeGeometryRequest** subroutine, which traverses the tree one level upward and calls the geometry manager responsible for the child geometry of a widget.

  The subroutines that get and modify resources traverse the tree upward to determine the inheritance of resources from the ancestors of a widget.

To facilitate up-traversal of the widget tree, each widget has a pointer to its parent widget. However, the **XtAppCreateShell** subroutine returns Shell widgets with a parent pointer of the value of **NULL**..

To facilitate down-traversal of the widget tree, each composite widget has a pointer to a list of children widgets. This list includes all normal children created, not just the subset of children that are managed by the composite geometry manager of the widget. In addition, every widget has a pointer to a list of pop-up children widgets.

Using Enhanced X-Windows to Initialize Toolkit

Before an application can call any of the Intrinsics subroutines, it must initialize the Toolkit by using the following:

- The **XtToolkitInitialize** subroutine, which initializes the Toolkit internals.
- The **XtCreateApplicationContext** subroutine, which initializes the per application state.

- The **XtDisplayInitialize** or **XtOpenDisplay** subroutine, which initialize the per display state.
- The **XtAppCreateShell** subroutine, which creates the initial widget.

Multiple instances of Toolkit applications can be implemented by a single program in a single address space. Each instance must be able to read input and dispatch events independently of any other instance.

Applications may need multiple display connections or the same widgets on multiple screens. To achieve this, the Intrinsics library define application contexts, which provide the information necessary to distinguish one application instance from another. A list of **Display** pointers for that application is the major component of an application context. The **XtAppContext** application context type is opaque to clients.

Using Enhanced X-Windows to Load Resource Database

The **XtDisplayInitialize** subroutine loads the application's resource database for this display, host, and application combination. Each resource database is kept on a per-display basis, and is loaded from five sources, if they exist, in the following order:

1. Application-specific class resource file on the local host.
2. Application-specific user resource file on the local host.
3. Resource property on the server or user preference resource file on the local host.
4. Per-host user environment resource file on the local host.
5. Application command line (**argv**).

The application-specific resource file name is constructed from the class name of the application and points to a site-specific resource file that is installed by the site manager, usually when the application is installed. The application resource file is **/usr/lpp/X11/appdefaults/***class*, where *class* is the application class name. This file should be provided by the application developer and may be required for the application to run properly.

The application-specific user resource file name is constructed from the class name of the application and points to a user-specific resource file. This file is owned by the application and typically stores user customizations. This file name is constructed by using the value of the user's **XAPPLRESDIR** environment variable and appending *class* to it, where *class* is the application name. If this environment variable is not defined, the value defaults to the user's home directory. If the resource file exists, it is merged into the resource database. The file may be provided with the application or constructed by the user.

The server resource file is the contents of the X Server RESOURCE_MANAGER property as returned by the **XOpenDisplay** subroutine. The server resource file is constructed entirely by the user and contains both display-independent and display-specific user preferences. If the RESOURCE_MANAGER property does not exist, the resource file in the user's home directory is used. This file is usually called the **.Xdefaults** file. If the resource file exists, it is merged into the resource database.

The user's environment resource file name is constructed by using the value of the user's *XENVIRONMENT* variable for the full path of the file. If the user's environment resource file exists, it is merged into the resource database. (This file name is user and host-specific.)

If the *XENVIRONMENT* variable does not exist, the **XtDisplayInitialize** subroutine searches the user's home directory for the **.Xdefaults-***host*: file, where *host* is the name of the system where the application is running. If this file exists, it is merged into the resource database.

The environment resource file should contain process-specific resource specifications intended to supplement the user-preference specifications in the server resource file.

Use the **XtDatabase** subroutine to obtain the resource database for a particular display. This routine returns the fully merged resource database built by the **XtDisplayInitialize** subroutine.

Related Information

The **XtAppCreateShell** subroutine, **XtCloseDisplay** subroutine, **XtCreateApplicationContext** subroutine, **XtDatabase** subroutine, **XtDestroyApplicationContext** subroutine, **XtDestroyWidget** subroutine, **XtDisplayInitialize** subroutine, **XtMakeGeometryRequest** subroutine, **XtRealizeWidget** subroutine, **XtWidgetToApplicationContext** subroutine.

Enhanced X-Windows Composite Widgets Overview

Composite widgets are a subclass of the **compositeWidgetClass**. Composite widget functions are implemented directly by the widget class or indirectly by the Intrinsics subroutines. Composite widgets can have an arbitrary number of children widgets and consequently control more than primitive widgets. In general, composite widgets handle the following:

- Overall management of child widgets from creation to destruction.
- Destruction of descendants when the composite widget is destroyed.
- Physical arrangement, or geometry management, of a displayable subset of managed children widgets.
- Mapping and unmapping of a subset of the managed child widgets.

Overall management is handled by the generic procedures of the **XtCreateWidget** and **XtDestroyWidget** subroutines. The **XtCreateWidget** subroutine adds child widgets to a parent widget by calling the *insert_child* procedure of the parent widget. The **XtDestroyWidget** subroutine removes child widgets from a parent widget by calling the *delete_child* procedure of the parent widget and also ensures that all children widgets of a destroyed composite widget get destroyed.

Only a subset of the total number of child widgets is actually managed by the geometry manager and possibly visible. For example, a multibuffer composite editor widget can allocate one child widget for each file buffer while only displaying a small number of the existing buffers. Windows in this displayable subset are managed windows and enter into geometry manager calculations. The other children widgets are unmanaged windows and are not mapped.

Child widgets are added to the managed set with the **XtManageChild** and **XtManageChildren** subroutines, and removed from the managed set by using the **XtUnmanageChild** and **XtUnmanageChildren** subroutines. These procedures notify the parent widget to recalculate the physical layout of its child widgets by calling the *change_managed* procedure of the parent widget. The **XtCreateManagedWidget** subroutine calls the **XtCreateWidget** and **XtManageChild** subroutines on the result.

Most managed child widgets are mapped, but some widgets can be in a state where they take up physical space, but do not show anything. Managed widgets are not mapped automatically if the *map_when_managed* field is the value of **False**. The default for the *map_when_managed* field is the value of **True**. This field is changed by using the **XtSetMappedWhenManaged** subroutine.

Each composite widget class has a geometry manager responsible for calculating where the managed child widgets appear within the window of the composite widget. Geometry management techniques fall into the following four classes:

Fixed boxes	Have a fixed number of child widgets created by the parent. All of these children widgets are managed and none make geometry manager requests.
Homogeneous boxes	Treat all child widgets equally and apply the same geometry constraints to each child widget. Many clients insert and delete widgets freely.
Heterogeneous boxes	Place each child widget in a specific location. This location is not usually specified in pixels because the window can be resized, but is expressed in terms of the relationship between a child widget and the parent widget or between the child widget and other specific children widgets. Heterogeneous boxes are usually subclasses of **Constraint**.
Shell boxes	Have only one child widget. This child widget is exactly the size of the shell. The geometry manager must communicate with the window manager if it exists. The box must also accept the **ConfigureNotify** events when the window size is changed by the window manager.

Using Enhanced X-Windows to Verify the Class of a Composite Widget

To determine if a given widget is a subclass of the **Composite** widget, use the **XtIsComposite** subroutine. This subroutine is equivalent to the **XtIsSubclass** subroutine with the *compositeWidgetClass* field specified.

Using Enhanced X-Windows to Add Children to a Composite Widget

To add a child widget to the list of children widgets of a parent widget, the **XtCreateWidget** subroutine calls the *insert_child* class routine of the parent widget. The *insert_child* procedure pointer in a composite widget is of the **XtWidgetProc** type.

Most composite widgets inherit the operation of their superclass. The *insert_child* routine of the composite widget calls the *insert_position* procedure and inserts the child widget at the specified location.

Some composite widgets define their own *insert_child* routine so that they can order their child widgets in some convenient way, create companion controller widgets for a new widget, or limit the number or type of their child widgets.

If there is not enough room to insert a new child widget in the child widget list (if *num_children* = *num_slots*), the *insert_child* procedure must first reallocate the list and update the *num_slots* field. The *insert_child* procedure calculates a location for the child widget, inserts the child in the location, and increments the *num_children* field.

Using Enhanced X-Windows to Insert Children in a Specific Order

Instances of composite widgets must specify the order in which their child widgets are kept. For example, an application may require a set of command buttons in some logical order grouped by function or buttons representing file names in alphabetical order.

The *insert_position* procedure pointer in a composite widget instance is of the **XtOrderProc** type.

Composite widgets that allow clients to order their child widgets are usually homogeneous boxes. These widgets call the *insert_position* procedure of the widget instance from the *insert_child* procedure of the class to determine where a new child widget is placed in the child widget array of the composite widget. A client of a composite class can apply different sorting criteria to widget instances of the class, passing in a different *insert_position* procedure when it creates each composite widget instance.

The return value of the *insert_position* procedure indicates how many children widgets go before the widget.

- When a value of 0 is returned, the widget goes before all other children widgets.
- When the *num_children* field is returned, the widget goes after all other children widgets. The *insert_position* default returns the *num_children* field. This default can be overridden by the resource list of a widget or by the argument list provided when the composite widget is created.

Using Enhanced X-Windows to Delete Children of Composite Widgets

To remove a child widget from the child widget array of a parent widget, the **XtDestroyWidget** subroutine calls the *delete_child* procedure of the composite parent widget. The *delete_child* procedure pointer is of the **XtWidgetProc** type.

Most widgets inherit the *delete_child* procedure from their superclass. Composite widgets that create companion widgets define a *delete_child* procedure to remove these companion widgets.

Using Enhanced X-Windows to Manage Children in a Managed Set

The Intrinsics library provide a set of generic subroutines for adding widgets to the managed set of a composite widget and removing widgets from the managed set of a composite widget. These generic subroutines call the *change_managed* procedure of the widget. The *change_managed* procedure pointer is of the **XtWidgetProc** type.

Using Enhanced X-Windows to Add Children to a Managed Set

To add a list of widgets to the geometry-managed displayable subset of its composite parent widget, the application must first create the widgets using the **XtCreateWidget** subroutine and then call the **XtManageChildren** subroutine.

Managing child widgets is independent of ordering, creating, and deleting child widgets. The layout routine of the parent widget considers child widgets with a managed field equal to the value of **True** and ignores all other child widgets. Some composite widgets, especially fixed boxes, call the **XtManageChild** subroutine from their *insert_child* procedure.

If the parent widget is realized, the *change_managed* procedure of the parent widget confirms that its set of managed child widgets has changed. The parent widget can reposition and resize any of its child widgets. The position of a child widget is changed by a call to the **XtMoveWidget** subroutine. This subroutine updates the *x* and *y* fields and calls the **XMoveWindow** subroutine if the widget is realized.

To change the size or border width of any of its child widgets, a composite widget calls the **XtResizeWidget** subroutine. This procedure first updates the **Core** fields, and then calls the **XConfigureWindow** subroutine if the widget is realized.

To add a single child widget to the list of managed child widgets of a parent widget, first create the child widget with the **XtCreateWidget** subroutine, and then use the **XtManageChild** subroutine. The **XtManageChild** subroutine constructs a **WidgetList** of length one and calls the **XtManageChildren** subroutine.

To create and manage a child widget in a single procedure, use the **XtCreateManagedWidget** subroutine. This subroutine calls the **XtCreateWidget** and **XtManageChild** subroutines.

Using Enhanced X-Windows to Remove Children from a Managed Set

To remove child widgets from the managed list of a parent widget, use the **XtUnmanageChildren** subroutine. The specified widgets are not destroyed by this subroutine and can be added to the managed list again at a later time.

To remove a single child widget from the managed set of a parent widget, use the **XtUnmanageChild** subroutine. The **XtUnmanageChild** subroutine constructs a widget list of length one and calls the **XtUnmanageChildren** subroutine.

These generic functions are low-level subroutines used by generic composite widget building subroutines. Composite widgets also can provide widget-specific high-level procedures for the creation and management of child widgets.

Using Enhanced X-Windows to Determine if a Widget is Managed

To determine the managed state of a given child widget, use the **XtIsManaged** macro. The **XtIsManaged** macro is implemented as a boolean macro and as a function. Both forms return the value of **True** if the specified child widget is managed and the value of **False** if the child widget is not managed.

Using Enhanced X-Windows to Control Widget Mapping

A managed widget is usually mapped. The mapping can be overridden by setting the **XtNmappedWhenManaged** resource for the widget when it is created or by setting the *map_when_managed* field to the value of **False**.

To change the value of the *map_when_managed* field of a widget, use the **XtSetMappedWhenManaged** subroutine. The **XtSetMappedWhenManaged** subroutine is equivalent to calling the **XtSetValues** subroutine and setting the new value for the mappedWhenManaged resource.

An alternative to using the **XtSetMappedWhenManaged** subroutine to control mapping is having the client set the *mapped_when_managed* field to the value of **False** and calling the **XtMapWidget** and **XtUnmapWidget** subroutines explicitly to map and unmap a widget.

Using Enhanced X-Windows with Constrained Composite Widgets

Constraint widgets are a subclass of the **compositeWidgetClass**. Constraint widgets manage the geometry of their child widgets based on constraints associated with each child widget. Constraints can be simple, such as the maximum width and height that the parent widget allows the child to occupy, or complex, such as instructions on how other child widgets should change if this child widget is moved or resized.

Constraint widgets let a parent widget define resources supplied for its child widgets. For example, if a constraint parent widget defines the maximum sizes for its child widgets, these new size resources are retrieved for each child widget as if they were resources defined by the child widget. Constraint resources can be included in an argument list or resource file just like any other resource for the child widget.

Constraint widgets have all the functionality of standard composite widgets. These widgets also process and act upon the constraint information associated with each of their child widgets.

Every widget has a *constraints* field. This field enables widgets and the Intrinsics library to keep track of the constraints associated with a child widget. The field contains the address of a parent-specific structure that contains constraint information about the child widget. If the parent widget is not a subclass of the **constraintWidgetClass**, the *constraints* field of the child widget is the value of **NULL**.

Subclasses of a Constraint widget can add more *constraint* fields to their superclass. To add more constraint fields, you define the constraint records in a private **.h** file using the same conventions as for widget records. For example, a widget that needs to maintain a maximum width and height for each child widget might define its constraint record as the following:

```
typedef struct {
   Dimension max_width, max_height;
} MaxConstraintPart;

typedef struct {
   MaxConstraintPart max;
} MaxConstraintRecord, *MaxConstraint;
```

A subclass of this widget that needs to maintain a minimum size can define its constraint record as the following:

```
typedef struct {
   Dimension min_width, min_height;
} MinConstraintPart;

typedef struct {
   MaxConstraintPart max;
   MinConstraintPart min;
} MaxMinConstraintRecord, *MaxMinConstraint;
```

Constraints are allocated, initialized, deallocated, and maintained by the Intrinsics library. The constraint class record part has several entries that facilitate this control. All entries in the **ConstraintClassPart** resource are information and procedures defined and implemented by the parent widget and called whenever actions are performed on the child widgets of the parent widget.

The **XtCreateWidget** subroutine uses the *constraint_size* field to allocate a constraint record when a child widget is created. This field specifies the number of bytes occupied by a constraint record. The **XtCreateWidget** subroutine also uses the constraint resources to fill in resource fields in the constraint record associated with a child widget. It then calls the constraint *initialize* procedure so the parent widget can compute constraint fields derived from constraint resources and moves or resizes the child widget to conform to the given constraints.

The **XtGetValues** and **XtSetValues** subroutines use the constraint resources to get or set the values of constraints associated with a child widget. The **XtSetValues** subroutine then calls the constraint *set_values* procedures so the parent widget can recompute derived constraint fields and move or resize the child widget as appropriate.

The **XtDestroyWidget** subroutine calls the constraint *destroy* procedure to deallocate dynamic storage associated with a constraint record. The constraint record must not be deallocated by the constraint *destroy* procedure, as the **XtDestroyWidget** subroutine deallocates the constraint record automatically.

Related Information

The **XtOrderProc** data type, **XtWidgetProc** data type.

The **XtIsComposite** macro, **XtIsManaged** macro.

The **XConfigureWindow** subroutine, **XMoveWindow** subroutine, **XtCreateManagedWidget** subroutine, **XtCreateWidget** subroutine, **XtDestroyWidget** subroutine, **XtGetValues** subroutine, **XtIsSubclass** subroutine **XtManageChild** subroutine, **XtManageChildren** subroutine, **XtMapWidget** subroutine, **XtMoveWidget** subroutine, **XtResizeWidget** subroutine, **XtSetMappedWhenManaged** subroutine, **XtSetValues** subroutine, **XtUnmanageChild** subroutine, **XtUnmanageChildren** subroutine, **XtUnmapWidget** subroutine.

Enhanced X-Windows Shell Widgets Overview

Shell widgets hold the top-level widgets of an application that communicate with the window manager. Shell widgets are designed to be as invisible as possible. A client application must create a shell widget it requires but does not handle the sizing of the widget.

If a shell widget is resized from the outside, typically by a window manager, the widget resizes its child widget automatically. Similarly, if the child widget of the shell widget wants to change size, it can make a geometry request to the shell widget to negotiate the size change with the outer environment. Client applications should not change the size of their shells directly.

Using Enhanced X-Windows to Define Shell Widgets

Widgets negotiate size and position with their parent widget. Widgets at the top-level of the hierarchy do not have parent widgets. Instead, these widgets must deal with the outside world. To provide for this, each top-level widget is encapsulated in a special widget called a **Shell** widget.

Shell widgets are a subclass of the **Composite** widget. They encapsulate other widgets and allow the encapsulated widgets to avoid the geometry clipping imposed by the standard parent-child window relationship. Shell widgets also provide a layer of communication with the window manager.

There are a total of seven classes of shell widgets. Three of the classes are internal and are not instantiated or subclassed by client applications. Four of the classes are defined for public use.

The three shell widget classes allocated for internal use are the following:

Shell	Provides the base class for shell widgets and the fields needed for all types of shells. The **Shell** widget class is a direct subclass of the **CompositeWidgetClass**.
WMShell	Contains fields needed by the common window manager protocol. The **WMShell** widget class is a subclass of the **Shell** widget class.
VendorShell	Contains fields used by vendor-specific window managers. The **VendorShell** widget class is a subclass of the **WMShell** widget class.

The four shell widget classes defined for public use are the following:

OverrideShell	Used for shell windows that completely bypass the window manager. A pop-up menu shell is an example of this class. The **OverrideShell** widget class is a subclass of the **Shell** widget class.
TransientShell	Used for shell windows manipulated by the window manager but not iconified separately. These shell windows are iconified by the window manager only if the main application shell is iconified. A dialog box associated with an application shell is an example of this class. The **TransientShell** widget class is a subclass of the **VendorShell** widget class.
TopLevelShell	Used for normal top-level windows. All additional top-level widgets required by an application are examples of this class. The

	TopLevelShell widget class is a subclass of the **VendorShell** widget class.
ApplicationShell	Used by the window manager to define the top-level window of a separate application instance. The **ApplicationShell** widget class is a subclass of the **TopLevelShell** widget class.

Using Enhanced X-Windows to Define ShellClassPart Fields

None of the shell widget classes has the following additional fields:

```
typedef struct { caddr_t extension; } ShellClassPart,
   OverrideShellClassPart, WMShellClassPart,
   VendorShellClassPart, TransientShellClassPart,
   TopLevelShellClassPart, ApplicationShellClassPart;
```

Shell widget classes have the (empty) *shell* fields immediately following the *composite* fields.

The predefined class records and pointers for shells are the following:

extern ShellClassRec ***shellClassRec***;
extern OverrideShellClassRec ***overrideShellClassRec***;
extern WMShellClassRec ***wmShellClassRec***;
extern VendorShellClassRec ***vendorShellClassRec***;
extern TransientShellClassRec ***transientShellClassRec***;
extern TopLevelShellClassRec ***topLevelShellClassRec***;
extern ApplicationShellClassRec ***applicationShellClassRec***;

The predefined class pointers for shells are the following:

extern WidgetClass ***shellWidgetClass***;
extern WidgetClass ***overrideShellWidgetClass***;
extern WidgetClass ***wmShellWidgetClass***;
extern WidgetClass ***vendorShellWidgetClass***;
extern WidgetClass ***transientShellWidgetClass***;
extern WidgetClass ***topLevelShellWidgetClass***;
extern WidgetClass ***applicationShellWidgetClass***;

The following opaque types and opaque variables are defined for generic operations on widgets that are a subclass of the **ShellWidgetClass**:

TYPES	VARIABLES
ShellWidget	***shellWidgetClass***
OverrideShellWidget	***overrideShellWidgetClass***
WMShellWidget	***wmShellWidgetClass***
VendorShellWidget	***vendorShellWidgetClass***
TransientShellWidget	***transientShellWidgetClass***
TopLevelShellWidget	***topLevelShellWidgetClass***
ApplicationShellWidget	***applicationShellWidgetClass***
ShellWidgetClass	
OverrideShellWidgetClass	
WMShellWidgetClass	
VendorShellWidgetClass	
TransientShellWidgetClass	
TopLevelShellWidgetClass	
ApplicationShellWidgetClass	

Using Enhanced X-Windows to Define ShellPart Fields

The various shells have additional fields defined in their widget records.

The full definitions of the various shell widgets have the *shell* fields following the *composite* fields:

Using Enhanced X-Windows Default Values for ShellPart Fields

Default values for fields common to all classes of public shells are filled in by the **Shell** resource lists and the **Shell** *initialize* procedures.

Related Information

The **ApplicationShellClassRec** data structure, **ApplicationShellPart** data structure, **OverrideShellClassRec** data structure, **OverrideShellPart** data structure, **ShellClassRec** data structure, **ShellPart** data structure, **ShellWidget** data structure, **TopLevelShellClassRec** data structure, **TopLevelShellPart** data structure, **TransientShellClassRec** data structure, **TransientShellPart** data structure, **VendorShellClassRec** data structure, **VendorShellPart** data structure, **WMShellClassRec** data structure **WMShellPart** data structure.

Enhanced X-Windows Pop-up Widgets Overview

Pop-up widgets are used to create windows outside of the window hierarchy defined by the widget tree. A pop-up is created and attached to its parent widget differently than a standard child widget. The window associated with a pop-up child widget is defined as a descendant of the root window such that it is not clipped by the parent window of the pop-up widget.

A parent widget of a pop-up widget does not actively manage its pop-up child widgets. A pop-up can be popped up from its parent widget and from other widgets.

The list of pop-up child widgets of a parent widget is maintained in the *popup_list* field in the **CorePart** structure of the parent widget. This pop-up list defines placement in the widget hierarchy for the pop-up to get resources and provides a list of child widgets for use by the **XtDestroyWidget** subroutine.

A **Composite** widget can have both normal and pop-up child widgets. The term *child* in this context refers to a standard geometry-managed child widget on the child widget list. A pop-up child widget refers to a child widget on a pop-up list.

There are three kinds of pop-up widgets , as follows:

modeless pop-up	Usually visible to the window manager. To the user, this pop-up looks like any other application. A modeless dialog box is an example of the modeless pop-up.
modal pop-up	Disables user-event processing by the application except for events that occur in the pop-up. This pop-up can be visible to the window manager. An example of a modal pop-up is a modal dialog box.
spring-loaded pop-up	Disables user-event processing by all applications except for events that occur in the pop-up. This pop-up is not visible to the window manager. An example of a spring-loaded pop-up is a menu.

Modal pop-ups and spring-loaded pop-ups are similar and should be coded the same. A single widget, such as a **ButtonBox** or a **Menu**, can be used as a modal pop-up and as a spring-loaded pop-up within the same application. The main difference between the modal pop-up and the spring-loaded pop-up is that each spring-loaded pop-up is brought up with the pointer and requires different processing by the Intrinsics library because of the grab caused by the pointer button.

Any kind of pop-up can pop up other widgets. Modal and spring-loaded pop-ups can constrain user events to the most recent pop-up or to any modal or spring-loaded pop-ups currently mapped.

Pop-up widget classes are subclasses of the **Shell** widget class responsible for communicating with the window manager.

Using Enhanced X-Windows to Create a Pop-up Shell

For a widget to pop up, it must be the child widget of a pop-up widget shell. The pop-up shell communicates with the window manager when geometry requests are made. It is also responsible for handling the bookkeeping associated with actual pop-up and pop-down actions. Each pop-up shell can have only one pop-up child widget. The shell and child widgets together are referred to as the pop-up. When a child widget pop-up is required, the pop-up shell must be specified.

To create a pop-up shell, use the **XtCreatePopupShell** subroutine. This subroutine attaches a pop-up shell to the pop-up list of the parent widget.

A spring-loaded pop-up invoked from a translation table must already exist at the time the transaction is called so the translation manager can locate the shell by name. Other pop-ups can be created as needed. Creating a pop-up later can be useful when you pop up an unspecified number of pop-ups. You can determine if an unused shell or a shell not currently popped up exists, and then create a new shell if necessary.

Using Enhanced X-Windows to Create Pop-up Children

The child widget of a pop-up shell can be created anytime after the pop-up shell is created. The pop-up child widget can be treated as either a static or dynamic entity.

An application can create the child widget of the pop-up shell at application startup. This action is appropriate for pop-up child widgets composed of a fixed set of widgets. The application can change the state of the subparts of the pop-up child widget as the application state changes. For example, if an application creates a static menu, it can call the **XtSetSensitive** or the **XtSetValues** subroutines on any of the buttons that make up the menu. Creating the pop-up child widget at application startup means that pop-up time can be minimized, especially if the application calls the **XtRealizeWidget** subroutine on the pop-up shell at startup time. When the pop-up child widget is needed, the widgets that make up the child widget already exist and only need to be mapped.

Alternatively, an application can postpone the creation of a pop-up child widget until it is needed. This minimizes application startup time and allows the pop-up child widget to reconfigure itself dynamically each time it is popped up. In this case, the pop-up child widget creation routine polls the application to find out if it should change any of its subparts.

Creating a pop-up child widget does not map the pop-up.

All shells have pop-up and pop-down callbacks. These callbacks provide mechanisms to make last-minute changes to a pop-up child widget before it is popped up or to change the child after it is popped down. Excessive use of pop-up callbacks can slow the pop-up action.

Using Enhanced X-Windows to Map a Pop-up Widget

Pop-ups can be popped up and mapped through several mechanisms:

- Call to the **XtPopup** subroutine.

- Use of a supplied callback such as the **XtCallbackNone**, the **XtCallbackNonexclusive**, or the **XtCallbackExclusive** subroutines.
- Use of the standard **MenuPopup** translation action.

A call to the **XtPopup** subroutine maps a pop-up from within an application. The **XtCallbackNone**, the **XtCallbackNonexclusive**, and the **XtCallbackExclusive** subroutines map a pop-up from the callback list of a specified widget. The **MenuPopup** translation action pops up a menu when a pointer button is pressed or when the pointer is moved into a window defined to produce the menu.

Using Enhanced X-Windows to Unmap a Pop-up Widget

Pop-ups can be popped down and unmapped through several mechanisms:

- Call to the **XtPopdown** subroutine.
- Use of the supplied callback **XtCallbackPopdown** subroutine.
- Use of the standard **MenuPopdown** translation action.

A call to the **XtPopdown** subroutine pops down and unmaps a pop-up from within an application. The **XtCallbackPopdown** subroutine pops down and unmaps pop-ups popped up by the **XtCallbackNone**, the **XtCallbackNonexclusive**, and the **XtCallbackExclusive** subroutines. The **MenuPopdown** translation action pops down and unmaps a spring-loaded menu when a pointer button is released or when the pointer is moved into a window.

Related Information

The **XtCallbackPopdown** callback procedure.

The **CorePart** data structure, **XtPopdownID** data structure.

The **XtCallbackExclusive** subroutine, **XtCallbackNone** subroutine, **XtCallbackNonexclusive** subroutine, **XtCreatePopupShell** subroutine, **XtDestroyWidget** subroutine, **XtPopdown** subroutine, **XtPopup** subroutine, **XtRealizeWidget** subroutine, **XtSetSensitive** subroutine, **XtSetValues** subroutine.

The **MenuPopdown** translation action, **MenuPopup** translation action.

Enhanced X-Windows Widget Geometry Overview

A widget does not directly control its size or location. The position of child widgets is usually the responsibility of the parent widget. However, the child widgets often have the best idea of their optimal sizes and preferred locations.

To resolve physical layout conflicts between sibling widgets or a child widget and its parent widget, the Intrinsics library provide a geometry management mechanism. Most **Composite** widgets have a *geometry_manager* field in the widget class record that is responsible for the size, position, and stacking order of the child widgets. Exceptions to this are fixed boxes which create child widgets and can ensure that these child widgets never initiate a geometry request.

Using Enhanced X-Windows to Initiate Geometry Changes

Parent widgets, child widgets, and clients initiate geometry changes differently. Because a parent widget has absolute control of the geometry of its child widgets, it changes the geometry directly by calling the **XtMoveWidget**, the **XtResizeWidget**, or the **XtConfigureWidget** subroutine. A child widget must ask its parent widget for a geometry change by calling the **XtMakeGeometryRequest** or the **XtMakeResizeRequest** subroutine to convey its request to its parent widget. An application or other client code initiates a geometry change by calling the **XtSetValues** subroutine on the appropriate geometry fields, thereby giving the widget the opportunity to modify or reject the client request before it gets propagated to the parent widget and the opportunity to respond appropriately to the reply of the parent widget.

When the geometry manager of a parent widget received a request from a child widget for a change in the child widgets size, position, border width, or stacking depth, the geometry manager can do the following:

- Allow the request
- Disallow the request
- Suggest a compromise.

When the geometry manager is asked to change the geometry of a child widget, the geometry manager can also rearrange and resize any or all of the other children widgets that it controls. The geometry manager can move child widgets around freely using the **XtMoveWidget** subroutine. When it resizes a child widget other than the one making the request, it calls the **XtResizeWidget** subroutine. The geometry manager can simultaneously move and resize a child widget with a single call to the **XtConfigureWidget** subroutine.

Often, geometry managers find they can satisfy a request only by reconfiguring a widget they do not control, such as when the **Composite** widget wants to change its own size. In this case, the geometry manager makes a request to the geometry manager of the parent. Geometry requests can cascade this way to arbitrary depth.

Because such cascaded arbitration of widget geometry can involve extended negotiation, windows are not actually allocated to widgets at application startup until all widgets are satisfied with their geometry.

Note: The Intrinsics library treatment of stacking requests is deficient in several areas. Stacking requests for unrealized widgets are granted but will have no effect. In addition, there is no way to use the **XtSetValues** subroutine to generate a stacking geometry request.

Note: After a successful geometry request (one that returns the **XtGeometryYes** value), a widget does not know whether or not its *resize* procedure has been called. Widgets should have *resize* procedures that can be called more than once without ill effects.

Using Enhanced X-Windows to Make General Geometry Manager Requests

To make a general geometry manager request from a widget, use the **XtMakeGeometryRequest** subroutine.

The return codes from geometry managers are as follows:

```
typedef enum _XtGeometryResult {
   XtGeometryYes,
   XtGeometryNo,
   XtGeometryAlmost,
   XtGeometryDone,
} XtGeometryResult;
```

Using Enhanced X-Windows in Making Resize Requests

Use the **XtMakeResizeRequest** subroutine to make a simple resize request from a widget. This subroutine is basically a simple interface to the **XtMakeGeometryRequest** subroutine.

Using Enhanced X-Windows to Manage Potential Geometry Changes

Sometimes a geometry manager cannot respond to a geometry request from a child widget without first making a geometry request to the widget's own parent widget (the requestor's grandparent widget). If the request to the grandparent widget would allow the parent widget to satisfy the original request, the geometry manager can make the intermediate geometry request as if it were the originator.

On the other hand, if the geometry manager already has determined that the original request cannot be completely satisfied (for example, if it always denies position changes), it tells the grandparent widget to respond to the intermediate request without actually changing the geometry because it does not know if the child widget will accept the compromise. To accomplish this, the geometry manager uses the **XtCWQueryOnly** value in the intermediate request.

When the **XtCWQueryOnly** value is used, the geometry manager needs to cache enough information to exactly reconstruct the intermediate request. If the grandparent widgets response to the intermediate query is the **XtGeometryAlmost** return code, the geometry manager caches the entire reply geometry in the event that the child widget accepts the parent widgets compromise.

If the grandparent widgets response is the **XtGeometryAlmost** return code, the entire reply geometry from the grandparent widget is cached when the **XtCWQueryOnly** value is not used. If the geometry manager is still able to satisfy the original request, it immediately accepts the grandparent widgets compromise and acts on the child widgets request. If the grandparent widgets compromise geometry is insufficient to allow the child widgets request and if the geometry manager is willing to offer a different compromise to the child widget, the grandparent widgets compromise is not accepted until the child widget has accepted the new compromise.

A compromise geometry returned with the **XtGeometryAlmost** return code is guaranteed only for the next call to the same widget. Therefore a cache of size one is sufficient.

Using Enhanced X-Windows to Manage Child Geometry

The *geometry_manager* procedure manages the geometry of a child. The procedure can change the *x*, *y*, *width*, *height*, and *border_width* values of a widget. The *geometry_manager* procedure pointer in a composite widget class is of the **XtGeometryHandler** type.

Sometimes the geometry manager can satisfy change requests only in part or in some altered form. For instance, the manager may fill a subset of requests such as size or position, or a request in a modified form, such as resizing a widget's window but not exactly to the requested size.

In such cases, changes are made only if the child widget agrees to the compromises. The geometry manager fills in the *geometry_return* parameter with the actual changes it can make, including an appropriate mask, and returns the **XtGeometryAlmost** return code. If a bit in the *geometry_return->request_mode* is the value of 0, the geometry manager does not change the corresponding value if the *geometry_return* parameter is used immediately in a new request. If a bit is one, the geometry manager changes that element to the corresponding value in the *geometry_return* parameter. More bits may be set in the *geometry_return* parameter than in the original request if the geometry manager in set to change other fields upon compromise acceptance from the child widget.

When the **XtGeometryAlmost** value is returned, the widget must decide if the compromise suggested in the *geometry_return* parameter is acceptable. If it is, the widget makes a call to the **XtMakeGeometryRequest** subroutine to change its geometry.

The geometry manager must grant the request if the next geometry request from this child widget uses the *geometry_return* parameter box filled in by an **XtGeometryAlmost** return code, if there have been no intervening geometry requests on either its parent widget or any of its other children widgets, and if possible. If the child widget gives its comfirmation immediately with the returned geometry, it usually gets an answer of the **XtGeometryYes** return code. However, the user's window manager can affect the final outcome.

To return the **XtGeometryYes** return code, the geometry manager frequently rearranges the position of other managed child widgets by calling the **XtMoveWidget** subroutine. However, some geometry managers change the size of other managed child widgets by calling the **XtResizeWidget** or the **XtConfigureWidget** subroutine. If the **XtCWQueryOnly** value is specified, the geometry manager must return how it will react to this geometry request without actually moving or resizing any widgets.

Geometry managers do not assume that the *request* and *geometry_return* parameters point to independent storage. The caller is permitted to use the same field for both. The geometry manager allocates its own temporary storage if necessary.

Using Enhanced X-Windows to Manage Widget Placement and Size

A child widget can be resized by its parent widget at any time. Widgets usually need to know when they have changed size so that they can layout their displayed data to match the new size.

If a class need not recalculate anything when a widget is resized, it can specify the value of **NULL** for the *resize* field in its class record. This case occurs only for widgets with simple display semantics. The *resize* procedure takes a widget as its only argument. The *x*, *y*, *width*, *height*, and *border_width* fields of the widget contain the new values. The *resize* procedure recalculates the layout of internal data as required. For example, a centered label in a window that changes size recalculates the starting position of the text.

The widget must make changes requested by the *resize* procedure. A widget must not issue an **XtMakeGeometryRequest** or **XtMakeResizeRequest** subroutine call from its *resize* procedure.

The **XtResizeWidget** subroutine is used to resize a sibling widget of a child widget making a geometry request. This procedure updates the geometry fields in the widget, configures the window if the widget is realized, and calls the *resize* procedure of the child widget to notify it of the changes. The *resize* procedure pointer is of the **XtWidgetProc** type.

The **XtResizeWindow** subroutine is used to resize a child widget that already has the new values of its width, height, and border width fields.

The **XtConfigureWidget** subroutine is used to move and resize a sibling widget of a child widget making a geometry request.

The **XtMoveWidget** subroutine is used to move a sibling widget of a child widget making a geometry request.

Using Enhanced X-Windows to Handle Preferred Geometry

Parent widgets can sometimes adjust their layouts to accommodate the preferred geometries of their child widgets. They can use the **XtQueryGeometry** subroutine to obtain the preferred geometry and use or ignore any portion of the response. Parent widgets are expected to call the **XtQueryGeometry** subroutine in their layout routine and wherever other information is significant after a *change_managed* procedure is called.

Related Information

The **XtGeometryResult** data structure, **XtWidgetGeometry** data structure.

The **XtGeometryHandler** data type, **XtWidgetProc** data type.

The **XConfigureWindow** subroutine, **XtConfigureWidget** subroutine, **XtMakeGeometryRequest** subroutine, **XtMakeResizeRequest** subroutine, **XtMoveWidget** subroutine, **XtQueryGeometry** subroutine, **XtRealizeWidget** subroutine, **XtResizeWidget** subroutine, **XtResizeWindow** subroutine, **XtSetValues** subroutine.

Enhanced X-Windows Event Manager Overview

While Enhanced X-Windows allows the reading and processing of events anywhere in an application, widgets in the Toolkit can neither directly read events nor grab the server or pointer. Widgets register procedures to call when an event or class of events occurs in that widget.

A typical application consists of start-up code followed by an event loop that reads events and dispatches them by calling the procedures that widgets have registered. The default event loop provided by the Intrinsics library is the **XtAppMainLoop** subroutine.

The event manager is a collection of subroutines that:

- Add or remove event sources other than X Server events, particularly timer interrupts and file input.
- Query the status of event sources.
- Add or remove procedures to be called when an event occurs for a particular widget.
- Enable and disable the dispatching of user-initiated events, such as keyboard and pointer events, for a particular widget.
- Constrain the dispatching of events to a cascade of pop-up widgets.
- Call the appropriate set of procedures currently registered when an event is read.

Most widgets do not need to call the event handler subroutines explicitly. The normal interface to X events is through the higher-level translation manager. The translation manager maps sequences of X events, with modifiers, into procedure calls. Applications rarely use event manager routines other than the **XtAppMainLoop** subroutine.

Using Enhanced X-Windows to Add and Delete Additional Event Sources

While most applications are driven only by X events, some applications need to incorporate other sources of input into the X Toolkit event handling mechanism. The event manager provides routines to integrate notification of timer events and file data that is pending into this mechanism.

Using Enhanced X-Windows to Add and Remove Input Sources

The **XtAppAddInput** and the **XtRemoveInput** subroutines control input gathering from files. The application registers the files with the Intrinsics library *read* routine. When input is pending on one of the files, the registered callback procedures are called.

The **XtAppAddInput** subroutine registers a new file as an input source for a given application.

The **XtRemoveInput** subroutine discontinues a source of input.

Using Enhanced X-Windows to Add and Remove Timeouts

The timeout facility notifies the application or the widget through a callback procedure that a specified time interval has elapsed. Time-out values are uniquely identified by an interval ID.

The **XtAppAddTimeOut** subroutine creates a timeout value.

The **XtRemoveTimeOut** subroutine clears a timeout value.

Using Enhanced X-Windows to Compress Events Using Event Filters

The event manager provides filters that can be applied to X user events. The filters screen out events that are redundant or temporarily unwanted. These filters handle Pointer motion compression, Enter/leave compression, and Exposure compression.

Using Enhanced X-Windows with Pointer Motion Compression

Widgets can have a hard time keeping up with pointer motion events and rarely need notification of every motion event. Setting the *compress_motion* widget class field to the value of **True** filters out redundant motion events. With this setting, when a request for an event returns a motion event, the Intrinsics library checks for other motion events immediately following the current one and ignores all but the last of them.

Using Enhanced X-Windows with Enter/Leave Compression

Sometimes pairs of enter and leave events with no intervening events can be ignored. An example is when a user moves the pointer across a widget without stopping in it. Setting the *compress_enterleave* widget class field to the value of **True** filters out pairs of enter and leave events with no intervening events. With this setting, enter and leave events are not delivered to the client if they are found together in the input queue.

Using Enhanced X-Windows with Exposure Compression

Many widgets prefer to process a series of exposure events as a single expose region rather than as individual rectangles. Widgets with complex displays might use the expose region as a clip list in a graphics context. Widgets with simple displays might ignore the region entirely and redisplay their whole window or get the bounding box from the region and redisplay only that rectangle.

In either case, these widgets do not require information on partial expose events. If the *compress_exposure* field in the widget class structure is set to the value of **True**, the event manager calls the widget's *expose* procedure only once for each series of exposure events. In this case, all the **Expose** events are accumulated into a region. When the final **Expose** event in a series (one with count the value of **0**) is received, the event manager replaces the rectangle in the event with the bounding box for the region and calls the widget's *expose* procedure, passing the modified exposure event and the region.

If the *compress_exposure* field is the value of **False**, the event manager calls the widget's *expose* procedure for every exposure event, passing it the event and a region value of **NULL**..

Using Enhanced X-Windows to Constrain Events

Modal widget pop-up lock out user input to an application except direct input to that widget.

When a modal menu or modal dialog box is popped up using the **XtPopup** subroutine, user events such as keyboard and pointer events that occur outside the modal widget are delivered to the modal widget or ignored. In this case, user events are not delivered to a widget outside of the modal widget.

Menus can pop up submenus, and dialog boxes can pop up further dialog boxes to create a pop-up cascade. In this case, user events are delivered to one of several modal widgets in the cascade.

Display-related events are delivered outside the modal cascade so that expose events keep the application display current. Events that occur within the cascade are delivered normally.

User events that are delivered to the most recent spring-loaded shell in the cascade when they occur outside the cascade are called remap events. These remap events include the **KeyPress**, **KeyRelease**, **ButtonPress**, and **ButtonRelease** events. The **MotionNotify**, **EnterNotify**, and **LeaveNotify** events are ignored if they occur outside the cascade. All other events are delivered normally.

The **XtPopup** subroutine uses the **XtAddGrab** and **XtRemoveGrab** subroutines to constrain user events to a modal cascade and subsequently to remove a grab when the modal widget goes away. There is usually no need to call them explicitly.

The modal cascade is used by the **XtDispatchEvent** subroutine when it tries to dispatch a user event. When at least one modal widget is in the widget cascade, the **XtDispatchEvent** subroutine first determines if the event should be delivered. It starts at the most recent cascade entry and follows the cascade up to and including the most recent cascade entry added with the *Exclusive* parameter set to the value of **True**.

This subset of the modal cascade along with all descendants of these widgets is the active subset. User events that occur outside the widgets in this subset are ignored or remapped. Modal menus with submenus generally add a submenu widget to the cascade with the *Exclusive* parameter set to the value of **False**. Modal dialog boxes that must restrict user input to the most deeply nested dialog box add a subdialog widget to the cascade with the *Exclusive* parameter set to the value of **True**. User events that occur within the active subset are delivered to the appropriate widget, usually a child or further descendant of the modal widget.

Regardless of where they occur on the screen, remap events are always delivered to the most recent widget in the active subset of the cascade with its *spring_loaded* field set to the value of **True**, if any such widget exists.

The **XtAddGrab** subroutine redirects user input to a modal widget.

The **XtRemoveGrab** subroutine removes the redirection of user input to a modal widget.

Using Enhanced X-Windows to Focus Events on a Child

The **XtSetKeyboardFocus** subroutine redirects keyboard input to a child widget of a **Composite** widget without calling the **XSetInputFocus** subroutine. Widgets requiring the input focus can call the **XSetInputFocus** subroutine explicitly.

To allow outside agents to cause a widget to get the input focus, each widget exports an *accept_focus* procedure. The widget returns whether or not it actually took the focus, so that the parent widget can give the focus to another widget.

Widgets that need to know when they lose the input focus must use the **Xlib** library focus notification mechanism explicitly by specifying translations for the **FocusIn** and **FocusOut** events. Widgets not requiring information on input focus have their *accept_focus* procedure pointer set to the value of **NULL**..

The **XtCallAcceptFocus** subroutine calls the *accept_focus* procedure of a widget.

Using Enhanced X-Windows to Query Event Sources

The event manager provides three subroutines to examine and read events, including file and timer events, that are in the queue. These subroutines handle the Intrinsics library equivalents of the **XPending**, **XPeekEvent**, and the **XNextEvent** subroutines.

The **XtAppPending** subroutine determines if there are events on the input queue for a given application.

The **XtAppPeekEvent** subroutine returns the value from the beginning of the input queue of a given application without removing input from the queue.

The **XtAppNextEvent** subroutine returns the value from the beginning of the input queue of a given application.

Using Enhanced X-Windows in Dispatching Events

The Intrinsics library provides subroutines that dispatch events to widgets or other application code. Every client interested in X events on a widget uses the **XtAddEventHandler** subroutine to register which events it is interested in and a procedure (event handler) to be called when the event happens in that window. The translation manager automatically registers event handlers for widgets that use translation tables.

The **XtAppProcessEvent** subroutine gives applications direct control of the processing of different types of input. This procedure is not usually called by client applications. The **XtAppProcessEvent** subroutine processes timer events by calling appropriate timer callbacks, alternate input by calling appropriate alternate input callbacks, and X events by calling the **XtDispatchEvent** subroutine.

Using Enhanced X-Windows to Handle the Application Input Loop

The **XtAppMainLoop** subroutine handles the processing of input from a given application. This procedure controls the main loop of X Toolkit applications and does not return. This main loop is basically an infinite loop that first calls the **XtAppNextEvent** subroutine, and then calls the **XtDispatchEvent** subroutine. Applications exit in response to some user action.

Applications can provide their own version of this loop by testing a global end flag or confirming that the number of top-level widgets is larger than 0 before circling back to the call to the **XtAppNextEvent** subroutine.

Using Enhanced X-Windows to Set and Check the Sensitivity State of a Widget

Many widgets have a mode in which they assume a different appearance (for example, are grayed out or stippled), do not respond to user events, and become dormant. When dormant, a widget is considered to be insensitive. When a widget is insensitive, the event manager does not dispatch events to the widget with a **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **MotionNotify**, **EnterNotify**, **LeaveNotify**, **FocusIn**, or **FocusOut** event type.

A widget can be insensitive because its *sensitive* field is set to the value of **False** or because one of its parents is insensitive, causing the widget's *ancestor_sensitive* field to be set to the value of **False**. A widget can but does not need to distinguish these two cases visually.

The **XtSetSensitive** subroutine sets the sensitivity state of a widget.

The **XtIsSensitive** subroutine checks the current sensitivity state of a given widget. This procedure is usually done by the parent widgets.

Using Enhanced X-Windows to Add Background Work Procedures

The Intrinsics library have limited support for background processing. Because most applications spend most of their time waiting for input, you can register an idle-time work procedure to be called when the Toolkit would otherwise block the **XtAppNextEvent** subroutine or the **XtAppProcessEvent** subroutine. Work procedure pointers are of the **XtWorkProc** type.

The **XtAppAddWorkProc** subroutine registers a work procedure for a given application. Multiple work procedures can be registered.

To remove a work procedure, either return the value of **True** from the procedure when it is called or use the **XtRemoveWorkProc** subroutine.

Using Enhanced X-Windows to Handle Widget Exposure and Visibility

Every primitive widget and some composite widgets display data on the screen by means of raw the **Xlib** library calls. Widgets must keep track of what they write to the screen. They must keep enough state to redisplay the window or parts of it if a portion is obscured and then redisplayed.

Using Enhanced X-Windows to Redisplay a Widget

The redisplay of a widget upon exposure is the responsibility of the *expose* procedure in the class record of the widget. The *expose* procedure pointer in a widget class is of the **XtExposeProc** type.

If a widget has no display semantics, it can specify the value of **NULL** for the *expose* field. If the *expose* procedure is the value of **NULL**, the **XtRealizeWidget** subroutine fills in a default bit gravity of **NorthWestGravity** before it calls the *realize* procedure of the widget.

Many composite widgets serve only as containers for their child widgets and have no defined *expose* procedure.

It often is possible to anticipate the display needs of several levels of subclassing. For example, rather than writing separate display procedures for the widgets `Label`, `Command`, and `Toggle`, you can write a single display routine in `Label` that uses display state fields like the following:

Boolean ***invert***
Boolean ***highlight***
Dimension ***highlight_width***

`Label` would have *invert* and *highlight* always the value of **False** and *highlight_width* field the value of 0. `Command` would dynamically set the *highlight* and *highlight_width* fields, but it would leave the *invert* field always the value of **False**. `Toggle` would dynamically set all three. In this case, the *expose* procedures for `Command` and `Toggle` inherit the *expose* procedure of their superclass.

Using Enhanced X-Windows Widget Visibility

Some widgets use substantial computing resources to display data. However, this resource is wasted if the widget is not actually visible on the screen, obscured by another application, or iconified.

The *visible* field in the **Core** widget structure tells the widget that it need not display data. This field has the value of **True** by the time an **Expose** event is processed if the widget is visible, but it usually has the value of **False** if the widget is not visible.

Widgets can use or ignore the *visible* field. They ignore the field if the *visible_interest* field in their widget class record is set to the value of **False**. In such cases, the *visible* field is initialized as the value of **True** and does not change. If the *visible_interest* field has the value of **True**, the event manager asks for the **VisibilityNotify** events for the widget and updates the *visible* field accordingly.

Using Enhanced X-Windows X Event Handlers

Event handlers are procedures called when specified events occur in a widget. Most widgets do not use event handlers explicitly. Instead, they use the Intrinsics library translation manager.

Event handler procedure pointers are of the **XtEventHandler** type.

Using Enhanced X-Windows Event Handlers That Select Events

The **XtAddEventHandler** subroutine registers an event handler procedure with the dispatch mechanism. This procedure is then called when an event matching the defined mask occurs on the specified widget.

The **XtRemoveEventHandler** subroutine removes a previously registered event handler.

Using Enhanced X-Windows Event Handlers That Do Not Select Events

The **XtAddRawEventHandler** subroutine enables clients to register an event handler procedure with the dispatch mechanism without causing the server to select for that event. It functions like the **XtAddEventHandler** subroutine except that it does not affect the event mask of the widget and does not call an **XSelectInput** subroutine for its events.

The **XtRemoveRawEventHandler** subroutine removes a previously registered raw event handler.

Using Enhanced X-Windows to Retrieve a Current Event Mask

The **XtBuildEventMask** subroutine retrieves the event mask for a given widget.

Related Information

The **XtWorkProc** data structure.

The **XtAcceptFocusProc** data type, **XtEventHandler** data type, **XtExposeProc** data type, **XtInputCallbackProc** data type.

The **XtIsSensitive** macro.

The **XtTimerCallbackProc** procedure.

The **XNextEvent** subroutine, **XPeekEvent** subroutine, **XPending** subroutine, **XSelectInput** subroutine, **XSetInputFocus** subroutine, **XtAddEventHandler** subroutine, **XtAddGrab** subroutine, **XtAddRawEventHandler** subroutine, **XtAppAddInput** subroutine, **XtAppAddTimeOut** subroutine, **XtAppAddWorkProc** subroutine, **XtAppMainLoop** subroutine, **XtAppNextEvent** subroutine, **XtAppPeekEvent** subroutine, **XtAppPending** subroutine, **XtAppProcessEvent** subroutine, **XtBuildEventMask** subroutine, **XtCallAcceptFocus** subroutine, **XtDispatchEvent** subroutine, **XtPopup** subroutine, **XtRealizeWidget** subroutine, **XtRemoveEventHandler** subroutine, **XtRemoveGrab** subroutine, **XtRemoveInput** subroutine, **XtRemoveRawEventHandler** subroutine, **XtRemoveTimeOut** subroutine, **XtRemoveWorkProc** subroutine, **XtSetKeyboardFocus** subroutine, **XtSetSensitive** subroutine.

Enhanced X-Windows Resource Management Overview

Widget writers need a large set of resources at the time of widget creation. Some resources come from the resource database, some from the argument list supplied in the call to the **XtCreateWidget** subroutine, and some from the internal defaults specified for the widget. Resources are obtained first from the argument list, then from the resource database for all resources not specified in the argument list, and last from the internal default.
A resource is a field in the widget record with a corresponding resource entry in the widget resource list or in the resource list of any of its superclasses. This field can be set by one of the following methods:

- The **XtCreateWidget** subroutine, by naming the field in the argument list.
- An entry in the default resource files, using either the name or class.
- Using the **XtSetValues** subroutine.

In addition, this field is readable by the **XtGetValues** subroutine.

Not all fields in a widget record are resources. Some fields provide bookkeeping for the generic routines, such as the *managed* and *being_destroyed* fields. Other fields provide local bookkeeping while still others are derivations of resources, such as GCs and pixmaps.

Using Enhanced X-Windows to Create Resource Lists

A resource entry specifies a field in the widget, the textual name and class of the field that argument lists and external resource files use to refer to the field, and a default value that the field should get if no value is specified.

The **XtGetResourceList** subroutine gets the resource list structure for a particular class. The **XtSetValues** and **XtGetValues** subroutines use the resource list to set and get the widget state.

The following is an example of an abbreviated version of the resource list in the Label widget:

```
/*Resources specific to Label*/
static XtResource resources[ ] = {
{XtNforeground, XtCForeground, XtRPixel,
   sizeof(Pixel), XtOffset(LabelWidget,
   label.foreground), XtRString,
   XtDefaultForeground},
{XtNfont, XtCFont, XtRFontStruct, sizeof
   (XFontStruct *), XtOffset(LabelWidget,
   label.font), XtRString, XtDefaultFont},
{XtNlabel, XtCLabel, XtRString, sizeof(String),
   XtOffset(LabelWidget, label.label), XtRString, NULL},
                                  .
                                  .
                                  .
}
```

The complete resource name for a field of a widget instance is the concatenation of the application name (from the **XtAppCreateShell** subroutine), the instance names of all the parents of the widget up to the **ApplicationShellWidget**, the instance name of the widget itself, and the resource name of the specified field of the widget.

The full resource class of a field of a widget instance is the concatenation of the application class (from the **XtAppCreateShell** subroutine), the widget class names of all the parents of the widget up to the **ApplicationShellWidget** (not the superclasses), the widget class name of the widget itself, and the resource name of the specified field of the widget.

Using Enhanced X-Windows to Chain Resource Lists from Superclass to Subclass

The **XtCreateWidget** subroutine gets resources as a superclass-to-subclass operation. The resources specified in the the Core resource list are called, then those in the subclass are called, and so on down to the resources specified for this widget class. Within a class, resources are called in the order they are declared.

Generally, if a widget resource field is declared in a superclass, that field is included in the resource list of the superclass and does not need to be included in the resource list of the subclass. For example, since the Core class contains a resource entry for the *background_pixel* field, the implementation of the Label widget does not need a resource entry for the *background_pixel* field as well. However, a subclass can override the resource entry for any field declared in a superclass by specifying a resource entry for that field in its own resource list. To provide new values that supersede the superclass defaults, override the resource entry. At class initialization time, resource lists for that class are scanned from the superclass down to the class to look for resources with the same offset. A matching resource in a subclass will be reordered to override the superclass entry. A copy of the superclass resource list is made to avoid affecting other subclasses of the superclass.

A widget does not do anything to get its own resources; the **XtCreateWidget** subroutine automatically gets resources for widgets before calling the class initialize procedure.

Some widgets have subparts that are not widgets but for which the widget needs to get resources. For example, the *Text* widget gets resources for its *source* and *sink*. Widgets with such needs call the **XtGetSubresources** subroutine.

Some widgets also need resources that are not specific to a widget but that apply to the overall application. In this case, widgets call the **XtGetApplicationResources** subroutine.

Related Information

The **XtConverter** data type, **XtResource** data structure.

The **XtArgsFunc** data type, **XtResourceDefaultProc** data type.

The **XtAppAddConverter** subroutine, **XtAppCreateShell** subroutine, **XtConvert** subroutine, **XtCreateWidget** subroutine, **XtDirectConvert** subroutine, **XtGetApplicationResources** subroutine, **XtGetResourceList** subroutine, **XtGetSubresources** subroutine, **XtGetSubvalues** subroutine, **XtGetValues** subroutine, **XtSetValues** subroutine.

Enhanced X-Windows Predefined Resource Converters Overview

Using Enhanced X-Windows to Convert Resources

The Intrinsics library provide a mechanism for registering representation converters that are automatically called by the resource calling subroutines. In addition, the Intrinsics library provide and register several commonly used converters. The resource conversion mechanism serves several purposes:

- Permits user and application resource files to contain ASCII representations of non-textual values.
- Allows textual or other representations of default resource values that are dependent upon the display, screen, or color map, and thus must be computed at run time.
- Caches all conversion source and result data. Conversions that require much computation or space (for example, string to translation table) or that require round trips to the server (for example, string to font or color) are performed only once.

The Intrinsics library defines all the representations used in the **Core**, **Composite**, **Constraint**, and **Shell** widgets. Furthermore, it registers the following resource converters:

From	To
XtRString	**XtRAcceleratorTable, XtRBoolean, XtRBool, XtRCursor, XtRDimension, XtRDisplay, XtRFile, XtRFloat, XtRFont, XtRFontStruct, XtRInt, XtRPixel, XtRPosition, XtRShort, XtRTranslationTable,** and **XtRUnsignedChar**
XtRColor	**XtRPixel**
XtRInt	**XtRBoolean, XtRBool, XtRColor, XtRDimension, XtRFloat, XtRFont, XtRPixel, XtRPixmap, XtRPosition, XtRShort,** and **XtRUnsignedChar**
XtRPixel	**XtRColor**

The string to pixel conversion has two predefined constants that work in contrast with each other (**XtDefaultForeground** and **XtDefaultBackground**). These constants evaluate the black and white pixel values of the widget's screen, respectively. If, however, the application uses reverse video, they evaluate the white and black pixel values of the widget's screen, respectively. The string-to-font and font structure converters recognize the **XtDefaultFont** constant and evaulate this to the font in the default graphics context of the screen.

Using Enhanced X-Windows to Write a New Resource Converter

Type converters use pointers to the **XrmValue** data structures (defined in the **<X11/Xresource.h>** header file) for input and output values.

A resource converter procedure pointer is of the **XtConverter** type.

Type converters should perform the following actions:

- Check that the number of arguments passed is correct.
- Attempt the type conversion.
- Return a pointer to the data in the *to* field if successful. Otherwise, call the **XtWarningMsg** subroutine and return without modifying the *to* field.

Most type converters take the data described by the specified *from* field and return data by writing into the specified *to* field. Some converters need other information, available in the specified *args* field. A type converter can invoke another type converter, allowing differing sources that may convert into a common intermediate result to make maximum use of the type converter cache.

The address written in **to->addr** cannot be a local variable of the converter because this is not valid after the converter returns. The address should be a pointer to a static variable, as shown in the following example where *screenColor* is returned.

The following is an example of a converter that takes a string and converts it to a pixel:

```
static void CvtStringToPixel(args, num_args,
                             fromVal, toVal)
   XrmValue *args;
   Cardinal *num_args;
   XrmValue *fromVal;
   XrmValue *toVal;
{
   static XColor screenColor;
   XColor exactColor;
   Screen *screen;
   Colormap colormap;
   Status status;
   char message[1000];
   XrmQuark q;
   String params[1];
   Cardinal num_params = 1;

   if (*num_args != 2)
      XtErrorMsg("cvtStringToPixel","wrongParameters",
         "XtToolkitError", "String to pixel
          conversion needs screen and colormap
          arguments",(String *)NULL,(Cardinal *) NULL);

   screen = *((Screen **) args[0].addr);
   colormap = *((Colormap *) args[1].addr);

   LowerCase((char *) fromVal->addr,message);
   q = XrmStringToQuark(message);

   if(q == XtQExtdefaultbackground){done(&screen->
      white_pixel, Pixel);return; }
   if(q == XtQExtdefaultforeground){done(&screen->
      black_pixel, Pixel);return; }

   if ((char) fromVal->addr&lrbk.0] == '#') { /* a
      color definition */

      status = XParseColor(DisplayOfScreen(screen),colormap,
         (String)fromVal->addr, &screenColor);
      if(status != 0) status = XAllocColor
         (DisplayOfScreen(screen), colormap, &screenColor);

   } else /* some color name */

      status = XAllocNamedColor(DisplayOfScreen(screen),
         colormap, (String)fromVal->addr, &screenColor,
         &exactColor);
   if(status == 0) {

      params[0]=(String)fromVal->addr;
      XtWarningMsg("cvtStringToPixel","noColormap",
            "XtToolkitError",
         "Cannot allocate colormap entry for\"%s\"",
            params,&num_params);

   } else {

      toVal->addr = (caddr_t)&screenColor.pixel;
      toVal->size = sizeof(Pixel);

   }
};
```

All type converters should define some set of conversion values that they will succeed on so these can be used in the resource defaults. This issue arises only with conversions, such as fonts and colors, where there is no string representation that all server implementations will necessarily recognize. For resources of this nature, the converter should define a symbolic constant, such as **XtDefaultForeground**, **XtDefaultBackground**, or **XtDefaultFont**.

The code for registering the **CvtStringToPixel** routine is the following:

```
static XtConvertArgRec colorConvertArgs[] = {
   {XtBaseOffset, (caddr_t) XtOffset(Widget,
      core.screen), sizeof(Screen*)}
   {XtBaseOffset, (caddr_t) XtOffset(Widget,
      core.colormap), sizeof(Colormap)}
};

XtAddConverter(XtRString, XtRPixel, CvtStringToPixel,
   colorConvertArgs, XtNumber(colorConvertArgs));
```

The conversion argument descriptors *colorConvertArgs* and *screenConvertArg* are predefined.

- The *screenConvertArg* descriptor puts the screen field for the widget into the *args[0]* field.
- The *colorConvertArgs* descriptor puts the screen field of the widget into the *args[0]* field, and the widget *colormap* field into the *args[1]* field.

Conversion routines should not just put a descriptor for the address of the base of the widget into the *args[0]* field, and use that in the routine. They should pass the actual values that the conversion depends on. By keeping the dependencies of the conversion procedure specific, it is more likely that subsequent conversions will find what they need in the conversion cache. This way the cache is smaller and has fewer and more widely applicable entries.

Using Enhanced X-Windows to Register a New Resource Converter

To register a new resource converter, use the **XtAppAddConverter** subroutine.

For the few type converters that need additional arguments, the Intrinsics conversion mechanism provides a method of specifying how these arguments should be computed. The **XtAddressMode** enumerated type and the **XtConvertArgRec** data structure specify how each argument is derived. Both the structure and the type are defined in the **<X11/Convert.h>** header file.

Using Enhanced X-Windows to Invoke Resource Converters

Resource-fetchinging subroutines call resource converters if the user specifies a resource that is a different representation from the desired representation or if the widget's default resource value representation is different from the desired representation. Examples of resource-calling subroutines are **XtGetSubresources** and **XtGetApplicationResources**.

To invoke conversions, use the **XtConvert** or **XtDirectConvert** subroutines.

Using Enhanced X-Windows to Read and Write Widget State

Any resource field in a widget can be read or written by a client. When writing, the widget decides what changes it will actually allow, and updates all derived fields appropriately.

The following list describes the subroutines to use for various widget state operations.

XtGetValues	Retrieves the current value of a resource associated with a widget instance.
XtGetSubvalues	Retrieves the current value of non-widget resource data associated with a widget instance.
XtSetValues	Modifies the current value of a resource associated with a widget instance.
XtSetSubvalues	Sets the current value of a non-widget resource associated with a widget instance.

Using Enhanced X-Windows to Format Resource Files

A resource file contains text representing the default resource values for an application or set of applications. The resource file is an ASCII text file that consists of a number of lines with the following EBNF syntax:

```
Xdefault        = {line "\n"}.

line            = (comment | production).

comment         = "!" string.

production      = resourcename ":" string.

resourcename    = ["*"] name {("." | "*") name}.

string          = {<any character not including eol>}.

name            = {"A"-"Z" | "a"-"z" | "0"-"9"}.
```

If the last character on a line is a \ (backslash), that line is assumed to continue on the next line. To include a new-line character in a string, use "–n."

Related Information

The **XrmValue** data structure, **XtConvertArgRec** data structure.

The **XtAlmostProc** data type, **XtArgsFunc** data type, **XtConverter** data type, **XtSetValuesFunc** data type.

The **XtAddressMode** enumerated type.

The **XtAppAddConverter** subroutine, **XtConvert** subroutine, **XtDirectConvert** subroutine, **XtGetApplicationResources** subroutine, **XtGetSubresources** subroutine, **XtGetSubvalues** subroutine, **XtGetValues** subroutine, **XtSetSubvalues** subroutine, **XtSetValues** subroutine, **XtWarningMsg** subroutine.

Enhanced X-Windows Translation Management Overview

Except under unusual circumstances, widgets do not hardwire the mapping of user events into widget behavior by using the event manager. Instead, they provide a default that can be overridden by the user.

The translation manager provides an interface to specify and manage the mapping of Enhanced X-Windows event sequences into widget-supplied functionality; for example, calling the *Abc* procedure when the **Y** key is pressed.

The translation manager uses two kinds of tables to perform translations:

Action table	This is in the widget class structure, and specifies the mapping of externally available procedure name strings to the corresponding procedure implemented by the widget class.
Translation table	This is also in the widget class structure, and specifies the mapping of event sequence to procedure name strings.

You can override the translation table in the class structure for a specific widget instance by supplying a different translation table for the widget instance. The resource name is **XtNtranslations.**

Using Enhanced X-Windows Action Tables

All widget class records contain an action table. In addition, an application can register its own action tables with the translation manager, so that the translation tables it provides to widget instances can access application functionality.

The action table of the application uses the **XtActionsRec** data structure.

For example, the Command widget has procedures to:

- Set the command button to indicate it is activated
- Reset the command button to its normal mode
- Highlight the button borders
- Unhighlight the button borders
- Notify any callbacks that the button has been activated.

The action table for the Command widget class makes these functions available to translation tables written for Command or any subclass. The string entry is the name used in translation tables. The procedure entry, which is usually spelled the same as the string, is the name of the C language procedure that implements that function:

```
XtActionsRec actionTable[] = {
   {"Set",           Set},
   {"Unset",         Unset},
   {"Highlight",     Highlight},
   {"Unhighlight",   Unhighlight},
   {"Notify",        Notify}
};
```

To declare an action table and register it with the translation manager, use the **XtAppAddActions** subroutine.

Using Enhanced X-Windows to Translate Action Names to Procedures

The translation manager uses a simple algorithm to convert the name of a procedure specified in a translation table to the actual procedure specified in an action table. When the widget is realized, it performs a search for the name in the following tables:

- The widget's class action table for the name
- The widget's superclass action table and on up the superclass chain
- The action tables registered with the **XtAddActions** subroutine, from the most recently added table to the oldest table.

The translation manager stops the search as soon as it finds a name. If no name is found, it generates an error.

Using Enhanced X-Windows Translation Tables

All widget instance records contain a translation table. A translation table specifies the action procedures called for an event or a sequence of events. The table is a string containing a list of translations from an event or an event sequence to one or more action procedure calls. The translations are separated from one another by new-line characters (ASCII LF). The translation table has no default value.

For example, the default behavior of Command is the following:

- Highlight on enter window
- Unhighlight on exit window
- Invert on left button down
- Call callbacks and reinvert on left button up.

The default translation table for Command is the following:

```
static String defaultTranslations =
   "<EnterWindow>:     Highlight()\n\
   <LeaveWindow>:      Unhighlight()\n\
   <Btn1Down>:         Set()\n\
   <Btn1Up>:           Notify() Unset()";
```

The *tm_table* field of the **CoreClass** record should be filled in at the time of static initialization with the string containing the default translations of the class. If a class must inherit the translations of its superclass, it can store the special value **XtInheritTranslations** into the *tm_table* field. After the class initialization procedures have been called, the Intrinsics library compile this translation table into an efficient internal form. Then, at widget creation time, this default table is used for any widgets that have not had their Core translations field set by the resource manager or the initialize procedures.

The resource conversion mechanism automatically compiles string translation tables that are resources. If a client uses translation tables that are not resources, it must compile them using the **XtParseTranslationTable** subroutine.

The Intrinsics use the compiled form of the translation table to register the necessary events with the event manager. Widgets need to specify only the action and translation tables for events to be processed by the translation manager.

Using Enhanced X-Windows Event Sequences

An event sequence is a comma-separated list of the Enhanced X-Windows event descriptions that describes a specific order of X events to map to a set of program actions. Each of these event descriptions consists of the followiing three parts:

- The X event type
- A prefix consisting of the modifier bits
- An event-specific suffix.

Various abbreviations can be used to make translation tables easier to read.

Using Enhanced X-Windows Action Sequences

Action sequences specify which program or widget actions to take in response to incoming events. An action sequence contains a series of action procedure call specifications, each of which gives the name of an action procedure and a list of string parameters, in parentheses, to pass to that procedure.

Using Enhanced X-Windows to Merge Translation Tables

Sometimes an application needs to add its own translations in place of or in addition to the translation of the widget. For example, a window manager provides subroutines to move a window. Normally, this window manager can move the window when any pointer button is pressed down in a title bar. It allows the user to specify other translations for the middle or right button down in the title bar, but it ignores any user translations for left button down.

To accomplish this, the window manager should first create the title bar and then merge the two translation tables into the title bar's translations. One translation table contains the translations that the window manager wants if the user has not specified a translation for a particular event or event sequence. The other translation table contains the translations that the window manager wants regardless of what the user has specified.

Three subroutines support this merging:

XtParseTranslationTable	Compiles a translation table.
XtAugmentTranslations	Nondestructively merges a compiled translation table into the existing widget translations.
XtOverrideTranslations	Destructively merges a compiled translation table into the existing widget translations.

Using Enhanced X-Windows to Change Translation Tables

To replace a widget's translations completely, use the **XtSetValues** subroutine on the **XtNtranslations** resource and specify a compiled translation table as the value.

To make it possible to easily modify translation tables in resource files, the string-to-translation-table resource type converter allows specification of whether the table replaces, augments, or overrides any existing translation table in the widget. To do this, the first character in the table should be a # (number sign), followed by `replace` (the default action), `augment`, or `override`, to indicate whether to replace, augment, or override any existing table.

To completely remove existing translations, use the **XtUninstallTranslations** subroutine.

Using Enhanced X-Windows Accelerators

Applications often need to be able to bind events in one widget to actions in another, such as invoking menu actions from the keyboard. To accomplish this, the Intrinsics library provide accelerators. An accelerator is a translation table that is bound with its actions in the context of a particular widget. The accelerator table can then be installed on a destination widget, which allows an action in the destination widget to execute an accelerator action as if it were triggered in the accelerator widget.

Each widget instance contains that widget's exported accelerator table. Each class of widget exports a method that takes a displayable string representation of the accelerators so that the widgets can display their current accelerators. The representation is the accelerator table in canonical representation form.

Use these routines to perform the following accelerator function:

XtParseAcceleratorTable	Parses an accelerator table.
XtInstallAccelerators	Installs accelerators from one widget onto another widget.
XtInstallAllAccelerators	Installs all accelerators from one widget and all of its descendants onto one destination.

Using Enhanced X-Windows to Convert Key Codes to KeySyms

The **XtKeyProc** data type specifies the interface definition for user defined **KeyCode**-to-**KeySym** translator procedures. A **KeyCode**-to-**KeySym** translator procedure takes a key code and modifiers and produces a key symbol. For any specified key translator function, the *ModifiersReturn* parameter will be a constant that indicates the subset of all modifiers examined by the key translator.

The translation manager provides support for automatically translating key codes in incoming key events to KeySyms.

The following key code translation operations are possible:

XtSetKeyTranslator	Registers a key translator.
XtRegisterCaseConverter	Registers a case converter. Pointers to case conversion procedures are of the **XtCaseProc** data type.
XtConvertCase	Determines the uppercase and lowercase equivalents for a KeySym.
XtTranslateKeycode	Invokes the currently registered Keycode-to-KeySym translator.

Using Enhanced X-Windows Translation Table File Syntax

A translation table file is an ASCII text file. The proper notation and syntax as well as common modifier names, event type values, supported abbreviations, canonical representations, and examples are found in the sections that follow.

Using Enhanced X-Windows Notation

Syntax is specified in EBNF notation with the following conventions:

[a]	Means either nothing or "a"
{ a }	Means 0 or more occurrences of "a"

All terminals are enclosed in " " (double quotes). Informal descriptions are enclosed in <> (angle brackets).

Using Enhanced X-Windows Syntax

The syntax of the translation table file is:

translationTable	= [directive] { production }
directive	= ("#replace" | "#override" | "#augment") "\n"
production	= lhs ":" rhs "\n"
lhs	= (event | keyseq) { "," (event | keyseq) }

keyseq	= “"” keychar {keychar} “"”
keychar	= [“^” | “$” “\”] <ISO Latin 1 character>
event	= [modifier_list] “<”event_type“>” [“(“count[“+”]”)”] {detail}
modifier_list	= ([“!” | “:”] {modifier})| “None”
modifier	= [“~”] modifier_name
count	= (“1” | “2” | “3” | “4” | ...)
modifier_name	= “@” <keysym> | <see the ModifierNames table that follows>
event_type	= <see the Event Types table that follows>
detail	= <event specific details>
rhs	= {name “(“ [params] ”)” }
name	= namechar { namechar }
namechar	= { “a”-“z” | “A”-“Z” | “0”-“9” | “$” | “_” }
params	= string {“,” string}.
string	= quoted_string | unquoted_string
quoted_string	= “"” {<Latin1 character>} “"”
unquoted_string	= {<Latin1 character except space, tab, “,”, newline, “)”>}

It is often convenient to include newlines in a translation table to make it more readable. In C language, indicate a newline with a “\n”:

```
"<Btn1Down>: DoSomething()\n\
<Btn2Down>: DoSomethingElse()"
```

Using Enhanced X-Windows Modifier Names

The *modifier* field is used to specify normal Enhanced X-Windows keyboard and button modifier mask bits. Modifiers are allowed on the **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **MotionNotify**, **EnterNotify**, **LeaveNotify** event types and their abbreviations:

An error occurs when a translation table that contains modifiers for any other events is parsed.

- Modifiers don't care if the *modifier_list* field has no entries and is not the value of **None**.
- Listed modifiers must be in the correct state and “don't care” about any other modifers if any modifiers are specified and an ! (exclamation point) is not specified.
- Listed modifers must be in the correct state and no other modifers can be asserted if an ! (exclamation point) is specified at the beginning of the modifier list.
- Modifiers preceded by a ~ (tilde) must not be asserted.
- Modifiers cannot be asserted if the value of **None** is specified.
- Standard modifiers in the event, which map the event key code into a keysym, are applied by the Intrinsics when a : (colon) is specified at the beginning of the modifier list.

  The default standard modifiers are the **Shift** and **Lock** keys. The resulting keysym must exactly match the specified keysym, and the nonstandard modifiers in the event must match the **modifier_list**. For example, ":<Key>a" is different from "<Key>A" and ":Shift<Key>A" is different from ":<Key>A".

- No standard modifiers are applied when a : (colon) is not specified. This makes "<Key>A" and "<Key>a", for example, equivalent.

In key sequences, a ^ (circumflex) is an abbreviation for the Control modifier; a $ (dollar sign) is an abbreviation for Meta; and a \ (backslash) is used to quote any character, in particular a " (double quote), a ~ (circumflex), a $ (dollar sign), or another \ (backslash). In brief:

No Modifiers:	None <event> detail
Any Modifiers:	<event> detail
Only these Modifiers:	! mod1 mod2 <event> detail
These modifiers and any others:	mod1 mod2 <event> detail

Using **None** for a **modifier_list** is the same as using an exclamation point with no modifiers.

Modifier	**Abbreviation**	**Meaning**
Ctrl	**c**	**Control** modifier bit
Shift	**s**	**Shift** modifier bit
Lock	**l**	**Lock** modifier bit
Meta	**m**	**Meta** key modifier*
Hyper	**h**	**Hyper** key modifier*
Super	**su**	**Super** key modifier*
Alt	**a**	**Alt** key modifier*
Mod1		**Mod1** modifier bit
Mod2		**Mod2** modifier bit
Mod3		**Mod3** modifier bit
Mod4		**Mod4** modifier bit
Mod5		**Mod5** modifier bit
Button1		**Button1** modifier bit
Button2		**Button2** modifier bit
Button3		**Button3** modifier bit
Button4		**Button4** modifier bit
Button5		**Button5** modifier bit
ANY		Any combination

*A key modifier is any modifier bit whose corresponding key code contains the corresponding left or right keysym.

For example, `m` or `Meta` means any modifier bit that maps to a key code whose keysym list contains `XK_Meta_L` or `XK_Meta_R`. This interpretation applies for each display, not globally or for each application context. The **Control**, **Shift**, and **Lock** modifier names refer explicitly to the corresponding modifier bits, and there is no additional interpretation of keysyms for these modifiers.

Because arbitrary keysyms can be associated with modifiers, the set of modifier key modifiers is extensible. The "@" <keysym> syntax means any modifier bit whose corresponding key code contains the specified keysym.

A modifier_list/keysym combination in a translation is matched with a modifiers/keycode combination in an event as follows:

If a : (colon) is used, the Intrinsics library call the registered **XtKeyProc** subroutine (the default subroutine is the **XtTranslateKey** subroutine) for the display with the keycode and modifiers. For a match to occur, (*modifiers* and ~*modifiers_return*) must equal *modifier_list* and *keysym_return* must equal the given keysym.

If a colon is not used, the Intrinsics masks off all don't care bits from the modifiers. The value must be equal to *modifier_list*. Then, for each possible combination of don't care modifiers in the *modifier_list* field, the Intrinsics call the registered **XtKeyProc** subroutine for that display with the keycode and that combination OR'd with the cared-about modifier bits from the event. The *keysym_return* field must match the keysym in the translation.

Using Enhanced X-Windows Event Types

The EventType field describes the **XEvent** types. The currently defined EventType values are the following:

Event Type	Value
Key	
KeyDown	KeyPress
Key Up	KeyRelease
BtnDown	ButtonPress
BtnUp	ButtonRelease
Motion	MotionNotify
PtrMoved	
MouseMoved	
Enter	EnterNotify
EnterWindow	
Leave	LeaveNotify
LeaveWindow	
FocusIn	FocusIn
FocusOut	FocusOut
Keymap	KeymapNotify
Expose	Expose
GrExp	GraphicsExpose
NoExp	NoExpose
Visible	VisibilityNotify
Create	CreateNotify
Destroy	DestroyNotify
Unmap	UnmapNotify
Map	MapNotify
MapReq	MapRequest
Reparent	ReparentNotify
Configure	ConfigureNotify
ConfigureReq	ConfigureRequest
Grav	GravityNotify
ResReq	ResizeRequest
Circ	CirculateNotify
CircReq	CirculateRequest
Prop	PropertyNotify

SelClr	SelectionClear
SelReq	SelectionRequest
Select	SelectionNotify
Clrmap	ColormapNotify
Message	ClientMessage
Mapping	MappingNotify

Using Enhanced X-Windows Supported Event Type Abbreviations

The *detail* field is event-specific and corresponds normally to the *detail* field of an **XEvent,** for example, <key>A. If no *detail* field is specified, then **ANY** is assumed. The supported abbreviations are the following:

Abbreviation	**Meaning**
Ctrl	**KeyPress** with control modifier
Meta	**KeyPress** with meta modifier
Shift	**KeyPress** with shift modifier
Btn1Down	**ButtonPress** with **Btn1** detail
Btn1Up	**ButtonRelease** with **Btn1** detail
Btn2Down	**ButtonPress** with **Btn2** detail
Btn2Up	**ButtonRelease** with **Btn2** detail
Btn3Down	**ButtonPress** with **Btn3** detail
Btn3Up	**ButtonRelease** with **Btn3** detail
Btn4Down	**ButtonPress** with **Btn4** detail
Btn4Up	**ButtonRelease** with **Btn4** detail
Btn5Down	**ButtonPress** with **Btn5** detail
Btn5Up	**ButtonRelease** with **Btn5** detail
BtnMotion	**MotionNotify** with any button modifier
Btn1Motion	**MotionNotify** with **Button1** modifier
Btn2Motion	**MotionNotify** with **Button2** modifier
Btn3Motion	**MotionNotify** with **Button3** modifier
Btn4Motion	**MotionNotify** with **Button4** modifier
Btn5Motion	**MotionNotify** with **Button5** modifier

A keysym can be specified as any of the standard keysym names: as a hexadecimal number prefixed with **0x** or **0X**, as an octal number prefixed with **0**, or as a decimal number. A keysym expressed as a single digit is interpreted as the corresponding Latin1 keysym. For example, `0` is the keysym `XK_0`. Other single character keysyms are treated as literal constants from Latin1. For example, `!` is treated as `0x21`. Standard keysym names are those defined in the **<X11/keysymdef.h>** header file, with the **XK_** prefix removed.

Using Enhanced X-Windows Canonical Representation

Every translation table has a unique, canonical text representation. This representation is passed to a widget's *display_accelerator* method to describe the accelerators installed on that widget. (The syntax of a translation table file is described in Using Enhanced X-Windows Syntax. The canonical representation of a translation table is the following:

translationTable	= { production }
production	= lhs “:” rhs “\n”
lhs	= event { “,” event }
event	= [modifier_list] “<”event_type“>” [“(”count[“+”]’)”] {detail}
modifier_list	= [“!” | “:”] {modifier})
modifier	= [“~”] modifier_name
count	= (“1” | “2” | “3” | “4” | ...)
modifier_name	= “@” <keysym> | The canonical modifier names are:

Ctrl	**Button1**	**Mod1**
Lock	**Button2**	**Mod2**
Shift	**Button3**	**Mod3**
	Button4	**Mod4**
	Button5	**Mod5**

event_type	= The canonical event types are the following:

KeyPress	**KeyRelease**
ButtonPress	**ButtonRelease**
MotionNotify	**LeaveNotify**
EnterNotify	**KeymapNotify**
FocusIn	**FocusOut**
Expose	**GraphicsExpose**
NoExpose	**VisibilityNotify**
CreateNotify	**DestroyNotify**
UnmapNotify	**MapNotify**
MapRequest	**ReparentNotify**
ConfigureNotify	**ConfigureRequest**
GravityNotify	**ResizeRequest**
CirculateNotify	**CirculateRequest**
PropertyNotify	**SelectionClear**
SelectionRequest	**SelectionNotify**
ColormapNotify	**ClientMessage**

detail	= <event specific details>
rhs	= {name “(” [params] “)” }
name	= namechar { namechar }
namechar	= { “a”-“z” | “A”-“Z” | “0”-“9” | “$” | “_” }
params	= string {“,” string}.
string	= quoted_string
quoted_string	= “"“ {<Latin1 character>} ”"”

Using Enhanced X-Windows Examples of Event Types

- Put more specific events in the table before more general events:

```
Shift <Btn1Down> : twas()\n\
<Btn1Down> : brillig()
```

- For double-click on **Button 1 Up** with **Shift**, use:

```
Shift<Btn1Up>(2) : and()
```

This is equivalent to:

```
Shift<Btn1Down>,Shift<Btn1Up>,Shift<Btn1Down>,
Shift<Btn1Up> : and()
```

with appropriate timers set between events.

- For double-click on **Button 1 Down** with **Shift** use:

```
Shift<Btn1Down>(2) : the()
```

It is equivalent to:

```
Shift<Btn1Down>,Shift<Btn1Up>,Shift<Btn1Down> : the()
```

with appropriate timers set between events.

- Mouse motion is always discarded when it occurs between events in a table where no motion event is specified:

```
<Btn1Down>,<Btn1Up> : slithy()
```

This is taken, even if the mouse jiggles a bit between the down and up events. Similarly, any motion event specified in a translation matches any number of motion events. If the motion event causes an action procedure to be invoked, the procedure is invoked after each motion event.

- If an event sequence consists of a sequence of events that is also a non-initial subsequence of another translation, it is not taken if it occurs in the context of the longer sequence. This usually occurs in the following types of sequences:

```
<Btn1Down>,<Btn1Up> : toves()\n\
<Btn1Up> : did()
```

The second translation is taken only if the button release is not preceded by a button press or if there are intervening events between the press and release sequence. This precaution is especially useful when using the repeat notation with buttons and keys because their expansion includes additional events and when specifying motion events because they are included between any two other events. In particular, mouse motion and double-click translations cannot exist in the same translation table.

- For single click on **Button 1 Up** with **Shift** and **Meta**, use:

```
Shift Meta<Btn1Down>, Shift Meta<Btn1Up>: gyre()
```

- The "+" notation allows you to indicate "for any number of clicks greater than or equal to count", such as:

```
Shift <Btn1Up>(2+): and()
```

- To indicate **EnterNotify** with any modifiers, use:

```
<Enter> : gimble()
```

- To indicate **EnterNotify** with no modifiers, use:

```
None <Enter> : in()
```

- To say **EnterNotify** with **Button 1 Down** and **Button 2 Up** and don't care about the other modifiers, use:

  ```
  Button1 ~Button2 <Enter> : the()
  ```

- To say **EnterNotify** with **Button1 Down** and **Button2 Down** exclusively, use:

  ```
  ! Button1 Button2 <Enter> : wabe()
  ```

  You do not need to use a ~ (tilde) with an ! (exclamation point).

Related Information

The **XtActionProc** procedure pointer.

The **XtActionsRec** data structure, **XtCaseProc** data type, **XtKeyProc** data type, **XtStringProc** data type.

The **XtAppAddActions** subroutine, **XtAugmentTranslations** subroutine, **XtConvertCase** subroutine, **XtInstallAccelerators** subroutine, **XtInstallAllAccelerators** subroutine, **XtOverrideTranslations** subroutine, **XtParseAcceleratorTable** subroutine, **XtParseTranslationTable** subroutine, **XtRegisterCaseConverter** subroutine, **XtSetKeyTranslator** subroutine, **XtSetValues** subroutine, **XtUninstallTranslations** subroutine.

Enhanced X-Windows Utility Subroutines Overview

The Enhanced X-Windows Toolkit provides a number of utility functions for Memory management, Graphics Contexts sharing, Merging Expose and Graphics Expose events, and Error handling.

Using Enhanced X-Windows to Manage Memory

The Toolkit memory management routines provide uniform checking for the value of **NULL** pointers and error reporting on memory allocation errors. These routines are completely compatible with the standard C language run time routines **malloc**, **calloc**, **realloc**, and **free** with this added functionality:

- The **XtMalloc**, the **XtNew**, the **XtCalloc**, and the **XtRealloc** subroutines return an error if there is insufficient memory.
- The **XtFree** subroutine returns if the value of **NULL** pointer is passed.
- The **XtRealloc** subroutine allocates new storage if the value of **NULL** pointer is passed.

Using Enhanced X-Windows to Share Graphics Contexts

The Toolkit provides a mechanism that allows clients to share **Graphics Contexts** (**GCs**), reducing both the number of **GCs** created and the number of calls to the server in an application. The **XtGetGC** subroutine obtains shared **GCs**. The **GCs** must be read-only when using the **XtGetGC** subroutine. If a changeable **GC** is needed, use the **XCreateGC** subroutine instead.

Using Enhanced X-Windows to Merge Exposure Events into a Region

The Toolkit provides the **XtAddExposureToRegion** subroutine that merges the **Expose** and **GraphicsExpose** events into a region that clients can process at once, rather than processing individually.

Using Enhanced X-Windows to Handle Errors

The Toolkit allows a client to register a procedure which is called when an error occurs. This facility reports and logs errors, but does not correct them or help you recover.

Related Information

The **XtErrorMsgHandler** data type.

The **XCreateGC** subroutine, **XtAddExposureToRegion** subroutine **XtAppError** subroutine, **XtAppErrorMsg** subroutine, **XtAppGetErrorDatabase** subroutine, **XtAppGetErrorDatabaseText** subroutine, **XtAppSetErrorHandler** subroutine, **XtAppSetErrorMsgHandler** subroutine, **XtAppSetWarningHandler** subroutine, **XtAppSetWarningMsgHandler** subroutine, **XtAppWarning** subroutine, **XtAppWarningMsg** subroutine, **XtCalloc** subroutine, **XtFree** subroutine, **XtGetGC** subroutine, **XtMalloc** subroutine, **XtNew** subroutine, **XtNewString** macro, **XtRealloc** subroutine **XtReleaseGC** subroutine.

List of Enhanced X-Windows Toolkit Data Structures

The **ApplicationShellClassRec** data structure.
The **ApplicationShellPart** data structure.
The **ApplicationShellWidget** data structure.
The **CompositeClassPart** data structure.
The **CompositePart** data structure.
The **ConstraintClassPart** data structure.
The **ConstraintPart** data structure.
The **CoreClassPart** data structure.
The **CorePart** data structure.
The **OverrideShellClassRec** data structure.
The **OverrideShellPart** data structure.
The **OverrideShellWidget** data structure.
The **ShellClassRec** data structure.
The **ShellPart** data structure.
The **ShellWidget** data structure.
The **TopLevelShellClassRec** data structure.
The **TopLevelShellPart** data structure.
The **TopLevelShellWidget** data structure.
The **TransientShellClassRec** data structure.
The **TransientShellPart** data structure.
The **TransientShellWidget** data structure.
The **VendorShellClassRec** data structure.
The **VendorShellPart** data structure.
The **VendorShellWidget** data structure.
The **WMShellClassRec** data structure.
The **WMShellPart** data structure.
The **WMShellWidget** data structure.
The **XrmValue** data structure.
The **XtAcceptFocusProc** data type.
The **XtActionProc** procedure pointer.
The **XtActionList** data structure.
The **XtAddressMode** enumerated type.
The **XtAlmostProc** data type.
The **ArgList** data structure.
The **XtArgsFunc** data type.
The **XtArgsProc** data type.

The **XtCallbackList** data structure.
The **XtCallbackProc** data type.
The **XtCaseProc** data type.
The **XtConvertArgRec** data structure.
The **XtConvertSelectionProc** data type.
The **XtConverter** data type.
The **XtErrorHandler** data type.
The **XtErrorMsgHandler** data type.
The **XtEventHandler** data type.
The **XtExposeProc** data type.
The **XtGeometryHandler** data type.
The **XtGeometryResult** data structure.
The **XtInitProc** data type.
The **XtInputCallbackProc** data type.
The **XtKeyProc** data type.
The **XtLoseSelectionProc** data type.
The **XtOrderProc** data type.
The **XtPopdownID** data structure.
The **XtProc** data type.
The **XtRealizeProc** data type.
The **XtResource** data structure.
The **XtResourceDefaultProc** data type.
The **XtSelectionCallbackProc** data type.
The **XtSelectionDoneProc** data type.
The **XtStringProc** data type.
The **XtTimerCallbackProc** procedure.
The **XtWidgetClassProc** data type.
The **XtWidgetGeometry** data structure.
The **XtWidgetProc** data type.
The **XtWorkProc** data structure.

CoreClassPart Data Structure

The common fields for all widget classes are defined in the **CoreClassPart data structure:**

```
typedef struct {
   WidgetClass superclass;
   String class_name;
   Cardinal widget_size;
   XtProc class_initialize;
   XtWidgetClassProc class_part_initialize;
   Boolean class_inited;
   XtInitProc initialize;
   XtArgsProc initialize_hook;
   XtRealizeProc realize;
   XtActionList actions;
   Cardinal num_actions;
   XtResourceList resources;
   Cardinal num_resources;
   XrmClass xrm_class;
   Boolean compress_motion;
   Boolean compress_exposure;
   Boolean compress_enterleave;
   Boolean visible_interest;
   XtWidgetProc destroy;
   XtWidgetProc resize;
   XtExposeProc expose;
   XtSetValuesFunc set_values;
   XtArgsFunc set_values_hook;
   XtAlmostProc set_values_almost;
   XtArgsProc get_values_hook;
   XtAcceptFocusProc accept_focus;
   XtVersionType version;
   _XtOffsetList callback_private;
   String tm_table;
   XtGeometryHandler query_geometry;
   XtStringProc display_accelerator;
   caddr_t extension;
} CoreClassPart;
```

Related Information

The **XtArgsFunc** data type.

CorePart Data Structure

The common fields for all widget instances are defined in the **CorePart** structure:

```
typedef struct _CorePart {
   Widget self;
   WidgetClass widget_class;
   Widget parent;
   XrmName xrm_name;
   Boolean being_destroyed;
   XtCallbackList destroy_callbacks;
   caddr_t constraints;
   Position x;
   Position y;
   Dimension width;
   Dimension height;
   Dimension border_width;
   Boolean managed;
   Boolean sensitive;
   Boolean ancestor_sensitive;
   XtEventTable event_table;
   XtTMRec tm;
   XtTranslations accelerators;
   Pixel border_pixel;
   Pixmap border_pixmap;
   WidgetList popup_list;
   Cardinal num_popups;
   String name;
   Screen *screen;
   Colormap colormap;
   Window window;
   Cardinal depth;
   Pixel background_pixel;
   Pixmap background_pixmap;
   Boolean visible;
   Boolean mapped_when_managed;
} CorePart;
```

The default values for the core fields are filled in by the **Core** resource list and the **Core** initialize procedure. The default values for the **CorePart** data structure are:

Field	Default Value
self	Address of the widget structure (may not be changed)
widget_class	*widget_class* argument to the **XtCreateWidget** subroutine (may not be changed)
parent	*parent* argument to the **XtCreateWidget** subroutine (may not be changed)
xrm_name	Encoded *name* argument to the **XtCreateWidget** subroutine (may not be changed)
being_destroyed	*being_destroyed* value of the parent widget
destroy_callbacks	NULL
constraints	NULL
x	0
y	0
width	0

height	0
border_width	1
managed	False
sensitive	True
ancestor_sensitive	Bitwise AND of *sensitive* & *ancestor_sensitive* fields of the parent widget
event_table	Initialized by the event manager
tm	Initialized by the translation manager
accelerators	NULL
border_pixel	XtDefaultForeground
border_pixmap	NULL
popup_list	NULL
num_popups	0
name	The *name* argument to the **XtCreateWidget** subroutine (may not be changed)
screen	Parent screen; top-level widget uses display specifier (may not be changed)
colormap	Default colormap for the screen
window	NULL
depth	Parent's depth; top-level widget uses root window depth
background_pixel	XtDefaultBackground
background_pixmap	NULL
visible	True
map_when_managed	True

CompositeClassPart Data Structure

In addition to the **Core** widget class fields, **Composite** widgets have the following class fields:

```
typedef struct {
   XtGeometryHandler geometry_manager;
   XtWidgetProc change_managed;
   XtWidgetProc insert_child;
   XtWidgetProc delete_child;
   caddr_t extension;
} CompositeClassPart;
```

Related Information

The **CompositePart** data structure.

CompositePart Data Structure

In addition to the **CorePart** fields, **Composite** widgets have the following fields defined in the **CompositePart** structure:

```
typedef struct {
   WidgetList children;
   Cardinal num_children;
   Cardinal num_slots;
   XtOrderProc insert_position;
} CompositePart;
```

The default values are filled in by the **Composite** resource list and the **Composite** initialize procedure. The default values for the **CompositePart** data structure are:

Field	Default Value
children	NULL
num_children	0
num_slots	0
insert_position	Internal function **InsertAtEnd**

Related Information

The **CompositeClassPart** data structure.

ConstraintClassPart Data Structure

In addition to the **Composite** class fields, **Constraint** widgets have the following class fields:

```
typedef struct {
   XtResourceList resources;
   Cardinal num_resources;
   Cardinal constraint_size;
   XtInitProc initialize;
   XtWidgetProc destroy;
   XtSetValuesFunc set_values;
   caddr_t extension;
} ConstraintClassPart;
```

Related Information

The **ConstraintPart** data structure.

ConstraintPart Data Structure

In addition to the **CompositePart** fields, **Constraint** widgets have the following fields defined in the **ConstraintPart** data structure:

```
typedef struct { int empty; } ConstraintPart;
```

Related Information

The **ConstraintClassPart** data structure.

ArgList Data Structure

```
typedef something ArgList;

typedef struct {
   String name;
   ArgList value;
} Arg, *ArgList;
```

The **ArgList** data structure is a pointer to the **Arg** data structure. The **ArgList** data type is a C language type which is large enough to contain the following: **caddr_t, char*, long, int*,** or a pointer to a function.

Many of the Intrinsics routines need to receive pairs of resource names and values, called an argument list. These are passed as an **ArgList** structure.

name	Specifies the name of the resource.
value	Contains the resource value if the size of the resource is less than or equal `ArgList`to the size of an **ArgList** structure. Otherwise, the *value* field is a pointer to the resource value.

Related Information

The **XtMergeArgLists** subroutine, **XtSetArg** subroutine.

XtInitProc Data Type

```
typedef void (*XtInitProc)(Widget, Widget)
   Widget request;
   Widget new;
```

The **XtInitProc** procedure is the **initialize** procedure (the procedure specified in the *initialize* field of a widget class) pointer type for a widget class.

An initialization procedure performs the following:

- Allocates space for and copies any resources that are referenced by address. For example, if a widget has a field that is a **String**, it cannot depend on the characters at that address remaining constant but must dynamically allocate space for the string and copy it to the new space. (note that you should not allocate space for or copy callback lists.)
- Computes values for unspecified resource fields. for example, if the width and height are 0, the widget should compute an appropriate width and height based on ohter resources. This is the only time that a widget should ever directly assign its own width and height.
- Computes values for uninitialized nonresource fields that are derived from resource fields. For example, graphics contexts (GCs) that the widget uses are derived from resources like background, foreground, and font.

An initialization procedure can also check certain fields for internal consistancy. For example, it makes no sense to specify a color map for a depth that does not support that color map.

A subclass compares the difference between a specified size and a size computed by a superclass by checking the *request* and *new* fields in the **XtInitProc** data type.

request	Specifies the widget with resource values as requested by the argument list, the resource database, and the widget defaults.

new	Specifies a widget with the new values, both resource and nonresource, that are allowed. A subclass initialize procedure compares the values to resolve potential conflicts.

XtArgsProc Data Type

```
typedef void (*XtArgsProc)(Widget, ArgList, Cardinal*);
   Widget w;
   ArgList args;
   Cardinal *num_args;
```

Description

The **XtArgsProc** specifies the interface for the **initialize_hook** and **get_values_hook** procedures of a widget.

If the **XtArgsProc** procedure is not the value of **NULL**, it is called immediately after the corresponding initialize procedure or in place of it, if the initialize procedure is the value of **NULL**..

Parameters

w	Specifies the widget ID.
args	Specifies the argument list to override the resource defaults.
num_args	Specifies the number of arguments in the argument list.

XtCallbackProc Data Type

```
typedef void (*XtCallbackProc)(Widget, caddr_t, caddr_t);
   Widget w;
   caddr_t client_data;
   caddr_t call_data;
```

The **XtCallbackProc** data type specifies the interface of a callback procedure to be used in callback lists. A client can register the callback and client-specific data (such as a pointer to additional information about the widget) in the *client_data* field. The client data should be the value of **NULL** if all necessary information is in the widget.

The *call_data* field allows the client application to specify data about the callback. This parameter helps the client avoid using the **XtGetValues** subroutine or a widget-specific subroutine to retrieve data from the widget. For example, if the Scrollbar executes the **thumbChanged** callback list, the *call_data* field passes the new position of the thumb.

Note: Do not use the *call_data* field for complex information.

Fields

w	Specifies the widget ID for which the callback is registered.
client_data	Specifies the data that should be passed to the client when the widget executes the client's callback. If all necessary information is in the widget, this parameter has the value of **Null**.
call_data	Specifies any callback data that should be passed to the client when the widget executes the client's callback.

Related Information

The **XtCallbackList** data structure.

The **XtCreateWidget** subroutine, **XtGetValues** subroutine.

XtCallbackList Data Structure

```
typedef struct {
   XtCallbackProc callback;
   caddr_t closure;
} XtCallbackRec, *XtCallbackList;
```

The **XtCallbackList** structure specifies the address of a null-terminated list, and is used to pass callback information to the **XtCreateWidget** subroutine, **XtSetValues** subroutine, or **XtGetValues** subroutine.

callback	Specifies a pointer to the callback procedure.
closure	Specifies any client data to be passed to the callback procedure.

Related Information

The **XtCreateWidget** subroutine, **XtGetValues** subroutine, **XtSetValues** subroutine.

XtWidgetProc Data Type

```
typedef void (*XtWidgetProc)(Widget)
   Widget w;
```

The **XtWidgetProc** data type specifies the interface for a user defined *destroy*, *resize*, *change_managed*, *insert_child*, or *delete_child* procedures of a widget. These procedures are stored in the following fields of a widget:

destroy	Destroy procedures are called in subclass-to-superclass order. Therefore, a widget's destroy procedure should only deallocate storage specific to the subclass and should not bother with the storage allocated by any of its superclasses. The destroy procedure should only deallocate resources that have been explicitly created by the subclass. Any resource that was obtained from the resource database or was passed in with an argument list was not created by the widget and, therefore, should not be destroyed by it. If a widget does not need to deallocate any storage, the destroy procedure entry in its widget class record can be the value of **NULL**.
resize	If the composite widget wishes to change the size or border width of any of its children, it calls the **XtResizeWidget** subroutine, which first updates the **Core** fields and then calls the **XConfigureWindow** subroutine if the widget is realized.

A child can be resized by its parent at any time. Widgets usually need to know when they have changed size so that they can lay out their displayed data again to match the new size. When a parent resizes a child, it calls **XtResizeWidget**, which updates the geometry fields in the widget., configures the window if the widget is realized, and calls the child's resize procedure to notify the child. The resize procedure pointer is of type **XtWidgetProc**.

If a class need not recalculate anything when a widget is resized, it can specify NULL for the resize field in its class record. This is an unusual case and should occur only for widgets with very trivial display semantics. The resize procedure takes a widget as its only argument. The x, y, width, height and border_width fields of the widget contain new values. The resize procedure should recalculate the layout of internal data as needed. (For example, a centered Label in a window that changes size should recalculate the starting position of the text.) The widget must obey resize as a command and must not treat it as a request. A widget must not issue an **XtMakeGeometryRequest** or **XtMakeResizeRequest** call from its resize procedure.

change_managed	Child widgets are added to and removed from the managed set by using the **XtManageChild**, **XtManageChildren**, **XtUnmanageChild**, and **XtUnmanageChildren** subroutines, which notify the parent to recalculate the physical layout of its children by calling the parent's *change_managed* procedure. The **XtCreateManagedWidget** convenience subroutine calls the **XtCreateWidget** and **XtManageChild** subroutines on the result.
insert_child	To add a child to the parent's list of children, the **XtCreateWidget** subroutine calls the parent's class routine *insert_child*.

Most composite widgets inherit their superclass' operation. The *insert_child* routine calls the *insert_position* procedure and inserts the child at the specified position.

Some composite widgets define their own *insert_child* routine so that they can order their children in some convenient way, create companion controller widgets for a new widget, or limit the number or type of their children widgets.

If there is not enough room to insert a new child in the children array (the *num_children* field of the widget == the *num_slots* field), the *insert_child* procedure must first reallocate the array and update the *num_slots* field of the widget. The *insert_child* procedure then places the child wherever it wants and increments the *num_children* field of the widget.

delete_child	Most widgets inherit the *delete_child* procedure from their superclass. Composite widgets that create companion widgets define their own *delete_child* procedure to remove these companion widgets. To remove the child from the parent's children array, the **XtDestroyWidget** function eventually causes a call to the composite parent's *delete_child* procedure.

The **XtWidgetProc** data type contains the following field:

w	Specifies the widget.

Related Information

The **CompositeClassPart** data structure, **ConstraintClassPart** data structure.

The **XFreeGC** subroutine, **XFreePixmap** subroutine, **XtAddEventHandler**subroutine, **XtAppAddTimeout** subroutine, **XtCalloc** subroutine, **XtDe stroyWidget** subroutine, **XtFree** subroutine, **XtGetGC** subroutine, **XtMalloc** subroutine, **XtRemoveEventHandler** subroutine, **XtRemoveTimeout** subroutine, **XtResizeWidget** subroutine.

XtOrderProc Data Type

```
typedef Cardinal (*XtOrderProc)(Widget);
   Widget w;
```

The **XtOrderProc** data type is the interface definition for the **insert_position** procedure in a composite widget instance. This procedure is useful when composite widgets need a specific order for their children widgets; it determines where a new child should go in the children list of the widget. This procedure is called from the **insert_child** procedure of the widget class.

The return value of the **insert_position** procedure indicates how many children should go before the widget

w	Specifies the widget.

Return Values

0	Indicates that the widget should go before all other children.
num_children	Indicates that the widget should go after all other children.

Related Information

The **CompositePart** data structure.

Enhanced X-Windows ShellClassRec Data Structure

```
typedef struct _ShellClassRec {
   CoreClassPart core_class;
   CompositeClassPart composite_class;
   ShellClassPart shell_class;
} ShellClassRec;
```

Related Information

The **ShellPart** data structure.

The **ShellWidget** data structure.

OverrideShellClassRec Data Structure

```
typedef struct _OverrideShellClassRec {
   CoreClassPart core_class;
   CompositeClassPart composite_class;
   ShellClassPart shell_class;
   OverrideShellClassPart override_shell_class;
} OverrideShellClassRec;
```

Related Information

The **OverrideShellPart** data structure.

The **OverrideShellWidget** data structure.

WMShellClassRec Data Structure

```
typedef struct _WMShellClassRec {
   CoreClassPart core_class;
   CompositeClassPart composite_class;
   ShellClassPart shell_class;
   WMShellClassPart wm_shell_class;
} WMShellClassRec;
```

Related Information

The **WMShellPart** data structure.

The **WMShellWidget** data structure.

VendorShellClassRec Data Structure

```
typedef struct _VendorShellClassRec {
   CoreClassPart core_class;
   CompositeClassPart composite_class;
   ShellClassPart shell_class;
   WMShellClassPart wm_shell_class;
   VendorShellClassPart vendor_shell_class;
} VendorShellClassRec;
```

Related Information

The **VendorShellPart** data structure.

The **VendorShellWidget** data structure.

TransientShellClassRec Data Structure

```
typedef struct _TransientShellClassRec {
   CoreClassPart core_class;
   CompositeClassPart composite_class;
   ShellClassPart shell_class;
   WMShellClassPart wm_shell_class;
   VendorShellClassPart vendor_shell_class;
   TransientShellClassPart transient_shell_class;
} TransientShellClassRec;
```

Related Information

The **TransientShellPart** data structure.

The **TransientShellWidget** data structure.

TopLevelShellClassRec Data Structure

```
typedef struct _TopLevelShellClassRec {
   CoreClassPart core_class;
   CompositeClassPart composite_class;
   ShellClassPart shell_class;
   WMShellClassPart wm_shell_class;
   VendorShellClassPart vendor_shell_class;
   TopLevelShellClassPart top_level_shell_class;
} TopLevelShellClassRec;
```

Related Information

The **TopLevelShellPart** data structure.

The **TopLevelShellWidget** data structure.

ApplicationShellClassRec Data Structure

```
typedef struct _ApplicationShellClassRec {
   CoreClassPart core_class;
   CompositeClassPart composite_class;
   ShellClassPart shell_class;
   WMShellClassPart wm_shell_class;
   VendorShellClassPart vendor_shell_class;
   TopLevelShellClassPart top_level_shell_class;
   ApplicationShellClassPart application_shell_class;
} ApplicationShellClassRec;
```

Related Information

The **ApplicationShellPart** data structure.

The **ApplicationShellWidget** data structure.

ShellPart Data Structure

```
typedef struct {
   String geometry;
   XtCreatePopupChildProc create_popup_child_proc;
   XtGrabKind grab_kind;
   Boolean spring_loaded;
   Boolean popped_up;
   Boolean allow_shell_resize;
   Boolean client_specified;
   Boolean save_under;
   Boolean override_redirect;
   XtCallbackList popup_callback;
   XtCallbackList popdown_callback;
} ShellPart;
```

allow_shell_resize	This field controls whether or not the widget contained by the shell is allowed to resize itself. If the value of the field is **False**, geometry requests return the value **XtGeometryNo**. The default value for the *allow_shell_resize* field is **False**.
client_specified	By default, the *client_specified* field is used internally.
create_popup_child_proc	The procedure defined by this field is called by the **XtPopup** subroutine. The default value for the *create_popup_child_proc* field is **NULL**.
geometry	Specifies size and position and is usually done only from a command line or a defaults file. The default value for the *geometry* field is **NULL.**
grab_kind	By default, the *grab_kind* field is used internally.
override_redirect	Setting the *override_redirect* field determines whether or not the shell window is visible to the window manager. If it is the value of **True**, the window is immediately mapped without the intervention of the manager. The default value for the *override_redirect* field is **True** for the **OverrideShell**, and **False** otherwise.
popdown_callback	This field is called during the **XtPopdown** subroutine. The default value for the *popdown_callback* field is **NULL**.
popped_up	By default, the *popped_up* field is used internally.
popup_callback	This is called during the **XtPopup** subroutine. The default value for the *popup_callback* field is **NULL**.
save_under	Setting the *save_under* field instructs the server to attempt to save the contents of windows obscured by the shell when it is mapped and to restore its contents automaticlly later. It can be useful for pop-up menus. The default value for the *save_under* field is **True** for the **OverrideShell** and **TransientShell** widget classes, and **False** otherwise.
spring_loaded	By default, the *spring_loaded* field is used internally.

Related Information

The **ShellClassRec** data structure.
The **ShellWidget** data structure.

OverrideShellPart Data Structure

```
typedef struct { int empty; } OverrideShellPart;
```

Related Information

The **OverrideShellClassRec** data structure.
The **OverrideShellWidget** data structure.

WMShellPart Data Structure

```
typedef struct {
   String title;
   int wm_timeout;
   Boolean wait_for_wm;
   Boolean transient;
   XSizeHints size_hints;
   XWMHints wm_hints;
} WMShellPart;
```

The common shell fields and their default values in the **WMShell** widget class and its subclasses are:

size_hints	Specifies resources for sizing the window. This can include the following:
max_height	The default value for the *max_height* field is **None**.
max_width	The default value for the *max_width* field is **None**.
min_height	The default value for the *min_height* field is **None**.
min_width	The default value for the *min_width* field is **None**.
height_inc	The default value for the *height_inc* field is **None**.
width_inc	The default value for the *width_inc* field is **None**.
title	This value is a string displayed by the window manager. The default value for the *title* field is the icon name if one is specified. Otherwise it is the name of the application.
transient	The default value for the *transient* field is **True** for the **TransientShell** widget class, and **False** otherwise.
wait_for_wm	The default value for the *wait_for_wm* field is **True**. This field is set to **False** when a shell does not receive confirmation of a geometry request to the window manager within the time defined for the *wm_timeout* field. When the field is the value of **False**, the shell does not wait for confirmation but relies on asynchronous notification.
wm_hints	Specifies resources for window manager hints. This can include the following:
min_aspect_x	The default value for the *min_aspect_x* field is **None**.

min_aspect_y	The default value for the *min_aspect_y* field is **None**.
max_aspect_x	The default value for the *max_aspect_x* field is **None**.
max_aspect_y	The default value for the *max_aspect_y* field is **None**.
input	The default value for the *input* field is **False**.
initial_state	The default value for the *initial_state* field is normal.
icon_pixmap	The default value for the *icon_pixmap* field is **None**.
icon_window	The default value for the *icon_window* field is **None**.
icon_x	The default value for the *icon_x* field is **None**.
icon_y	The default value for the *icon_y* field is **None**.
icon_mask	The default value for the *icon_mask* field is **None**.
window_group	The default value for the *window_group* field is **None**.
wm_timeout	Limits the amount of time a shell is to wait for confirmation of a geometry request to the window manager. If no confirmation comes before the defined timeout, the shell assumes the window manager is not functioning properly and sets the *wait_for_wm* field to the value of **False**. The default value for the *wm_timeout* field is five seconds.

Related Information

The **WMShellClassRec** data structure.
The **WMShellWidget** data structure.

VendorShellPart Data Structure

```
typedef struct {
   int vendor_specific;
} VendorShellPart;
```

Related Information

The **VendorShellClassRec** data structure.
The **VendorShellWidget** data structure.

TransientShellPart Data Structure

```
typedef struct { int empty; } TransientShellPart;
```

Related Information

The **TransientShellClassRec** data structure.
The **TransientShellWidget** data structure.

TopLevelShellPart Data Structure

```
typedef struct {
   String icon_name;
   Boolean iconic;
} TopLevelShellPart;
```

The common shell fields and their default values for the **TopLevel** shells are:

icon_name	The default value for *icon_name* is the name of the shell widget. This field contains a string for display in the icon of the shell.
iconic	The default value for *iconic* is **False**. Setting this field to **True** is an alternative way to set the *initialState* resource to indicate that a shell is displayed initially as an icon.

Related Information

The **TopLevelShellClassRec** data structure.
The **TopLevelShellWidget** data structure.

ApplicationShellPart Data Structure

```
typedef struct {
   char *class;
   XrmClass xrm_class;
   int argc;
   char **argv;
} ApplicationShellPart;
```

The common shell fields and their default values for **Application** shells are:

argc	This field is used to initialize the WM_COMMAND standard property. The default value for *argc* is 0 (zero).
argv	This field is used to initialize the WM_COMMAND standard property. The default value for *argv* is **NULL**.

Related Information

The **ApplicationShellClassRec** data structure.
The **ApplicationShellWidget** data structure.

ShellWidget Data Structure

```
typedef struct {
   CorePart core;
   CompositePart composite;
   ShellPart shell;
} ShellRec, *ShellWidget;
```

Related Information

The **ShellClassRec** data structure.
The **ShellPart** data structure.

OverrideShellWidget Data Structure

```
typedef struct {
   CorePart core;
   CompositePart composite;
   ShellPart shell;
   OverrideShellPart override;
} OverrideShellRec, *OverrideShellWidget;
```

Related Information

The **OverrideShellClassRec** data structure.

The **OverrideShellPart** data structure.

WMShellWidget Data Structure

```
typedef struct {
   CorePart core;
   CompositePart composite;
   ShellPart shell;
   WMShellPart wm;
} WMShellRec, *WMShellWidget;
```

Related Information

The **WMShellClassRec** data structure.

The **WMShellPart** data structure.

VendorShellWidget Data Structure

```
typedef struct {
   CorePart core;
   CompositePart composite;
   ShellPart shell;
   WMShellPart wm;
   VendorShellPart vendor;
} VendorShellRec, *VendorShellWidget;
```

Related Information

The **VendorShellClassRec** data structure.

The **VendorShellPart** data structure.

TransientShellWidget Data Structure

```
typedef struct {
   CorePart core;
   CompositePart composite;
   ShellPart shell;
   WMShellPart wm;
   VendorShellPart vendor;
   TransientShellPart transient;
} TransientShellRec, *TransientShellWidget;
```

Related Information

The **TransientShellClassRec** data structure.

The **TransientShellPart** data structure.

TopLevelShellWidget Data Structure

```
typedef struct {
   CorePart core;
   CompositePart composite;
   ShellPart shell;
   WMShellPart wm;
   VendorShellPart vendor;
   TopLevelShellPart topLevel;
} TopLevelShellRec, *TopLevelShellWidget;
```

Related Information

The **TopLevelShellClassRec** data structure.

The **TopLevelShellPart** data structure.

ApplicationShellWidget Data Structure

```
typedef struct {
   CorePart core;
   CompositePart composite;
   ShellPart shell;
   WMShellPart wm;
   VendorShellPart vendor;
   TopLevelShellPart topLevel;
   ApplicationShellPart application;
} ApplicationShellRec, *ApplicationShellWidget;
```

Related Information

The **ApplicationShellClassRec** data structure.

The **ApplicationShellPart** data structure.

XtWorkProc Data Type

```
typedef Boolean(*XtWorkProc)(caddr_t)
   caddr_t client_data;
```

The **XtWorkProc** data type specifies the interface definition for user defined idle–time work procedures. The user defined procedure must return the value of **True** if it is done (that is, the work is complete).

client_data	Specifies the client data generated when the work procedure was registered.

Related Information

The **XtAppAddWorkProc** subroutine.

XtExposeProc Data Type

Syntax

```
typedef void (*XtExposeProc)(Widget, XEvent *, Region)
   Widget w;
   XEvent *event;
   Region region;
```

Description

The **XtExposeProc** data type specifies the interface definition for user defined **expose** procedure in widget classes. The expose procedure redisplays a widget upon exposure (This redisplay is the responsibility of the widget.). If a widget has no display semantics, specify the value of **NULL** for its **expose** procedure. For example, many composite widgets serve as containers for their children only and have no expose procedure.

Note: If the **XtExposeProc** procedure is the value of **NULL**, the **XtRealizeWidget** subroutine fills in the default bit gravity of **NorthWestGravity** before it calls the widget realize procedure.

Fields

event	Specifies the exposure event that identifies the rectangle that requires redisplaying. This parameter contains the bounding box for the *Region* parameter, if the *compress_exposure* field of the widget is the value of **True**.
region	Specifies the union of all rectangles in this exposure sequence. This parameter is the value of **NULL** if the *compress_exposure* field of the widget is the value of **False**.
w	Specifies the ID of the widget instance that requires displaying.

Related Information

The **XtRealizeWidget** subroutine.

XtEventHandler Data Type

```
typedef void (*XtEventHandler)(Widget, caddr_t, XEvent*);
   Widget w;
   caddr_t client_data;
   XEvent *event;
```

The **XtEventHandler** data type is the interface definition for user defined event handler procedures for widgets that must use event handlers explicitly. Most widgets use the translation manager, instead of using event handlers explicitly.

client_data	Specifies the client–specific information registered with the event handler. If the event handler is registered by the widget, this parameter is the value of **NULL**.
event	Specifies the triggering event.
w	Specifies the widget ID for which to handle events.

XtWidgetGeometry Data Structure

The **XtWidgetGeometry** data structure is similar to a corresponding **Xlib** structure.

```
typedef unsigned long XtGeometryMask;

typedef struct {
   XtGeometryMask request_mode;
   Position x, y;
   Dimension width, height;
   Dimension border_width;
   Widget sibling;
   int stack_mode;
} XtWidgetGeometry;
```

The following *request_mode* values are from the **<X11/X.h>** header file:

```
#define CWX                    (1<<0)
#define CWY                    (1<<1)
#define CWWidth                (1<<2)
#define CWHeight               (1<<3)
#define CWBorderWidth          (1<<4)
#define CWSibling              (1<<5)
#define CWStackMode            (1<<6)
```

The Intrinsics also support the following value:

```
#define XtCWQueryOnly          (1<<7)
```

The **XtCWQueryOnly** value indicates that the corresponding geometry request is only a query asking what would happen if this geometry request were made and that no widgets are actually changed.

The **XtMakeGeometryRequest** subroutine (like the corresponding **Xlib XConfigureWindow** subroutine) uses *request_mode* to determine which fields in the **XtWidgetGeometry** structure you want to specify.

The *stack_mode* values are defined in the **<X11/X.h>** header file:

```
#define Above                  0
#define Below                  1
#define TopIf                  2
#define BottomIf               3
#define Opposite               4
```

The Intrinsics also support the following value:

```
#define XtSMDontChange         5
```

The **XtSMDontChange** value indicates that the widget requires its current stacking order preserved.

Related Information

The **XtGeometryResult** data structure.

XtGeometryResult Data Structure

The return codes from geometry managers are:

```
typedef enum _XtGeometryResult {
   XtGeometryYes,
   XtGeometryNo,
   XtGeometryAlmost,
   XtGeometryDone,
} XtGeometryResult;
```

Related Information

The **XtWidgetGeometry** data structure.

XtGeometryHandler Data Type

Syntax

```
typedef XtGeometryResult(*XtGeometryHandler)(Widget,
                                   XtWidgetGeometry*,
                                   XtWidgetGeometry*)
   Widget w;
   XtWidgetGeometry *request;
   XtWidgetGeometry *geometry_return;
```

Description

The **XtGeometryHandler** data type specifies the interface definition for the **geometry_manager** procedure in a composite widget. A class can inherit the geometry manager of its superclass during class initialization.

The same definition is also used for the **query_geometry** procedure in a widget. The **query_geometry** procedure:

- Examines the bits set in the *request–>request_mode* parameter.
- Evaluates the preferred geometry of the widget.
- Stores the result in the *geometry_return* parameter. It sets the bits in the *geometry_return–>request_mode* parameter to the corresponding geometry fields.
- Generates the appropriate return value.

A bit set to the value of **0** in the mask field of the request means that the child widget does not care about the value of the corresponding field. Then the geometry manager can change it as it wishes. A bit set to **1** means that the child wants that geometry element changed to the value in the corresponding field.

If the geometry manager can satisfy all changes requested and if **XtCWQueryOnly** is not specified, it updates the widget's x, y, width, height, and border_width values appropriately. Then it returns **XtGeometryYes**, and the value of the geometry_return argument is undefined. The widget's window is moved and resized automatically by **XtMakeGeometryRequest**.

Homogeneous composite widgets often find it convenient to treat the widget making the request the same as any other widget, possibly reconfiguring it as part of its layout process, unless **XtCWQueryOnly** is specified. If it does this, it should return **XtGeometryDone** to inform **XtMakeGeometryRequest** that it does not need to do the configuration itself.

Although **XtMakeGeometryRequest** resizes the widget's window (if the geometry manager returns **XtGeometryYes**), it does not call the widget class's resize procedure. The requesting widget must perform whatever resizing calculations are needed explicitly.

If the geometry manager chooses to disallow the request, the widget cannot change its geometry. The value of the geometry_return parameter is undefined, and the geometry manager returns **XtGeometryNo**.

Sometimes the geometry manager cannot satisfy the request exactly, but may be able to satisfy a similar request. That is, it could satisfy only a subset of the requests or a lesser request. In such cases, the geometry manager fills in the *geometry_return* field with the actual changes it is willing to make, including an appropriate mask, and returns **XtGeometryAlmost**. If a bit in the *geometry_return–>request_mode* is **0**, the geometry manager does not change the corresponding value if the *geometry_return* field is used immediately in a new request. If a bit is **1**, the geometry manager does change that element to the corresponding value in the *geometry_return* field. More bits may be set in the *geometry_return* field than in the original request if the geometry manager intends to change other fields should the child accept the compromise.

When the **XtGeometryAlmost** value is returned, the widget must decide if the compromise suggested in the *geometry_return* field is acceptable. If it is, the widget must not change its geometry directly; rather, it must make another call to the **XtMakeGeometryRequest** subroutine.

If the next geometry request from this child uses the *geometry_return* box filled in by an **XtGeometryAlmost** return and if there have been no intervening geometry requests on either its parent or any of its other children, the geometry manager must grant the request, if possible. That is, if the child asks immediately with the returned geometry, it should get an answer of **XtGeometryYes**. However, the user's window manager may affect the final outcome.

To return an **XtGeometryYes** value, the geometry manager frequently rearranges the position of other managed children by calling the **XtMoveWidget** subroutine. However, a few geometry managers may sometimes change the size of other managed children by calling the **XtResizeWidget** or the **XtConfigureWidget** subroutine. If **XtCWQueryOnly** is specified, the geometry manager must return how it would react to this geometry request without actually moving or resizing any widgets.

Geometry managers must not assume that the *request* and *geometry_return* fields point to independent storage. The caller is permitted to use the same field for both, and the geometry manager must allocate its own temporary storage , if necessary.

Fields

geometry_return	Specifies the geometry request returned by the geometry manager.
request	Specifies the request for a geometry change.
w	Specifies the widget.

Return Values

XtGeometryAlmost	Indicates that at least one field in the *PreferredReturn* parameter is different from the corresponding field in the *Intended* parameter or if a bit is set in *PreferredReturn* parameter that is not set in the *Intended* parameter of the **query_geometry** procedure.
XtGeometryNo	Indicates that the preferred geometry is identical to the current geometry.

XtGeometryYes	Indicates that the proposed geometry change is acceptable without modification.

Related Information

The **XtQueryGeometry** subroutine.

XtResource Data Structure

The declaration for the **XtResource** data structure is:

```
typedef struct {
   String resource_name;
   String resource_class;
   String resource_type;
   Cardinal resource_size;
   Cardinal resource_offset;
   String default_type;
   caddr_t default_address;
} XtResource, *XtResourceList;
```

The following list describes the content of each of these structure fields:

resource_name Contains the name used by clients to access the field in the widget. This name starts with a lower–case letter. It is spelled almost the same as the field name, except that underscores (_) are deleted, and the next leter replaced by its uppercase counterpart. For example, the resource name for *background_pixel* is **backgroundPixel**. Widget header files typically contain a symbolic name for each resource name. All resource names, classes, and types used by the Intrinsics are in the **<X11/StringDefs.h>** header file. The Intrinsics symbolic resource names begin with **XtN** and are followed by the string name; for example, **XtNbackgroundPixel** for **backgroundPixel**.

resource_class A resource class has two functions:

- It isolates an application from different representations that widgets can use for a similar resource.
- It lets an application specify values for several resources with a single name. A resource class should be chosen to span a group of closely related fields.

For example, a widget can have several pixel resources, such as *background*, *foreground*, *border*, *block cursor*, *mouse cursor*, and so on. Typically, the background defaults to white and everything else defaults to black. The resource class for each of these resources in the resource list should be chosen so that it takes a minimal number of entries in the resource database to make *background* offwhite and everything else darkblue:

In this case, the background pixel should have a resource class of **Background** and all the other pixel entries should have a resource class of **Foreground**. Then, the resource file needs just two lines to change all pixels to offwhite or darkblue:

***Background**:	offwhite
***Foreground**:	darkblue

Similarly, a widget may have several resource fonts, such as normal and bold, but all fonts should have the class **Font**. Changing all fonts requires one line in the default file:

Font: Rom14.500

Resource class names begin with a capitalized letter. This name is preceded by **XtC** (for example. XtCBackground).

resource_type The physical representation type of the resource. This name begins with an uppercase letter and is spelled the same as the type name of the field. The resource type is used when resources are called to convert from the resource database format (usually String) or the default resource format (often String) to the desired physical representation. The Intrinsics define the following resource types:

Resource Type	**Structure or Field Type**
XtRAcceleratorTable	XtAccelerators
XtRBoolean	Boolean
XtRBool	Bool
XtRCallback	XtCallbackList
XtRColor	XColor
XtRCursor	Cursor
XtRDimension	Dimension
XtRDisplay	Display*
XtRFile	FILE*
XtRFloat	float
XtRFont	Font
XtRFontStruct	XFontStruct*
XtRFunction	(*)()
XtRInt	int
XtRPixel	Pixel
XtRPixmap	Pixmap
XtRPointer	caddr_t
XtRPosition	Position
XtRShort	short
XtRString	char*
XtRTranslationTable	XtTranslations
XtRUnsignedChar	unsigned char
XtRWidget	Widget
XtRWindow	Window

resource_size The size of the physical representation in bytes and should be specified as `sizeof(type)` so that the compiler can fill in the value.

resource_offset The offset in bytes of the field within the widget. Use the **XtOffset** macro to retrieve this value.

default_type The representation type of the default resource value. If *default_type* is different from *resource_type* and the *default_type* is needed, the resource manager invokes a conversion procedure from *default_type* to *resource_type*. Whenever possible, the default type should be identical to the resource type to minimize widget creation time. However, there are sometimes no values of the type that the application can easily specify. In this case, the value should be one

	that the converter will work for, such as **XtDefaultForeground** for a pixel resource.
default_address	The address of the default resource value. The default is used if a resource is not specified in the argument list or in the resource database or if the conversion from the representation type stored in the resource database fails, which can happen for reasons (for example, a misspelled entry in a resource file).

Two special representation types, **XtRImmediate** and **XtRCallProc**, can only be used as default resource types. **XtRImmediate** indicates that the value in the *default_address* field is the actual value of the resource, rather than the address of the value. The value must be in correct representation type for the resource. No conversion is possible since there is no source representation type.

XtRCallProc indicates that the value in the *default_address* field is a procedure variable. This procedure is automatically invoked with the *widget*, *resource_offset*, and a pointer to the **XrmValue** data structure in which to store the result.

Related Information

The **<X11/StringDefs.h>** header file.

The **XrmValue** data structure.

The **XtOffset** macro.

XtResourceDefaultProc Data Type

The **XrmValue** data structure is of **XtResourceDefaultProc** data type, which has the following structure:

```
typedef void (*XtResourceDefaultProc)(Widget, int, XrmValue*)
   Widget widget;
   int offset;
   XrmValue *value;
```

The **XtResourceDefaultProc** data type fills in the *addr* field of the *value* parameter with a pointer to the default data in its correct type.

Note: The *default_address* field in the resource structure is declared as a **caddr_t**. On some machine architectures, this may be insufficient to hold procedure variables.

When the *default_address* field of the **XtResource** data structure contains the resource type of **XtRCallProc**, the **XtResourceDefaultProc** data type is automatically called with the widget, the *resource_offset* field, and a pointer to the **XrmValue** data structure in which to store the result.

The fields of the **XtResourceDefaultProc** data type are as follows:

widget	Specifies the widget whose resource is to be obtained.
offset	Specifies the offset of the field in the widget record.
value	Specifies the resource value to fill in.

Related Information

The **XtResource** data structure.

XrmValue Data Structure

```
typedef struct {
   unsigned int size;
   caddr_t addr;
} XrmValue, *XrmValuePtr;
```

XtConverter Data Type

```
typedef void (*XtConverter)(XrmValue*, Cardinal*,
               XrmValue*, XrmValue*);
   XrmValue *args;
   Cardinal *num_args;
   XrmValue *from;
   XrmValue *to;
```

Description

The **Xtconverter** data type specifies the interface definition for subroutines that convert resources from one type to another.

Type converters do the following actions:

- Check to see that the number of arguments passed is correct
- Attempt the type conversion
- If successful, return a pointer to the data in the to parameter; otherwise, call **XtWarningMsg** and return without modifying the to parameter.

Most type converters just take the data described by the from parameter and return data by writing into the specified to parameter. A few need other information which is available in the specified argument list. A type converter can invoke another type converter, which allows differing sources that may convert into a common intermediate result to make maximum use of the type converter cache.

The address written to–>addr can not be that of a lock variable because this is not valid after the converter returns. It should be a pointer to a static variable.

The **XtConverter** data structure contains the following fields:

args	Specifies a list of additional **XrmValue** arguments to the converter if additional context is needed to perform the conversion. Otherwise, this field is the value of **NULL**. For example, the string–to–font converter needs the widget's screen, or the string–to–pixel converter needs the widget's screen and colormap.
num_args	Specifies the number of additional **XrmValue** arguments if additional context is needed. Otherwise, this field is the value of **0**.
from	Specifies the value to convert.
to	Specifies the descriptor to use to return the converted value.

XtAddressMode Enumerated Type

```
typdef enum {
  /*address mode              parameter representation*/
   XtAddress,                 /*address*/
   XtBaseOffset,              /*offset*/
   XtImmediate,               /*constant*/
   XtResourceString,          /*resource name string*/
   XtResourceQuark,           /*resource name quark*/
} XtAddressMode;
```

Related Information

The **XtConvertArgRec** data structure.
The **XtAppAddConverter** subroutine.

XtConvertArgRec Data Structure

```
typedef struct {
   XtAddressMode address_mode;
   caddr_t address_id;
   Cardinal size;
} XtConvertArgRec, *XtConvertArgList;
```

address_id	Specifies the address of the resource. See the *address_mode* field definition for details.
address_mode	Specifies how the *address_id* field should be interpreted. This field may have the following values:
XtAddress	Causes *address_id* to be interpreted as the address of the data.
XtBaseOffset	Causes *address_id* to be interpreted as the offset from the widget base.
XtImmediate	Causes *address_id* to be interpreted as a constant.
XtResourceString	Causes *address_id* to be interpreted as the name of a resource that is to be converted into an offset from a widget base.
XtResourceQuark	Is an internally compiled form of an **XtResourceString**.
size	Specifies the length of the data in bytes.

Related Information

The **XtAddressMode** enumerated type.
The **XtAppAddConverter** subroutine.

XtActionList Data Structure

```
typedef struct _XtActionsRec {
   String action_name;
   XtActionProc action_proc;
} XtActionsRec, *XtActionList;
```

action_name Specifies the name used in translation tables to access the procedure.

action_proc	Specifies a procedure pointer of type **XtActionProc**, which points to a procedure that implements the functionality.

Related Information

The **XtActionProc** procedure pointer.

XtActionProc Procedure Pointer

```
typedef void (*XtActionProc)(Widget, XEvent*,
            String*, Cardinal*);
   Widget w;
   XEvent *event;
   String *params;
   Cardinal *num_params;
```

The **XtActionProc** procedure pointer specifies the interface definition for action procedures. All widget class records contain an action table. In addition, an application can register its own action tables with the translation manager so that the translation tables it provides to widget instances can access application functionality.

w	Specifies the widget that caused the action to be called.
event	Specifies the event that caused the action to be called. If the action is called after a sequence of events, then the last event in the sequence is used.
params	Specifies a pointer to the list of strings that were specified in the translation table as arguments to the action.
num_params	Specifies the number of arguments specified in the translation table.

Related Information

The **XtActionList** data structure.

The **XtAppAddActions** subroutine.

XtKeyProc Data Type

The translation manager provides support for automatically translating key codes in incoming key events into KeySyms. The **XtKeyProc** data type provides the interface definition for Keycode to KeySym translators.

```
typedef void (*XtKeyProc)(Display*, KeyCode, Modifiers,
            Modifiers*, KeySym*);
   Display *display;
   KeyCode keycode;
   Modifiers modifiers;
   Modifiers *modifiers_return;
   KeySym *keysym_return;
```

The **XtKeyProc** data type takes a Keycode and modifiers and produces a KeySym. For a given key translator subroutine, the *modifiers_return* parameter will be a constant that indicates the subset of all modifiers that are examined by the key translator. Applications should register this key converter with the **XtSetKeyTranslator** subroutine.

The following fields are included in the **XtKeyProc** data type:

display	Specifies the display that the key code is from.
keycode	Specifies the key code to translate.

modifiers	Specifies the modifiers to the key code.
modifiers_return	Returns a mask that indicates the subset of all modifiers are examined by the key translator.
keysym_return	Returns the resulting KeySym.

Related Information

The **XtCaseProc** data type.

The **XtRegisterCaseConverter** sburoutine, **XtSetKeyTranslator** subroutine, **XtTranslateKeycode** subroutine.

XtCaseProc Data Type

```
typedef void (*XtCaseProc)(KeySym*, KeySym*, KeySym*);
   KeySym *keysym;
   KeySym *lower_return;
   KeySym *upper_return;
```

The **XtCaseProc** data type specifies the interface for a case converter procedure. It allows capitalization of nonstandard key symbols. A case conversion routine should be registered with the Intrinsics library by using the **XtRegisterCaseConverter** subroutine. The **XtConvertCase** subroutine calls the appropriate user defined **XtCaseProc** *Procedure*. If there is no case distinction, the user defined **XtCaseProc** *Procedure* should store the key symbols in both return values.

keysym	Specifies the key symbol for the conversion.
lower_return	Specifies the lowercase equivalent for the key symbol.
upper_return	Specifies the uppercase equivalent for the key symbol.

Related Information

The **XtConvertCase** subroutine, **XtRegisterCaseConverter** subroutine.

XtAcceptFocusProc Data Type

```
typedef Boolean (*XtAcceptFocusProc)(Widget, Time);
   Widget w;
   Time *time;
```

The **XtAcceptFocusProc** data type defines the user written interface for the *accept_focus* procedure. To allow outside agents to cause a widget to get the input focus, every widget exports an accept_focus procedure. The widget returns even when it does not accept the focus, so that the parent can give the focus to another widget.

Widgets that must know when they lose the input focus should use the **Xlib** library focus notification mechanism explicitly by specifying translations for the **FocusIn** and **FocusOut** events.

Widgets that do not want the input focus should set the *accept_focus* procedure pointer to the value of **NULL**.

Widgets that need the input focus can call the **XSetInputFocus** subroutine explicitly.

time	Specifies the X time of the event causing the *accept_focus* procedure.

w	Specifies the widget ID.

Related Information

The **XtCallAcceptFocus** subroutine, **XSetInputFocus** subroutine, **XtSetKeyboardFocus** subroutine.

XtAlmostProc Data Type

```
typedef void (*XtAlmostProc)(Widget, Widget, XtWidgetGeometry*,
                XtWidgetGeometry*);
   Widget w;
   Widget new_widget_return;
   XtWidgetGeometry *request;
   XTWidgetGeometry *reply;
```

The **XtAlmostProc** data type defines the interface for the *set_values_almost* field of a widget. This field is a procedure pointer. Most classes inherit this operation from their superclass by specifying **XtInheritSetValuesAlmost** in the class initialization. The core *set_values_almost* field accepts the compromise suggest.

The *set_values_almost* procedure is called when the geometry manager cannot satisfy a client's request to set the window geometry with the **XtSetValues** subroutine. The geometry manager returns the **XtGeometryAlmost** value with a compromise geometry.

The *set_values_almost* procedure takes the original geometry and the compromise geometry and determines an acceptable compromise, which may be the current compromise or a different compromise. It returns the results in the *new_widget_return* field, which is then sent to the geometry manager for another try.

new_widget_return	Specifies the new widget which will store the geometry changes.
reply	Specifies the compromise geometry returned by the geometry manager.
request	Specifies the original geometry request sent to the geometry manager.
w	Specifies the widget ID on which the geometry change is requested.

XtArgsFunc Data Type

```
typedef Boolean (*XtArgsFunc)(Widget, Arglist, Cardinal*);
   Widget w;
   ArgList args;
   Cardinal *num_args;
```

The **XtArgsFunc** specifies the interface for the *set_values_hook* procedure pointer of a widget. Widgets with a subpart can set the resource values with the **XtSetValues** subroutine and a *set_values_hook* procedure.

args	Specifies the argument list for the **XtCreateWidget** subroutine.
num_args	Specifies the number of arguments in the argument list.
w	Specifies the widget ID whose non–widget resource values are to be changed.

Related Information

The **CoreClassPart** data structure.

The **XtCreateWidget** subroutine, **XtSetValues** subroutine.

XtPopdownID Data Structure

```
typedef struct {
   Widget shell_widget;
   Widget enable_widget;
} XtPopdownIDRec, *XtPopdownID;
```

The **XtPopdownID** structure identifies the widgets involved in the **XtCallbackPopdown** subroutine. The address of an **XtPopdownID** structure is passed as the client data to the **XtCallbackPopdown** subroutine.

enable_widget	Specifies the widget used to pop the (pop-up) shell.
shell_widget	Specifies the pop-up shell to be popped down.

Related Information

The **XtCallbackPopdown** subroutine, **XtSetSensitive** subroutine.

XtConvertSelectionProc Data Type

```
typedef Boolean (*XtConvertSelectionProc)(Widget, Atom*, Atom*,
                Atom*, caddr_t*, unsigned long*, int*);
   Widget w;
   Atom *selection;
   Atom *target;
   Atom *type_return;
   caddr_t *value_return;
   unsigned long *length_return;
   int *format_return;
```

The **XtConvertSelectionProc** data type is the interface definition for user defined procedures that get the value of a selection as a given type from the current selection owner.

Each **XtConvertSelectionProc** data type should respond to the target value **TARGETS** by returning a value containing the list of the targets that will be used for the selection conversion.

format_return	Specifies a pointer into which the size (in bits) of the data elements of the selection value is to be stored.
length_return	Specifies a pointer into which the number of elements in the *ValueReturn* parameter is to be stored.
selection	Specifies the atom that describes the type of selection requested. (For example, **XA_PRIMARY** or **XA_SECONDARY**.)
target	Specifies the type of the selection requested, for example, a filename, text, or a window.
type_return	Specifies a pointer to an atom that will store the converted value of the selection. For example, a filename or text could specify **XA_STRING** as the value for this parameter.

value_return	Specifies a pointer into which a pointer to the converted value of the selection is stored. The selection owner is responsible for allocating this storage.

If the selection owner provided an **XtSelectionDoneProc** procedure for the selection, the storage is owned by the selection owner. Otherwise, the storage is owned by the Intrinsics selection mechanism.

w	Specifies the ID of the widget that currently owns the selection.

Return Values

False	Indicates that the conversion did not take place. The values of the return parameters are undefined.
True	Indicates that the selection owner successfully converted the selection to the target type.

Related Information

The **XtLoseSelectionProc** data type, **XtSelectionDoneProc** data type, **XtSelectionCallbackProc** data type.

The **XtDisownSelection** subroutine, **XtGetSelectionValue** subroutine, **XtGetSelectionValues** subroutine, **XtOwnSelection** subroutine.

XtLoseSelectionProc Data Type

```
typedef void (*XtLoseSelectionProc)(Widget, Atom*);
   Widget w;
   Atom* selection;
```

The **XtLoseSelectionProc** data type specifies the interface definition for user defined procedures that are called by the Intrinsics selection mechanism to inform the specified widget that it has lost ownership of the specified selection. The user defined procedures do not initiate the loss of the selection ownership.

selection	Specifies the atom that describes the selection type.
w	Specifies the widget that has lost selection ownership.

Related Information

The **XtConvertSelectionProc** data type, **XtSelectionCallbackProc** data type, **XtSelectionDoneProc** data type.

The **XtOwnSelection** subroutine.

XtSelectionCallbackProc Data Type

```
typedef void (*XtSelectionCallbackProc)(Widget, caddr_t, Atom*,
              Atom*, caddr_t, unsigned long*, int*);
   Widget w;
   caddr_t client_data;
   Atom* selection;
   Atom* type;
   caddr_t value;
   unsigned long* length;
   int* format;
```

The **XtSelectionCallbackProc** data type specifies the interface for the user defined procedures that are called by the Intrinsics selection mechanism to deliver the requested selection to the requester.

An **XT_CONVERT_FAIL** atom specified in the *Type* parameter, upon return of the user defined procedure, indicates that the selection conversion failed because the selection owner did not respond within the Intrinsics selection time–out interval.

Parameters

client_data	Specifies a value passed in by the widget when the selection was requested.
format	Specifies the size in bits of the data elements of value.
length	Specifies the number of elements in value.
selection	Specifies the type of selection that was requested.
type	Specifies the type used to represent the selection value (for example, **XA_STRING**).
value	Specifies a pointer to the selection value.

The requesting client owns the storage allocated for this parameter. Use the **XtFree** subroutine to de–allocate the storage space when this routine completes.

w	Specifies the widget that requested the selection value.

Related Information

The **XtConvertSelectionProc** data type, **XtLoseSelectionProc** data type, **XtSelectionDoneProc** data type.

The **XtFree** subroutine, **XtGetSelectionValue** subroutine, **XtGetSelectionValues** subroutine.

XtSelectionDoneProc Data Type

```
typedef void (*XtSelectionDoneProc)(Widget, Atom*, Atom*);
   Widget w;
   Atom* selection;
   Atom* target;
```

The **XtSelectionDoneProc** data type specifies the interface definition for user defined procedures that are called by the Intrinsics selection mechanism to inform the selection owner when a selection requester has retrieved a selection value successfully.

Once the selection owner registers an **XtSelectionDoneProc**, the procedure will be called once for each conversion that it performs. This procedure is called after the converted value has been transferred successfully to the requester.

The selection owner that registers an **XtSelectionDoneProc** also owns the storage containing the converted selection value.

selection	Specifies the atom that describes the selection type that was converted.
target	Specifies the target type to which the conversion was done.
w	Specifies the ID of the widget that owns the converted selection.

Related Information

The **XtConvertSelectionProc** data type, **XtLoseSelectionProc** data type, **XtSelectionCallbackProc** data type.

XtErrorHandler Data Type

```
typedef void (*XtErrorHandler)(String);
   String message;
```

The **XtErrorHandler** data type specifies the interface definition for the low–level error and warning handler procedure. The error handler should display the specified message string in an appropriate manner.

message	Specifies the error message.

Related Information

The **XtAppSetErrorHandler** subroutine, **XtAppSetWarningHandler** subroutine.

XtErrorMsgHandler Data Type

```
typedef void (*XtErrorMsgHandler)(String, String, String,
              String, String *, Cardinal *);
   String name;
   String type;
   String class;
   String defaultp;
   String * params;
   Cardinal* num_params;
```

The **XtErrorMsgHandler** data type specifies the interface definition for high–level error and warning handler procedures.

The standard **printf** notation is used to substitute the parameters into the message.

class	Specifies the resource class of the error message.
defaultp	Specifies a default message if an error database entry is not found.
name	Specifies the name that is concatenated with the specified type to form the resource name of the error message. The specified name can be a general error, such as **invalidParameters** or **invalidWindow**.
num_params	Specifies the number of values in the parameter list.
params	Specifies a pointer to a list of values to be substituted in the message.
type	Specifies the type that is concatenated with the name to form the resource name of the error message. The specified type gives extra information.

Related Information

The **XtAppSetErrorMsgHandler** subroutine, **XtAppSetWarningMsgHandler** subroutine.

XtInputCallbackProc Data Type

```
typedef void (*XtInputCallbackProc)(caddr_t, int*, XtInputId*);
   caddr_t client_data;
   int* source;
   XtInputId* id;
```

The **XtInputCallbackProc** data type specifies the interface definition for callback procedures used when there are file events.

client_data	Specifies the client data registered for this procedure in the **XtAppAddInput** subroutine.
id	Specifies the ID returned from the corresponding **XtAppAddInput** subroutine.
source	Specifies the source file descriptor generating the event.

Related Information

The **XtAppAddInput** subroutine, **XtRemoveInput** subroutine.

XtProc Data Type

```
typedef void (*XtProc)();
```

The **XtProc** data type specifies the interface definition for the class initialization procedure pointer type. Most class records can be initialized completely at compile time. In some cases, however, a class may need to register type converters or perform other sorts of one–time initialization.

A widget class indicates that is has no class initialization procedure by specifying the value of **NULL** in its *class_initialize* field.

Related Information

The **XtWidgetClassProc** data type.

XtWidgetClassProc Data Type

```
typedef void (*XtWidgetClassProc)(WidgetClass);
  WidgetClass widget_class;
```

The **XtWidgetClassProc** data type provides the interface definition for the user defined procedures that will be stored in the *class_part_initialize* procedure field of a widget.

Widgets have class initializations will be called exactly once. Some classes need to perform additional initializations for fields in its part of the class record. These are performed not just for the particular class, but subclasses as well.

For classes that do not define new class fields and do not need extra processing, the value of **NULL** should be specified in the *class_part_initialize* field of a widget class.

widget_class	Specifies the class of the widget.

Related Information

The **XtProc** data type.

XtRealizeProc Data Type

```
typedef void (*XtRealizeProc)(Widget, XtValueMask*,
                XSetWindowAttributes*);
   Widget w;
   XtValueMask *value_mask;
   XSetWindowAttributes *attributes;
```

The **XtRealizeProc** data type specifies the interface definition for the user defined realize procedure in a widget class.

The generic **XtRealizeWidget** subroutine fills in a mask and a corresponding **XSetWindowAttributes** data structure. It sets the following fields based on information in the **Core** structure:

- The *background_pixmap* field (or the *background_pixel* field if the *background_pixmap* field is NULL) is filled in from the corresponding field.
- The *border_pixmap* field (or the *border_pixel* field if the *border_pixmap* field is NULL) is filled in from the corresponding field.
- The *event_mask* field is filled in based on the event handlers registered, the event translations specified, whether the *expose* field is non-NULL, and whether the *visible_interest* field is **True**.
- The *bit_gravity* field is set to **NorthWestGravity** if the *expose* field is NULL.
- The *do_not_propagate_mask* field is set to propagate all pointer and keyboard events up the window tree. A composite widget can implement functionality caused by an event anywhere inside it (including on top of children widgets) as long as children do not specify a translation for the event.

All other fields in attributes (and the corresponding bits in *value_mask*) can be set by the *realize* procedure.

Note that because the *realize* procedure is not a chained operation, the widget class *realize* procedure must update the **XSetWindowAttributes** structure with all the appropriate fields from non-**Core** superclasses.

A widget class can inherit its *realize* procedure from its superclass during class initialization. The *realize* procedure defined for **Core** calls the **XtCreateWindow** subroutine with the passed *ValueMask* and *Attributes* parameters and with the *WindowClass* and *VisualPtr* parameters set to **CopyFromParent**. Both **CompositeWidgetClass** and **ConstraintWidgetClass** inherit this *realize* procedure, and most new widget subclasses can do the same.

The most common noninherited *realize* procedures set the bit_gravity in the mask and the attributes to the appropriate value and then create the window. For example, depending on its justification, the *bit_gravity* field can be set to the value of **WestGravity, CenterGravity,** or **EastGravity**. Consequently, shrinking the widget just moves the bits appropriately, and no **Expose** event is needed for repainting.

If the children of a composite widget should be realized in a particular order (typically to control the stacking order), that composit widget should call the **XtRealizeWidget** subroutine on its children in the appropriate order from within its own *realize* procedure.

Widgets that have children and that are not a subclass of **compositeWidgetClass** are responsible for calling the **XtRealizeWidget** subroutine on their children, usually from within the *realize* procedure.

The **XtRealizeProc** data type contains the following fields:

attributes	Specifies the window attributes to use in the **XCreateWindow** subroutine.
value_mask	Specifies the fields from the attributes structure to use.
w	Specifies the ID of the widget.

Related Information

The **XSetWindowAttributes** data structure.

The **XtCreateWindow** subroutine, **XtRealizeWidget** subroutine.

XtSetValuesFunc Data Type

```
typedef Boolean (*XtSetValuesFunc)(Widget, Widget, Widget);
   Widget current;
   Widget request;
   Widget new;
```

The **XtSetValuesFunc** data type specifies the inderface definition for the user defined procedure in the *set_values* field of a widget class. This procedure should recompute any field derived from resources that are changed. If no recomputation is necessary and if none of the resources specific to a subclass require the window to be redisplayed when their values are changed, you can specify the value of **NULL** for the *set_values* field in the class record.

Like the **initialize** field, the *set_values* field mostly deals with the fields defined in the subclass, but it has to resolve conflicts with its superclass, especially conflicts over width and height. Sometimes it is necessary for a subclass to overwrite values filled in by its superclass, particularly in the case of size calculations.

The *new* and *request* parameters provide information for a subclass to determine the difference between a specified size and a size computed by its superclass. The *request* parameter is the widget as originally requested. The *new* parameter starts with the values of the *request* parameter, but has been modified by all superclass *set_values* fields called so far. A widget does not need to refer to the *request* parameter unless it must resolve conflicts between the widget in the *current* parameter and the widget in the *new* parameter. Any changes, including geometry changes, that the widget needs to make should be made to the widget specified in the *new* parameter.

The *set_values* field must return a Boolean value that indicates if the widget needs to be redisplayed. A change in the geometry fields alone does not require the *set_values* field to return the value of **True**; the X Server will eventually generate an **Expose** event, if necessary. After calling all the *set_values* fields, the **XtSetValues** subroutine forces a redisplay by calling the **XClearArea** subroutine if any of the *set_values* procedures returned the value of **True**. Therefore, a *set_values* field should not do its own redisplaying.

The *set_values* field should not do any work in response to changes in geometry. A widget should do any geometry–related work in its **Resize** procedure.

It is permissible to call **XtSetValues** before a widget is realized. Therefore, the *set_values* field must not assume that the widget is realized.

current	Specifies a copy of the widget as it was before the **XtSetValues** subroutine was called.

request Specifies a copy of the widget with all values changed as specified in the **XtSetValues** subroutine before any class *set_values* fields have been called.

new Specifies the widget with the new values that are actually allowed.

Related Information

The **XClearArea** subroutine, **XtSetValues** subroutine.

XtStringProc Data Type

```
typedef void (*XtStringProc)(Widget, String)
   Widget w;
   String* string;
```

The **XtStringProc** data type specifies the interface definition for the user defined procedures stored in the *display_accelerator* field of a widget. Accelerators can be specified in default files, and the string representation is the same as for a translation table.

However, the interpretation of the #augment and #override directives apply to what will happen when the accelerator is installed, that is, whether or not the accelerator translations will override the translations in the destination widget. The default is #augment, which means that the accelerator translations have lower priority than the destination translations. The #replace directive is ignored for accelerator tables.

string Specifies the string representation of the accelerators for this widget.

w Specifies the ID of the widget that the accelerators are installed on.

Related Information

The **CoreClassPart** data structure.

The **XtParseAcceleratorTable** subroutine, **XtInstallAccelerators** subroutine, **XtInstallAllAccelerators** subroutine.

XtTimerCallbackProc Procedure

```
typedef void (*XtXtTimerCallbackProc)
            (caddr_t, XtIntervalID*);
   caddr_t client_data;
   XtIntervalId *id;
```

The **XtTimerCallbackProc** procedure specifies the interface definition for the user defined procedures to be used when time-outs expire.

client_data Specifies the client data that was registered for this procedure in the **XtAppAddTimeOut** subroutine.

id Specifies the ID returned from the corresponding **XtAppAddTimeOut** subroutine.

Related Information

The **XtAppAddTimeOut** subroutine.

The X Protocol Overview

The Core protocol is composed of requests, replies, errors and events. General information about protocol formats, syntax, types, errors, keyboard key assignments, pointers, and predefined atoms is followed by general information about connection setup, connection close, event generation, and flow control.

Using the X Protocol Request Format

Every protocol request contains an 8-bit major opcode and a 16-bit length field expressed in units of 4 bytes. A protocol request consists of a 4-byte header containing the major opcode, the length field, and a data byte, followed by zero or more additional bytes of data. Unused bytes in a protocol request are not required to be the value of 0.

The length field in the protocol request defines the total length of the protocol request, including the header. The length 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, an error is generated.

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, and all other fields in extension requests, are not defined by the Core protocol. Extensions are intended to contain multiple requests.

Major opcodes 128 through 255 are reserved for extensions.

Every protocol request on a given connection is implicitly assigned a sequence number starting with 1 used in replies, errors, and events.

Using the X Protocol Reply Format

Every protocol reply contains a 32-bit length field expressed in units of 4 bytes. A protocol reply consists of 32 bytes, followed by zero or additional bytes of data, as specified in the length field. Unused bytes within a protocol reply are not guaranteed to be the value of 0. A protocol reply contains the least significant 16 bits of the sequence number of the corresponding protocol request.

Using the X Protocol Error Format

Protocol error reports are 32 bytes long. A protocol error includes an 8-bit error code. Every error includes the major and minor opcodes of the failed request and the least significant 16 bits of the sequence number of the request. Unused bytes within an error are not guaranteed to be zero. Error codes 128 through 255 are reserved for extensions.

The failing resource ID is returned for the following errors:

Colormap	**Cursor**
Drawable	**Font**
GContext	**IDChoice**
Pixmap	**Window**

The atom that failed is returned for **Atom** errors.

The value that failed is returned for **Value** errors.

Other Core errors return no additional data.

Using the X Protocol Event Format

Protocol events are 32 bytes long. Protocol events contain an 8-bit type code. The most significant bit in this code is set if the event is generated by a **SendEvent** request. Unused bytes within a protocol event are not guaranteed to be the value of 0. Event codes 64 through 127 are reserved for extensions, although the Core protocol does not define a mechanism for selecting interest in such events. Every Core event, with the exception of **KeymapNotify**, contains the least significant 16 bits of the sequence number of the last protocol request issued by the client that was, or is currently being, processed by the server.

Using the X Protocol Syntax

The following conventions are used to help you identify certain components.

Component	Syntax
set of alternatives	Appear in **boldface** within braces ({...}).
set of structure components	Appear within brackets ([...]).
types	Appear in UPPERCASE.
alternative values	Appear in Initial Caps.
requests	Appear in this format: **RequestName** *arg1: type1* ... *rgN: typeN* => *result1: type1* ... *resultM: typeM* Errors: **kind2, ..., kindK** Description. The return symbol ("=>") in the request code definition indicates the request has one or more replies. If no symbol is present, then the request has no reply and is asynchronous. Errors can still be reported.
events	Appear in this format: **EventName** *value1: type1* ... *valueN: typeN* Description

Using Enhanced X-Windows Common Protocol Types

INT16 and **CARD16** are defined in the **Xmd.h** file of machine-dependent declarations.

Type	Description
ARC	[*x, y*: **INT16** *width, height*: **CARD16** *angle1, angle2*: **INT16**]

ATOM	32-bit value (top 3 bits guaranteed to be 0)
BITGRAVITY	**{Forget, Static, NorthWest, North, NorthEast, West, Center, East, SouthWest, South, SouthEast}**
BITMASK	Various requests contain arguments of the following form: *value-mask*: **BITMASK** These values allow the client to specify a subset of a heterogeneous collection of optional arguments. The *value-mask* specifies which arguments are to be provided; each such value is assigned a unique bit position. The representation of the **BITMASK** typically contains more bits than there are defined values; unused bits in the *value-mask* must be the value of 0. Otherwise, the server generates a **Value** error.
BOOL	**{True, False}**
BUTMASK	**{Button1, Button2, Button3, Button4, Button5}**
BUTTON	**CARD8**
BYTE	8-bit value
CHAR2B	[*byte1, byte2*: **CARD8**]
CARD8	8-bit unsigned integer
CARD16	16-bit unsigned integer
CARD32	32-bit unsigned integer
COLORMAP	32-bit value (top 3 bits guaranteed to be 0)
CURSOR	32-bit value (top 3 bits guaranteed to be 0)
DEVICEEVENT	**{KeyPress, KeyRelease, ButtonPress, ButtonRelease, PointerMotion, Button1Motion, Button2Motion, Button3Motion, Button4Motion, Button5Motion, ButtonMotion}**
DRAWABLE	**WINDOW** or **PIXMAP**
EVENT	**{KeyPress, KeyRelease, OwnerGrabButton, ButtonPress, ButtonRelease, EnterWindow, LeaveWindow, PointerMotion, PointerMotionHint, Button1Motion, Button2Motion, Button3Motion, Button4Motion, Button5Motion, ButtonMotion, Exposure, VisibilityChange, ResizeRedirect, StructureNotify, SubstructureNotify, SubstructureRedirect, FocusChange, PropertyChange, ColormapChange, KeymapState}**
FONT	32-bit value (top 3 bits guaranteed to be 0)
FONTABLE	**FONT** or **GCONTEXT**
GCONTEXT	32-bit value (top 3 bits guaranteed to be 0)
HOST:	[*family*: **{Internet}** *address*: **LISTofBYTE**] The length, format, and interpretation of a **HOST** address are specific to the family
INT8	8-bit signed integer

INT16	16-bit signed integer
INT32	32-bit signed integer
KEYCODE	**CARD8**
KEYBUTMASK	**KEYMASK** or **BUTMASK**
KEYMASK	{**Shift, Lock, Control, Mod1, Mod2, Mod3, Mod4, Mod5**}
KEYSYM	32-bit value (top 3 bits guaranteed to be 0)
LISTofFOO	A type name in the form of **LISTofFOO** means a counted list of elements of type **FOO**. The size of the length field may vary and is not necessarily the same size as a **FOO**. In some cases, the size may be implicit and not fully specified in this document. Except where explicitly noted, 0 length lists are legal.
LISTofVALUE	Various requests contain arguments of the following form: *value-list*: LISTofVALUE These arguments allow the client to specify a subset of a heterogeneous collection of *optional* arguments. The *value-list* contains one value for each bit set to 1 in the associated BITMASK, from the least to the most significant bit in the mask. Each value is represented with 4 bytes, but the actual value occupies only the least significant bytes as required. The values of the unused bytes do not matter.
OR	A type of the form T1 or ... or Tn means the union of the indicated types; a single-element type is given as the element without enclosing braces.
PIXMAP	32-bit value (top 3 bits guaranteed to be 0)
POINT	[*x, y*: **INT16**]
POINTEREVENT	{**ButtonPress, ButtonRelease, EnterWindow, LeaveWindow, PointerMotion, PointerMotionHint, Button1Motion, Button2Motion, Button3Motion, Button4Motion, Button5Motion, ButtonMotion KeymapState**}
RECTANGLE	[*x, y*: **INT16** *width, height*: **CARD16**] The [*x, y*] coordinates of a **RECTANGLE** specify the upper-left corner.
STRING8	**LISTofCARD8**
STRING16	**LISTofCHAR2B** The primary interpretation of *large* characters in a **STRING16** is that these characters are composed of two bytes used to index a 2-D matrix; hence the use of **CHAR2B** rather than **CARD16**. This corresponds to the JIS/ISO method of indexing 2-byte characters. It is expected that most large fonts will be defined with 2-byte matrix indexing. For large fonts constructed with linear indexing, a **CHAR2B** can be interpreted as a 16-bit number by treating *byte1* as the most significant byte. This means clients should always transmit such 16-bit character values most significant byte first, as the sever will never byte swap **CHAR2B** quantities.
TIMESTAMP	**CARD32**
VISUALID	32-bit value (top 3 bits guaranteed to be 0)

VALUE	32-bit quantity (used only in **LISTofVALUE**)
WINDOW	32-bit value (top 3 bits guaranteed to be 0)
WINGRAVITY	{**Unmap, Static, NorthWest, North, NorthEast, West, Center, East, SouthWest, South, SouthEast**}

Using the X Protocol Error Codes

Usually, when a request terminates in an error, the request has no side effects (that is, there is no partial execution). The only requests for which this is not true are the **ChangeWindowAttributes**, the **ChangeGC**, the **PolyText8**, the **PolyText16**, the **FreeColors**, the **StoreColors**, and the **ChangeKeyboardControl** protocol requests.

These error codes can be returned by the protocol requests. An error code can be returned for more than one reason.

Error Code	**Cause**
Access	A client attempted to grab a key or button combination already grabbed by another client.
	A client attempted to free a colormap entry that it did not already allocate.
	A client attempted to store into a read-only or an unallocated colormap entry.
	A client attempted to modify the access control list from a host other than the local (or otherwise authorized client).
	A client attempted to select an event type that another client has already selected and only one client can select at a time.
Alloc	The server failed to allocate the requested resource or server memory.
	Note: This only covers allocation errors at a very course level and is not intended to cover all cases of a server running out of allocation space in the middle of service. The semantics when a server runs out of allocation space are left unspecified, but the server may generate an **Alloc** error on any request for this reason and that client should be prepared to receive such events and discard them.
ATOM	A value for an **ATOM** argument does not name a defined **ATOM**.
Colormap	A value for a **Colormap** argument does not name a defined **Colormap**.
	The variable type is extended by union with a set of fixed alternatives, for example, **<WINDOW or PointerRoot or None>**.
Cursor	A value for a **Cursor** argument does not name a defined **Cursor**.
	The variable type is extended by union with a set of fixed alternatives, for example, **<WINDOW or PointerRoot or None>**.
Drawable	A value for a **Drawable** argument does not name a defined window or **Pixmap**.
	The variable type is extended by union with a set of fixed alternatives, for example, **<WINDOW or PointerRoot or None>**.
Font	A value for a **Font** arguemnt does not name a defined **Font**.
	A value for a **Fontable** argument does not name a defined font or a defined **GContext**.

	The variable type is extended by union with a set of fixed alternatives, for example, **<WINDOW or PointerRoot or None>**.
GContext	A value for a **GContext** argument does not name a defined **GContext**.
	The variable type is extended by union with a set of fixed alternatives, for example, **<WINDOW or PointerRoot or None>**.
IDChoice	The value chosen for a resource identifier is either not included in the range assigned to the client or is already in use.
Implementation	The server does not implement some aspect of the request. A server that generates this error for a core request is deficient. As such, this error is not listed for any of the requests, but clients should be prepared to receive such errors and handle or discard them.
Length	The length of a request is shorter or longer than that minimally required to contain the variables.
	The length of a request exceeds the maximum length required accepted by the server.
Match	In a graphics request, the root and depth of the **GContext** argument does not match that of the destination.
	An **InputOnly** window is used as a **Drawable** argument.
	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.
Name	A font or color of the specified name does not exist.
Pixmap	A value for a **Pixmap** argument does not name a defined **Pixmap**.
	The variable type is extended by union with a set of fixed alternatives, for example, **<WINDOW or PointerRoot or None>**.
Request	The major or minor opcode does not specify a valid request.
Value	Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the arguemnts type is accepted. Any argument defined as a set of alternatives can generate this error due to the encoding.
Window	A value for a **Window** argument does not name a defined **Window**.
	The variable type is extended by union with a set of fixed alternatives, for example, **<WINDOW or PointerRoot or None>**.

Using Enhanced X-Windows with Predefined Atoms

Predefined atoms are not necessary and may not be useful in all environments, but eliminate many **InternAtom** requests in most applications. Note that *predefined* is used for numeric values, not for required semantics. The Core protocol imposes no semantics on these names, except as they are used in the **FONTPROP** structures.

The following names have predefined atom values. (These atom names should be all uppercase.)

ARC	ITALIC_ANGLE	STRING
ATOM	MAX_SPACE	SUBSCRIPT_X
BITMAP	MIN_SPACE	SUBSCRIPT_Y
CAP_HEIGHT	NORM_SPACE	SUPERSCRIPT_X

CARDINAL	NOTICE	SUPERSCRIPT_Y
COLORMAP	PIXMAP	UNDERLINE_POSITION
COPYRIGHT	POINT	UNDERLINE_THICKNESS
CURSOR	POINT_SIZE	VISUALID
CUT_BUFFER0	PRIMARY	WEIGHT
CUT_BUFFER1	QUAD_WIDTH	WINDOW
CUT_BUFFER2	RECTANGLE	WM_CLASS
CUT_BUFFER3	RESOLUTION	WM_CLIENT_MACHINE
CUT_BUFFER4	RESOURCE_MANAGER	WM_COMMAND
CUT_BUFFER5	RGB_BEST_MAP	WM_HINTS
CUT_BUFFER6	RGB_BLUE-MAP	WM_ICON_NAME
CUT_BUFFER7	RGB_COLOR_MAP	WM_ICON_SIZE
DRAWABLE	RGB_DEFAULT_MAP	WM_NAME
END_SPACE	RGB_GRAY_MAP	WM_NORMAL_HINTS
FAMILY_NAME	RGB_GREEN_MAP	WM_SIZE_HINTS
FONT	RGB_RED_MAP	WM_TRANSIENT_FOR
FONT_NAME	SECONDARY	WM_ZOOM_HINTS
FULL_NAME	STRIKEOUT_ASCENT	X_HEIGHT
INTEGER	STRIKEOUT_DESCENT	

Names private to a particular application or end user, but stored in globally accessible locations, should begin with two leading underscores. To avoid conflicts with other names, for which semantics might be imposed (either at the protocol level or in terms of higher level user interface models), names beginning with an underscore should be used for atoms that are private to a particular vendor or organization. To guarantee that conflicts between vendors and organizations are avoided, additional prefixes should be used, but the mechanism for choosing such prefixes is not defined here.

Using Enhanced X-Windows to Set Up a Remote Connection

For remote clients, the X protocol can be built on top of any reliable byte stream. Use the following steps to set up a connection.

The client must send an initial byte of data to identify the byte order to be used. The value of the byte must be octal 102 or 154.

The octal 102 (ASCII uppercase B) means values are transmitted with the most-significant byte first. Octal 154 (ASCII lowercase l) means values are transmitted with the least-significant byte first. Except where explicitly noted in the protocol, all 16-bit and 32-bit quantities sent by the client must be transmitted with this byte order, and all 16-bit and 32-bit quantities returned by the server will be transmitted with this byte order.

After the byte-order byte, the following information is transmitted by the client at connection setup time:

- protocol-major-version: **CARD16**
- protocol-minor-version: **CARD16**
- authorization-protocol-name: **STRING8**
- authorization-protocol-data: **STRING8**

The *version* numbers indicate the version of the protocol the client expects the server to implement.

The *authorization* name indicates the authorization protocol the client expects the server to use and the data specific to that protocol.

Specifying valid authorization mechanisms is not part of the Core protocol. IH's hoped eventually, one authorization protocol will be agreed upon. In the meantime, a server that implements a protocol other than the protocol the client expects or a server that implements only the host-based mechanism can simply ignore this information. If both name and data strings are empty, this occurrence is to be interpreted as *no explicit authorization.*

The client receives the information in the following order.

• success:	**BOOL**
• protocol-major-version:	**CARD16**
• protocol-minor-version:	**CARD16**
• length:	**CARD16**

The *length* is the amount (in units of 4-bytes) of additional data to follow.

The version numbers are for future protocol revisions. Generally, the *major* version would increment for incompatible changes and the *minor* version would increment for small upward compatible changes. If there are no other protocol revisions, the *major* version is 11, and the *minor* version is zero.

The *protocol version* numbers indicate the protocol version supported by the server. The protocol version supported by the server may not equal the version sent by the client. The server can, but does not have to, refuse connections from clients with a version different from the version supported by the server. A server can, but does not have to, support more than one version simultaneously.

The client receives the following additional data if authorization fails:

• reason:	**STRING8**

The client receives the following additional data if authorization is accepted:

• vendor:	**STRING8**
• release-number:	**CARD32**
• resource-id-base,resource-id-mask:	**CARD32**
• image-byte-order:	{**LSBFirst, MSBFirst**}
• bitmap-scanline-unit:	{8, 16, 32}
• bitmap-scanline-pad:	{8, 16, 32}
• bitmap-bit-order:	{**LeastSignificant, MostSignificant**}
• pixmap-formats:	**LISTofFORMAT**
• roots:	**LISTofSCREEN**
• motion-buffer-size:	**CARD32**
• maximum-request-length:	**CARD16**
• min-keycode.max-keycode:	**KEYCODE**

where:

FORMAT:	[depth: **CARD8**, bits-per-pixel: {1, 4, 8, 16, 24, 32} scanline-pad: {8, 16, 32}]

SCREEN:	[root: **WINDOW** width-in-pixels, height-in-pixels: **CARD16** width-in-millimeters, height-in-millimeters: **CARD16** allowed-depths: **LISTofDEPTH** root-depth: **CARD8** root-visual: **VISUALID** default-colormap: **COLORMAP** white-pixel, black-pixel: **CARD32** min-installed-maps, max-installed-maps: **CARD16** backing-stores: {**Never, WhenMapped, Always**} save-unders: **BOOL** current-input-masks: **SETofEVENT**]
DEPTH:	[depth: **CARD8** visuals: **LISTofVISUALTYPE**]
VISUALTYPE:	[visual-id: **VISUALID** class: {**StaticGray, StaticColor, TrueColor, GrayScale, PseudoColor, DirectColor**} red-mask: **CARD32** green-mask: **CARD32** blue-mask: **CARD32** bits-per-rgb-value: **CARD8** colormap-entries: **CARD16**]

The following information is global to the server:

- The *vendor string* gives some identification to the owner of the server implementation.
- The *vendor* controls the semantics of the release number.
- The *resource-id-mask* contains a single contiguous set of bits (at least 18). The client allocates resource IDs by choosing a value with only a subset of these bits set and ORing it with *resource-id-base* for the following types: **WINDOW, PIXMAP, CURSOR, FONT, GCONTEXT,** and **COLORMAP**. Only values constructed in this way can be used to name newly created resources over this connection. Resource IDs never have the top three bits set. The client is not restricted to linear or contiguous allocation of resource IDs. An ID, once freed, can be reused.

  An ID must be unique with respect to the IDs of all other resources, not just other resources of the same type. Note, however, that the value spaces of resource identifiers, atoms, visualids, and keysyms are distinguished by context. As such, they are not required to be disjointed. For example, a given numeric value might be both a valid window ID and a valid atom as well as a valid keysym.
- Although the server is, in general, responsible for byte-swapping data to match the client, images are always transmitted and received in formats (including byte order) specified by the server. The byte order for images is given by *image-byte-order*, and applies to each scanline unit in **XYFormat** (bitmap) format and to each pixel value in **ZFormat**.
- A bitmap is represented in scanline order. Each scanline is padded to a multiple of bits as given by *bitmap-scanline-pad*. The pad bits are of arbitrary value.

  The scanline is quantified in multiples of bits as given by *bitmap-scanline-unit*. The *bitmap-scanline-unit* is always less than or equal to the *bitmap-scanline-pad*.

  Within each unit, the leftmost bit in the bitmap is either the least-significant bit or the most-significant bit in the unit, as given by *bitmap-bit-order*. If a pixmap is represented in

XYFormat, each plane is represented as a bitmap, and the planes appear from the most-significant to the least-significant in bit order, with no padding between planes.

- The *pixmap-formats* contains one entry for each depth value. The entry describes the **ZFormat** used to represent images of that depth. An entry for a depth is included if any screen supports that depth, and all screens supporting that depth must support only that **ZFormat** for that depth. In **ZFormat**, the pixels are in scanline order, left to right, within a scanline.

  The number of bits used to hold each pixel is given by *bits-per-pixel.* These may be larger than strictly required by the depth, in which case the least-significant bits hold the pixmap data and the values of the unused high-order bits are undefined. When the *bits-per-pixel* is four, the order of nibbles in the byte is the same as the *image-byte-order.* When the *bits-per-pixel* is one; the format is identical for bitmap format. Each scanline is padded to a multiple of bits as given by *scanline-pad*; when *bits-per-pixel* is one, this will be identical to *bitmap-scanline-pad.*

- The server implementation, which is transparent to the protocol, determines how a pointing device moves on the screen. No geometry among screens is defined.

- The server can retain the recent history of pointer motion with a finer granularity than is reported by **MotionNotify** events. Such history is available by using the **GetMotionEvents** request. The approximate size of the history buffer is given by *motion-buffer-size.*

- The *maximum-request-length* specifies in 4-byte units the maximum length of a request that can be accepted by the server. Requests larger than this value generate a **Length** error. If an error is generated, server reads and discards the entire request; the *maximum-request-length* is always at least 4096. Requests, up to and including 16384 bytes in length, are accepted by all servers.

- The *min-keycode* specifies the smallest keycode values transmitted by the server. The *min-keycode* is never less than 8.

- The *max-keycode* specifies the largest keycode values transmitted by the server. The *max-keycode* is never greater than 255. Not all keycodes in this range are required to have corresponding keys.

The following information is provided about each screen:

- The *allowed-depths* specifies which pixmap and window depths are supported. Pixmaps are supported for each depth listed. Windows of that depth are supported if at least one visual type is listed for the depth. A pixmap depth of one is always supported and listed, but windows of depth one might not be supported. A depth of zero is never listed, but zero-depth **InputOnly** windows are always supported.

- The *root-depth* specifies the depth of the root window.

- The *root-visual* specifies the visual type of the root window.

- The *width-in-pixels* and *height-in-pixels* specify the size of the root window. The *width-in-pixels* and *height-in-pixels* cannot be changed.

- The *class* of the root window is always **InputOutput**.

- The *width-in-millimeters* and *height-in-millimeters* can be used to determine the physical size and the aspect ratio.

- The *default-colormap* is the colormap initially associated with the root window. Clients with minimal color requirements, that are creating windows of the same depth as the root window, may want to allocate from this colormap by default.

- The *black-pixel* and *white-pixel* can be used in implementing a monochrome application. These pixel values are for permanently allocated entries in the *default-colormap*. The actual RGB values can be set on some screens and may not actually be black and white. The names black and white are intended to convey the expected relative intensity of the colors.

  The border of the root window is initially a pixmap filled with the *black-pixel*. The initial background of the root window is a pixmap filled with some unspecified two-color pattern using *black-pixel* and *white-pixel*.

- The *min-installed-maps* specifies the number of colormaps that can be guaranteed to be installed simultaneously with **InstallColormap**, regardless of the number of entries allocated in each colormap.

- The *max-installed-maps* field specifies the maximum number of colormaps that can be installed simultaneously, depending on their allocations. Multiple static-visual colormaps with identical contents but different resource IDs should be considered as a single colormap for the purposes of this number. The values for a single hardware colormap is a one.

- The *backing-stores* indicates when the server supports backing stores for this screen. However, the backing store might have limited storage for the number of windows it can support simultaneously.

- If *save-unders* field is the value of **True**, then the server can support the *save-under* mode in **CreateWindow** and **ChangeWindowAttributes**, although again it may be limited storage.

- The *current-input-events* field is the value returned by **GetWindowAttributes** for the *all-event-masks* field for the root window.

The following information is provided about each visual type:

- A visual type can be listed for more than one depth or for more than one screen.

- For **PseudoColor**, a pixel value indexes a colormap that produces independent RGB values. The RGB values can be changed dynamically.

- For **GrayScale**, a pixel value indexes a colormap that produces independent RGB values, except that the primary that drives the screen is undefined. Consequently, the client should always store the same value for red, green, and blue in colormaps.

- For **DirectColor**, a pixel value is decomposed into separate RGB subfields, and each subfield separately indexes the colormap indexes for the corresponding value. The RGB values can be changed dynamically.

  The *red-mask*, *green-mask*, and *blue-mask* are defined for **DirectColor**. Each mask has one contiguous set of bits set to 1 with no intersections. Each mask usually has the same number of one bits.

- For **TrueColor**, a pixel value is decomposed into separate RGB subfields, and each subfield separately indexes the colormap indexes for the corresponding value. This colormap has predefined read-only RGB values which are server-dependent, but provide near-linear or linear increasing ramps in each primary.

  The *red-mask*, *green-mask*, and *blue-mask* fields are defined for **TrueColor**. Each mask has one contiguous set of bits set to 1 with no intersections. Each mask usually has the same number of one bits.

- For **StaticColor**, a pixel value indexes a colormap that produces independent RGB values. This colormap has predefined read-only RGB values, which are server-dependent.
- For **StaticGray**, a pixel value indexes a colormap that produces independent RGB values. The red, green, and blue values in this colormap are equal for any single pixel value, resulting in shades of gray. **StaticGray** with a 2-entry colormap can be thought of as monochrome.
- The *bits-per-rgb-value* field specifies the log base 2, the number of distinct color intensity values (individually) of red, green, and blue. This number is not necessarily related to the number of colormap entries. Actual RGB values are always passed in the protocol within a 16-bit spectrum, with zero being minimum intensity and 65535 being the maximum intensity. On hardware that provides a linear zero-based intensity ramp, the following relationship exists:

```
hw-intensity = protocol-intensity / (65536 /
 total-hw-intensities)
```

- Colormap entries are indexed from 0. The *colormap-entries* defines the number of colormap entries available in a newly created colormap. For **DirectColor** and **TrueColor**, this is usually two to the power of the maximum number of one bits in *red-mask*, *green-mask*, and *blue-mask*.

Using Enhanced X-Windows to Close Connections to the Server

At the time of connection close, all event selections made by the client are discarded in the following ways:

- If the client has the pointer actively grabbed, an **UngrabPointer** is performed, .
- If the client has the keyboard actively grabbed, an **UngrabKeyboard** is performed. All passive grabs by the client are released.
- If the client has the server grabbed, an **UngrabServer** is performed, disowning all selections owned by the client.
- All selections owned by the client are disowned.
- If close-down mode is **RetainPermanent** or **RetainTemporary**, then all resources, including colormap entries, allocated by the client are marked as permanent or temporary. This marking does not prevent other clients from destroying these resources explicitly.
- If the mode is **Destroy**, all client resources are destroyed.

  When the resources of a client and the members in the client save-set, that is an inferior of a window created by the client, are destroyed, the save-set window is reparented to the closest ancestor so that the save-set window is not an inferior of a window created by the client. If the save-set window is unmapped, a **MapWindow** request is performed on it, even if it is not an inferior of a window created by the client. The reparenting leaves unchanges the absolute coordinates (with respect to the root window) of the upper-left outer corner of the same set window. After save-set processing, all windows created by the client are destroyed. For each non-window resource created by the client, the appropriate **Free** request is performed. All colors and colormap entries allocated by the client are freed.

  When the last connection to a server closes, a server goes through a cycle of having no connections and having some connections. The server resets its state to its initial state as if it had just been started, with each transition of the state having no connections (that is, the **Destroy** close-down mode forced a connection closing). The server starts the cycle by destroying all lingering resources of clients that were terminated in **RetainPermanent** or **RetainTemporary** mode. The X Server does the following:

- Deletes everything except the predefined atom identifiers
- Deletes all the properties on all root windows
- Resets all device maps and attributes (key click, bell volume, acceleration) to initial value
- Resets the access control list to initial value
- Restores the standard root tiles and cursors to initial value
- Restores the default font path to initial value
- Restores the input focus to state **PointerRoot**.

Closing a connection with a close-down mode of **RetainPermanent** or **RetainTemporary** does not reset the server.

Using Enhanced X-Windows to Generate an Event

When a button is pressed with the pointer in window *W* and no active pointer grab is in progress, then the ancestors of *W* are searched from the root down for a passive grab to activate. If a matching passive grab on the button does not exist, then an active grab is started automatically for the client receiving the event, and the *last-pointer-grab* time is set to the current server time. The effect is equivalent to a **GrabButton** protocol request with the following arguments:

Argument	Value
event-window	The event window.
event-mask	Selected pointer events of the client on the event window.
pointer-mode	**Asynchronous**.
keyboard-mode	**Asynchronous**.
owner-events	**True** if the client has **OwnerGrabButton** selected on the event window. Otherwise, it is set to the value of **False**.
confine-to	None.
cursor	None.

The grab is terminated automatically when all the buttons are released. **UngrabPointer** and **ChangeActiveGrab** can both be used to modify the active grab.

Using Enhanced X-Windows to Control Flow and Concurrency

When the server is writing to a specific connection, it can stop reading from that connection. However, if the writing blocks, the server must continue to service other connections. The server is not required to buffer more than a single request per connection at one time. For a given connection to the server, a client can block while reading from the connection, but should undertake to read (events and errors) when writing would block. Failure on the part of a client to do this could result in a deadlocked connection, although deadlock is unlikely unless the transport layer has very little buffering, or the client attempts to send large numbers of requests without reading replies or checking for errors and events.

If a server is implemented with internal concurrency, the overall effect must be that individual requests are executed to completion in a serial order, and that requests from a given connection are executed in delivery order. In other words, the total execution order is a shuffle of the individual streams. The execution of a request includes validating all arguments, collecting all data for any reply, and generating (and queueing) all required events, but does not include the actual transmission of the reply and the events. In addition, the effect of any other cause, such as the activation of a pointer motion that can generate multiple events, must effectively generate (and queue) all required events indivisibly with respect to all other causes and requests. For a request from a given client, any events destined for that client which are caused by executing the request must be sent to the client before any error or reply is sent.

Related Information

The **QueryFont** protocol request, **SetCloseDownMode** protocol request, **SetSelectionOwner** protocol request.

The aixterm Command HFT Display Characteristics Overview

VTLs are supported with the following characteristics:

- Locator reports are given for the following mouse events:
  - `ButtonPress`
  - `ButtonRelease`
  - `MotionNotify` (with any button down)
  - `LeftDownMotion`
  - `MiddleDownMotion`
- Locator reports are given in absolute coordinates.
- Locator motion is processed in compressed mode.
- Applications need not display a locator cursor.

Function	**HFT**	**aixterm**
TERMINAL CONTROL		
Query I/O Error	ioctl	no support
Query Device	ioctl	no support
Reconfigure	ioctl	no support
Set Echo and Break Map	ioctl	no support
Set Keyboard Map	ioctl	hftctl
Get Virtual Terminal ID	ioctl	no support
Query Device ID	ioctl	no support
Query Physical Device (ID=0)	ioctl	no support
Query HFT device	ioctl	hftctl
Query Mouse/Tablet	ioctl	hftctl
Query LPFKs	ioctl	no support
Query Dials	ioctl	no support
Query PS	ioctl	hftctl
Enable / Disable Sound	ioctl	no support
Enter / Exit Monitor Md	ioctl	no support
Query Screen Manager	ioctl	no support

Control Screen Manager	ioctl	no support
KSR Protocol	VTD	VTD
Set KSR Color Palette	VTD	no support
Change Fonts	VTD	no support
Cursor Representation	VTD	VTD
INPUT		
Dead Key Support	read	supported
Untranslated Key	read	supported
Input Device Report	read	supported
mouse	VTL	no support
tablet	VTL	supported
dials	VTL	no support
Lighted Program Function Keys	VTL	no support
Sound Complete	signal	no support
OUTPUT		
Write ASCII (Code Page 850)	write	supported
Set LEDs	VTD	no support
Set LPFKs	VTD	no support
Set Dials	VTD	no support
Sound output	VTD	no support
Cancel Sound	VTD	no support
Change Physical Display	VTD	no support

Related Information

The **aixterm** command

Using the aixterm Command Datastream Support

The following is a list of the escape sequences supported by the **aixterm** command.

Some escape sequences activate and deactivate an alternate screen buffer that is the same size as the display area of the window. This capability allows the contents of the screen to be saved and restored. When the alternate screen is activated, the current screen is saved and replaced with the alternate screen. The saving of lines scrolled off of the window is disabled until the original screen is restored.

This table uses these abbreviations in the righthand column:

Xv	Supported by the **aixterm** command running in vt100 mode.
Xh	Supported by the **aixterm** command running in hft mode.
H	Found in the hft datastream.
V	Found in the vt100 datastream.

Name	Function	Datastream	Support
	SINGLE–BYTE CONTROLS		
BEL	Bell	0x07	Xv, Xh, H, V
BS	Backspace	0x08	Xv, Xh, H, V
HT	Horizontal tab	0x09	Xv, Xh, H, V
LF	Linefeed	0x0A	Xv, Xh, H, V
VT	Vertical tab	0x0B	Xv, Xh, H, V
FF	Form feed	0x0C	Xv, Xh, H, V
CR	Carriage return	0x0D	Xv, Xh, H, V
SO	Shift out	0x0E	Xv, Xh, H, V
SI	Shift in	0x0F	Xv, Xh, H, V
DC1	Device control 1	0x11	H, V
DC3	Device control 3	0x13	H, V
CAN	Cancel	0x18	H, V
SUB	Substitute (also cancels)	0x1A	H, V
ESC	Escape	0x1B	Xv, Xh, H, V
SS4	Single Shift 4	0x1C	H
SS3	Single Shift 3	0x1D	H
SS2	Single Shift 2	0x1E	H
SS1	Single Shift 1	0x1F	H
cbt	cursor back tab	ESC [Pn Z	Xv, Xh, H
cha	cursor horizontal absolute	ESC [Pn G	Xv, Xh, H
cht	cursor horizontal tab	ESC [Pn I	H
ctc	cursor tab stop control	ESC [Pn W	H
cnl	cursor next line	ESC [Pn E	H
cpl	cursor preceding line	ESC [Pn F	Xv, Xh, H
cpr	cursor position report	ESC [Pl; Pc R	Xv, Xh, H, V
cub	cursor backward	ESC [Pn D	Xv, Xh, H, V
cud	cursor down	ESC [Pn B	Xv, Xh, H, V
cuf	cursor forward	ESC [Pn C	Xv, Xh, H, V
cup	cursor position	ESC [Pl; Pc H	Xv, Xh, H, V
cuu	cursor up	ESC [Pn A	Xv, Xh, H, V
cvt	cursor vertical tab	ESC [Pn Y	H
da1	DEVICE ATTRIBUTES request (host to vt100) request (host to vt100) response (vt100 to host)	 ESC [c ESC [0 c ESC [? 1 ; 2 c	 Xv, Xh, V Xv, Xh, V Xv, Xh, V
dch	delete character	ESC [Pn P	Xv, Xh, H
decaln	screen alignment display	ESC # 8	Xv, Xh, V

deck-pam	keypad application mode	ESC =	Xv, V
deckpnm	keypad numeric mode	ESC >	Xv, V
decrc	restore cursor & attributes	ESC 8	Xv, Xh, V
decsc	save cursor & attributes	ESC 7	Xv, Xh, V
decstbm	set top & bottom margins	ESC [Pt; Pb r	Xv, Xh, V
dl	delete line	ESC [Pn M	Xv, Xh, H
dsr	device status report 0 response from vt100: ready 5 command from host: please report status 6 command from host: report active position 13 error report sent from virtual terminal to host	ESC [Ps n	 Xv, Xh, V Xv, Xh, V Xv, Xh, H, V H
dmi	disable manual input	ESC ' (back quote)	H
emi	enable manual input	ESC b	H
ea	erase area 0 erase to end of area 1 erase from area start 2 erase all of area	ESC [Ps O	 Xv, Xh, H Xv, Xh, H Xv, Xh, H
ed	erase display 0 erase to end of display 1 erase from display star 2 erase all of display	ESC [Ps J	 Xv, Xh, H, V Xv, Xh, H, V Xv, Xh, H, V
ef	erase field–e,s,all 0 erase to end of field 1 erase from field start 2 erase all of field	ESC [Ps N	 Xv, Xh, H Xv, Xh, H Xv, Xh, H
el	erase line 0 erase to end of line 1 erase from start of line 2 erase all of line	ESC [Ps K	 Xv, Xh, H, V Xv, Xh, H, V Xv, Xh, H, V
ech	erase character	ESC [Pn X	Xv, Xh, H
hts	horizontal tab stop	ESC H	Xv, Xh, H, V
hvp	horizontal and vertical position	ESC [Pl; Pc f	Xv, Xh, H, V
ich	insert character	ESC [Pn @	Xv, Xh, H
il	insert line	ESC [Pn L	Xv, Xh, H
ind	index	ESC D	Xv, Xh, H, V
ls2	lock shift G2	ESC n	Xv
ls3	lock shift G3	ESC o	Xv
nel	next line	ESC E	Xv, Xh, H, V
ksi	keyboard status information	ESC [Ps p	H

pfk	PF key report	ESC [Pn q	Xh, H
rcp	restore cursor position	ESC [u	Xv, Xh, H
ri	reverse index	ESC M	Xv, Xh, H, V
ris	reset to initial state	ESC c	Xv, Xh, H, V
rm	reset mode, ANSI specified modes: See "set mode" below in this column.	ESC [Ps;...;Ps	
	reset mode, other private modes and XTERM private modes: See "set mode" below in this column.	ESC [? Ps;...;Ps l	
	restore mode, other private modes and XTERM private modes: See "set mode" below in this column.	ESC [? P;...;Ps r	
	save mode, other private modes and XTERM private modes: See "set mode". below in this column.	ESC [? Ps;...;Ps s	
scp	save cursor postion	ESC [s	Xv, Xh, H
scs	select character set		
	United Kingdom Set	ESC (A (GO) ESC) A (G1) ESC * A (G2) ESC + A (G3)	Xv, V Xv, V Xv, V Xv, V
	ASCII Set (USASCII)	ESC (B (G0) ESC) B (G1) ESC * B (G2) ESC + B (G3)	Xv, V Xv, V Xv, V Xv, V
	special graphics	ESC (0 (G0) ESC) 0 (G1) ESC * 0 (G2) ESC + 0 (G3)	Xv, V Xv, V Xv, V Xv, V
sd	scroll down	ESC [Pn T	H
sl	scroll left	ESC [Pn Sp @	H
sr	scroll right	ESC [Pn Sp A	H
ss2	single shift G2	ESC N	Xv
ss3	single shift G3	ESC O	Xv
su	scroll up	ESC [Pn S	Xv, Xh, H

sgr	set graphic rendition	ESC [Ps m	
	0 normal		Xv, Xh, H, V
	1 bold		Xv, Xh, H, V
	4 underscore		Xv, Xh, H, V
	5 blink (appears as bold)		Xv, Xh, H, V
	7 reverse		Xv, Xh, H, V
	8 invisible		Xh, H
	10..17 fonts		Xh, H
	30..37 foreground colors		Xh, H
	40..47 background colors		Xh, H
	90..97 foreground colors		Xh, H
	100..107 background colors		Xh, H
sg0a	set GO character set	ESC (<	Xh, H
sg1a	set G1 character set	ESC) <	Xh, H
sm	set mode		
	ANSI specified modes	ESC [Ps;...;Ps h	
	4 IRM insert mode		Xv, Xh, H
	12 SRM send/rec mode		H
	18 TSM tab stop mode		H
	20 LNM linefeed/newline		Xv, Xh, H, V
	Other private modes	ESC [? Ps;...;Ps h	
	1 normal/application cursor		Xv, V
	3 80/132 columns		Xv, Xh, V
	4 smooth/jump scroll		Xv, Xh, V
	5 reverse/normal video		Xv, Xh, V
	6 origin/normal		Xv, Xh, V
	7 on/off autowrap		Xv, Xh, H, V
	8 on/off autorept		Xv, Xh, V
	21 CNM CR–NL		H
	XTERM private modes		
	40 132/80 column mode		Xv, Xh
	41 curses(5) fix		Xv, Xh
	42 hide/show scrollbar		Xv, Xh
	43 on/off save scroll text		Xv, Xh
	44 on/off margin bell		Xv, Xh
	45 on/off reverse wraparound		Xv, Xh
	47 alternate/normal screen buffer		Xv, Xh
	48 reverse/normal status line		Xv, Xh
	49 page/normal scroll mode		Xv, Xh
tbc	tabulation clear	ESC [Ps g (default Ps =0)	
	0 clear horizontal tab stop at active position		Xv, Xh, H, V
	1 vertical tab at line indicated by cursor		H
	2 horizontal tabs on line		H
	3 all horizontal tabs		Xv, Xh, H, V
	4 all vertical tabs		H

VTD	virtual terminal data	ESC [x	Xv, Xh, H
VTL	virtual terminal device input	ESC [y	Xh, H
VTR	vt raw keyboard input	ESC [w	Xh, H
vts	vertical tab stop	ESC I	H
xes	erase status line	ESC [? E	Xv, Xh
xrs	return from status line	ESC [? F	Xv, Xh
xhs	hide status line	ESC [? H	Xv, Xh
xss	show status line	ESC [? S	Xv, Xh
xgs	go to column of status line	ESC [? Ps T	Xv, Xh
xst	set text parameters 0 change window name and title to Pt	ESC] Ps ; Pt \007	Xv, Xh Xv, Xh

Related Information

The **aixterm** command.

List of the Xlib Library Cursor Fonts

The following is a list of cursors and their shapes in fonts that are currently available with Enhanced X-Windows. These cursors are defined in the **/usr/include/XII/cursorfont.h** directory.

#define	**XC_num_glyphs**	154
#define	**XC_X_cursor**	0
#define	**XC_arrow**	2
#define	**XC_based_arrow_down**	4
#define	**XC_based_arrow_up**	6
#define	**XC_boat**	8
#define	**XC_bogosity**	10
#define	**XC_bottom_left_corner**	12
#define	**XC_bottom_right_corner**	14
#define	**XC_bottom_side**	16
#define	**XC_bottom_tee**	18
#define	**XC_box_sprial**	20
#define	**XC_center_ptr**	22
#define	**XC_circle**	24
#define	**XC_clock**	26
#define	**XC_coffee_mug**	28
#define	**XC_cross**	30
#define	**XC_cross_reverse**	32
#define	**XC_crosshair**	34
#define	**XC_diamond_cross**	36

```
#define     XC_dot                  38
#define     XC_dotbox               40
#define     XC_double_arrow         42
#define     XC_draft_large          44
#define     XC_draft_small          46
#define     XC_draped_box           48
#define     XC_exchange             50
#define     XC_fleur                52
#define     XC_gobbler              54
#define     XC_gumby                56
#define     XC_hand1                58
#define     XC_hand2                60
#define     XC_heart                62
#define     XC_icon                 64
#define     XC_iron_cross           66
#define     XC_left_ptr             68
#define     XC_left_side            70
#define     XC_left_tee             72
#define     XC_leftbutton           74
#define     XC_ll_angle             76
#define     XC_lr_angle             78
#define     XC_man                  80
#define     XC_middlebutton         82
#define     XC_mouse                84
#define     XC_pencil               86
#define     XC_pirate               88
#define     XC_plus                 90
#define     XC_question_arrow       92
#define     XC_right_ptr            94
#define     XC_right_side           96
#define     XC_right_tee            98
#define     XC_rightbutton          100
#define     XC_rtl_logo             102
#define     XC_sailboat             104
#define     XC_sb_down_arrow        106
#define     XC_sb_h_double_arrow    108
#define     XC_sb_left_arrow        110
#define     XC_sb_right_arrow       112
```

```
#define    XC_sb_up_arrow          114
#define    XC_sb_v_double_arrow    116
#define    XC_shuttle              118
#define    XC_sizing               120
#define    XC_spider               122
#define    XC_spraycan             124
#define    XC_star                 126
#define    XC_target               128
#define    XC_tcross               130
#define    XC_top_left_arrow       132
#define    XC_top_left_corner      134
#define    XC_top_right_corner     136
#define    XC_top_side             138
#define    XC_top_tee              140
#define    XC_trek                 142
#define    XC_ul_angle             144
#define    XC_umbrella             146
#define    XC_ur_angle             148
#define    XC_watch                150
#define    XC_xterm                152
#define    XC_aixwm                154
```

Related Information

The **XCreateFontCursor** subroutine.

Chapter 7. Curses and Extended Curses

Curses and Extended Curses Overview

The IBM AIX Version 3 for RISC System/6000 contains two libraries of routines to support input and output to the terminal screen. These libraries are **curses** and **Extended curses.**

curses is a set of screen control routines. This library is included for compatibility with existing application programs.

Minicurses is a subset of **curses** that does not allow you to manipulate more than one window. You invoke this subset by issuing –DMINICURSES as a CC option. This subset is smaller and faster than the full **curses** interface.

The full **curses** interface allows you to manipulate data structures. These data structures are called WINDOWS, and they can be thought of as two–dimensional arrays of characters representing all or part of the screen. A default window called **stdscr** is supplied, and you can create others using the **newwin** routine.

Windows are referred to by variables declared WINDOW *. The type WINDOW is defined in **curses.h** to be a a C structure. You manipulate these data structures with routines provided by **curses** and **extended curses**. The most basic routines are **move** and **addch**. The **refresh** routine makes the screen look like **stdscr** (standard screen).

Routine names beginning with W allow you to specify a window. Routine names not beginning with a W affect **stdscr**.

Using the Curses Subroutine Library

The **curses** subroutine package updates the screen with reasonable optimization. You use the **term.h** header file only if you are using the Terminfo Level Subroutines.

To initialize the routines which are described in the **curses** library, you must call the **initscr** routine before using any other routines which affect windows and screens. The routine **endwin** should be called before exiting. To get character–at–a–time input without echoing, call the **cbreak** and **noecho** routines. Most interactive screen–oriented programs require the character–at–a–time input without echoing.

If the environment variable **TERMINFO** is defined, any program using **curses** checks for a local terminal definition before checking in **/usr/lib/terminfo**. For example, **TERM** is set to vt100, then normally, the compiled file is found in **/usr/lib/terminfo/v/vt100**. (The directory name **v** is copied from the first letter of vt100 to avoid creating large directories.) If, for example, **TERMINFO** is set to **/usr/mark/myterms**, **curses** first checks **/usr/mark/myterms/v/vt100**. If this file does not exist, **curses** then checks **/usr/lib/terminfo/v/vt100**. This is useful for developing experimental definitions or when write permission in **/usr/lib/terminfo** is not available.

Note: The plotting library (plot), and the **curses** library (**curses**), both use the names erase and move. The **curses** versions are macros. If you need both libraries, put the plot code in a different source file than the **curses** code, or include the following statements in the plot code:

```
#undef move( )
#undef erase( )
```

Using Terminfo Level Subroutines

To use the terminfo level subroutines of **curses**, include the **curses.h** and **term.h** files, in that order, to get the definitions for these strings, numbers, and flags. Programs should call the **setupterm** subroutine before using any of the other **terminfo** subroutines. The **setupterm** subroutine defines the set of terminal–dependent variables defined in the **terminfo** file.

If your program needs only one terminal, you can specify the **–DSINGLE** flag to the C compiler. This results in static references instead of dynamic references to capabilities. The result is more concise code, but only one terminal can be used at a time for the program.

Capabilities with a Boolean value have the value 1 if the capability is present and 0 if it is not. Numeric capabilities have a value of –1 if the capability is missing and a value of 0 or greater if it is present. String capabilities have a NULL value if the capability is missing and otherwise have type char * and point to a character string that contains the capability. Special character codes that use the \ (backslash) and ^ (circumflex) characters are transformed into the appropriate ASCII characters. Padding information of the form $ <*time*>, and parameter information befinning with % (percent) are left uninterpreted. The **tputs** routine interprets padding information. The **tparm** routine interprets parameter information.

All **terminfo** strings (including the output of **tparm**) should be printed using **tputs** or **putp**. Before exiting, your program should call **reset_shell_mode** to restore the tty modes. Programs desiring shell escapes can call **reset_shell_mode** before the shell is called, and **reset_prog_mode** after returning from the shell.

Using the Extended Curses Subroutine Library

Extended Curses is an enhancement to the **curses** set of routines that provides extended funtion for:

- Expanded character set
- Color
- Multiple character attributes
- Error detection and handling
- Efficient handling of a window–oriented screen presentation, including:
  - Window stacking and layers
  - Linked scrolling of windows
  - Scrolling data in windows that are partially covered
  - Automatic tracking of active panes
- 2–byte characters for international character support
- **locator** input support

Use the preceding routines for new program development or to increase the function of existing programs.

The **extended curses** library contains a set of C language routines that provide the following enhancements:

- Updates a screen.
- Gets input from the terminal in a screen–oriented fashion.

- Moves the cursor from one point to another independent of other screen activities.
- Creates and manages a screen containing windows, panels and panes.
- A wider range of display attributes
- Generalized drawing of boxes
- Terminal–independent input data processing
- Extended window control
- Pane, panel, and field concepts
- Support for extended characters
- Handling of locator input.

The routines do the most common type of terminal–dependent funtions. The routines use the file **/usr/lib/terminfo** to describe what the terminal can do.

You can use motion optimization by itself. You can use screen updating and input without knowing about either motion optimization or the data.

The following information is provided to assist you in using the **Extended curses** Subroutine Library:

- Extended curses Subroutine Library terminology
- Using Terminfo Level Subroutines
- Using Screen Update Routines
- Changing Display Attributes
- Controlling Input with the keypad, extended, and trackloc Routines
- Scrolling Windows
- Improving Performance

Extended curses Subroutine Library Terminology

The following terms are used in the **Extended curses** routines:

window	The internal representation of what a portion of the display may look like at some point in time. Windows can be any size from the entire display screen to a single character.
screen	A window that is large as the display screen. A screen named **stdscr** is automatically provided.
terminal	Sometimes called a terminal screen. A special screen that is the Extended Curses package's understanding of what the work station's display screen currently looks like. The terminal screen is identified by a window named **curscr**, which should not be accessed directly by the user. Instead, changes should be made to **stdscr** (or a user–defined screen) and then **refresh** (or **wrefresh**) should be called to update the terminal.
presentation space	The array that contains the data and attributes associated with a window.
pane	An area of the display that shows all or part of the data contained in a presentation space associated with that pane.
active pane	The pane in which the text cursor is positioned. A pane must be active before you can enter information.
panel	A group of one or more panes that are treated as a unit. The panes of a panel are displayed together, erased together, and usually represent a unit

of information to a person using the application. A panel is represented on the display as a rectangular area that is tiled (completely filled) with panes.

field	An area in a presentation space into which the program accepts input.
extended character	A character other than 7–bit ASCII that can be represented in either 1 or 2 bytes. (See dstream..)
NLSCHAR	A data type that represents a character from code page P0, P1, or P2. It is defined to be equivalent to **unsigned short**. A single **NLSCHAR** variable can contain either a 1–byte or a 2–byte character. The 1–byte characters are stored in the low–order byte with the high–order byte set to 0; this is the same way that they are stored as integers. The 2–byte characters are stored with the single–shift control code in the high–order byte and the character data code in the low–order byte. This data type has no relation to the **NLchar** data type.

Using Screen Update Routines

The screen is a matrix of character positions that can contain any character from the character set that can be displayed on your terminal. Do not use control characters except when the descriptions of the library indicate that you can. The actual dimensions of the matrix are different for each type of terminal. These dimensions are defined when the **initscr** routine calls the **terminfo** initialization subroutine, **setupterm**. The routines enforce the following limits on the terminal:

Coordinate	**Description**
lines	If the teminal specification defines less than 5 lines, the routines use a value of 24 lines.
columns	If the terminal specification defines less than 5 columns, the routines use a value of 80 columns.

Line 0 is at the top of the screen. Line values in the routine syntax are represented by `line`. Column 0 is at the left side of the screen. Column values in the routine syntax are represented by `col`. When used in calls to the library routines, the `line` values come first.

```
move(line, col);
```

To update the screen, the routines must know what the screen currently looks like and what it should be changed to. The routines define the **WINDOW** data type to hold this information. This data type is a structure that describes a window image to the routines, including the starting position on the screen (the (`line`, `col`) coordinates of the upper left corner) and size.

You can think of a window as an array of characters on which to make changes. Using the window, a program builds and stores an image of a portion of the terminal that it later transfers to the actual screen. When the window is complete, use one of the following routines to transfer the window to the terminal:

refresh	Transfers the contents of **stdscr** to the terminal.
wrefresh	Transfers the contents of a named window (not **stdscr**) to the terminal.
ecrfpl	Transfers the contents of a named panel to the terminal.
ecrfpn	Transfers the contents of a named pane to the terminal.

This two–step process maintains several different copies of a window in memory and selects the proper one to display at any time. In addition, the program can change the contents of the screen in any order. When it has made all of the changes, the library routines update the terminal in an efficient manner.

Changing Display Attributes

Use the color and display characteristics defined in the following table for defining color and display attribute characters.section defining color and display attribute characteristics. The values of these variables depend on the capabilities of the current terminal and the priorities that you assign to the attributes. Change the values of these variables with the **sel_attr** routine. See the Screen Attributes Sample Program code fragment to see how to change the default set of attributes.

The section entitled Attribute Byte Order lists the order that the attribute bytes are packed.

Name	**Attribute**
UNDERSCORE	Display characters with underline.
REVERSE	Display characters in reverse video.
NORMAL	Display characters without highlighting (return to normal).
INVISIBLE	Do not display characters.
STANDOUT	Display characters in high intensity (can be used with other attribute colors). On many terminals, this is the same as **BOLD**.
BOLD	Display characters in bold font (or high intensity on some terminals).
BLINK	Display blinking characters (can be used with other attribute colors).
DIM	Display characters in reduced intensity.
PROTECTED	Protected display field.
F_BLACK	Set foreground color to black.
F_BLUE	Set foreground color to blue.
F_GREEN	Set foreground color to green.
F_CYAN	Set foreground color to cyan.
F_RED	Set foreground color to red.
F_MAGENTA	Set foreground color to magenta.
F_BROWN	Set foreground color to brown.
F_WHITE	Set foreground color to white.
B_BLACK	Set background color to black.
B_BLUE	Set background color to blue.
B_GREEN	Set background color to green.
B_CYAN	Set background color to cyan.
B_RED	Set background color to red.
B_MAGENTA	Set background color to magenta.
B_BROWN	Set background color to brown.
B_WHITE	Set background color to white.
FONT0	Select defined character font 0.
FONT1	Select defined character font 1.
FONT2	Select defined character font 2.
FONT3	Select defined character font 3.
FONT4	Select defined character font 4.
FONT5	Select defined character font 5.
FONT6	Select defined character font 6.
FONT7	Select defined character font 7.

Attribute Byte Order

The characteristics that a program selects for the terminal are loaded into the attribute byte associated with the data being displayed. Select as many of the attributes as needed, but those selected are packed into the attribute byte in the following order:

1. BOLD,
2. REVERSE
3. F_WHITE
4. F_RED
5. F_BLUE
6. F_GREEN
7. F_BROWN
8. F_MAGENTA
9. F_CYAN
10. F_BLACK
11. B_BLACK
12. B_RED
13. B_BLUE
14. B_GREEN
15. B_BROWN
16. B_MAGENTA
17. B_CYAN
18. B_WHITE
19. UNDERSCORE
20. BLINK
21. INVISIBLE
22. DIM
23. STANDOUT
24. PROTECTED
25. FONT0
26. FONT1
27. FONT2
28. FONT3
29. FONT4
30. FONT5
31. FONT6
32. FONT7
33. NULL

Controlling Input with the keypad, extended, and trackloc Routines

The **keypad** routine allows a program to recognize control sequences in the input without searching the input or introducing device dependencies. If **keypad** is active, it scans all input data for control sequences. If it finds a control sequence, it returns the associated code to the program instead of the actual control sequence. The control codes are shown in the following table. These codes are defined in the file **cur02.h** with values greater than 0x100.

To get international character support processing on input, **extended** must be active. If **extended** is turned off, shift codes and data codes input separately.

To get tracking of the locator cursor on the screen, **trackloc** must be active. If **trackloc** is turned off, the application has to handle the tracking of the locator cursor.

Name	**Description**
KEY_NOKEY	No keyboard data and no delay on
KEY_BREAK	Break
KEY_DOWN	Cursor down
KEY_UP	Cursor up
KEY_LEFT	Cursor left
KEY_RIGHT	Cursor right
KEY_HOME	Home – top left
KEY_BACKSPACE	Backspace

KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert character mode start
KEY_EIC	Exit insert character mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll forward
KEY_SR	Scroll backward (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab stop
KEY_CTAB	Clear tab stop
KEY_CATAB	Clear all tab stops
KEY_ENTER	Enter key
KEY_SRESET	Soft reset key
KEY_RESET	Hard reset key
KEY_PRINT	Print of copy
KEY_LL	Lower left (last line)
KEY_A1	Pad upper left
KEY_A3	Pad upper right
KEY_B2	Pad center
KEY_C1	Pad lower left
KEY_C3	Pad lower right
KEY_DO	DO key
KEY_QUIT	QUIT key
KEY_CMD	Command key
KEY_PCMD	Previous command key
KEY_NPN	Next pane key
KEY_PPN	Previous pane key
KEY_CPN	Command pane key
KEY_END	End key
KEY_HLP	Help key
KEY_SEL	Select key
KEY_SCR	Scroll right key
KEY_SCL	Scroll left key
KEY_TAB	Tab key
KEY_BTAB	Back tab key
KEY_NEWL	New–line key
KEY_F0	Function key – 128 values
KEY_F(n)	Not used
KEY_ESC1	Added to the ending character code for ESC sequences in the form ESC c with c in the range 0x30 – 0x7f. The value sent is in the range 0x200 to 0x24f.
KEY_ESC2	Added to the ending character code for ESC sequences in the form ESC [s c with c in the range 0x40 – 0x7f. The value sent is in the range 0x250 to 0x28f.

To use the control sequences in a program, first use a call to the **keypad** routine:

```
keypad(TRUE);
```

Scrolling Windows

If a window includes the lower right corner of the terminal screen, the flag byte bit, **_SCROLLWIN**, in the WINDOW structure for that window is set. This bit indicates that if a character is placed in the lower right corner of the window, the terminal inserts a blank line at the bottom of the screen (scrolls) to make room for more information. If a program defines the window with **scrollok** to allow it to scroll and place a character in the lower right corner of the window, the routines automatically call the **scroll** routine.

If a program does not define the window with **scrollok**, scrolling is not allowed. When a character is placed in the lower right corner of the window, the routines reset the current `col` coordinate to zero (beginning of line) and does not scroll.

To move a window to the lower right corner, use the **mvwin** routine. The **_SCROLLWIN** flag bit for that window is not automatically set. However, the **wrefresh** routine handles that window as if the **_SCROLLWIN** flag bit were set.

Improving Performance

To speed up output, create an output buffer using statements similar to the following program fragment:

```
#include <stdio.h>

char     obuf[BUFSIZE];

main( )
{
    setbuf (stdout, obuf);
          .
          .
          .
    /* rest of program */
}
```

How to Write to a Window

Procedure

1. Use the following functions to change the contents of a window:

   addch(c)
   addstr(str)
   box(win, vert, hor)
   cbox(win)
   chgat(num_chars, mode)
   clear
   clearok(scr, boolf)
   clrtobot
   clrtoeol
   colorend
   colorout(mode)
   delch
   deleteln
   ecactp
   ecshpl
   ecrfpl
   ecrfpn
   ecrmpl
   ecrmpn
   erase

fullbox(win, vert, hor, topl, topr, botl, botr)
insch(c)
insertln
move(line, col)
overlay(win1, win2)
overwrite(win1, win2)
printw(fmt, arg1, arg2, ...)
refresh
standend
standout
waddfld

2. Use the **refresh** routine to transfer the contents of the current window to the screen after all changes to the window are complete. The **refresh** routine does not rewrite any part of the window that has not changed since the last refresh call. To force the whole window to be rewritten, use the **touchwin** routine before the **refresh** routine. Also use **ecrfpn** to refresh a pane, and **ecrfpl** to refresh a panel.

How to Control the Screen

Procedure

1. Use the following library routines to control and manipulate the windows, panes, and panels on the screen:

delwin(win)
ecadpn
ecaspn
ecbpls
ecbpns
ecdfpl
ecdppn
ecdspl
ecdvpl
ecrlpl
endwin
gettmode
getyx(win, line, col) inch
initscr
leaveok(win, boolf)
longname
mvcur(lastline, lastcol, newline, newcol)
mvwin(win, line, col)
newview(orig_win, num_lines, num_cols)
newwin(lines, col, begin_line, begin_col)
nl
nonl
resetty
savetty
scroll(win)
scrollok(win, boolf)
setterm(name)
subwin(win, lines, cols, begin_line, begin_col)
touchwin(win)

trackloc
tstp
unctrl(ch) vscroll(view_win, deltaline, deltacol)
vscroll(view_win, deltaline, deltacol)

How to Define Panels and Panes

Procedure

1. To define a panel, provide the following information about the panel:
   - The size of the panel as it appears on the display.
   - The location on the display of the upper left corner of the panel.
   - Whether the panel is to have a border.
   - How the panel is to be divided into panes.
2. To define a pane within a panel, provide the following information:
   - The size of the presentation space associated with the pane.
   - The relative size of the pane within the panel.
   - Whether the pane is to have a border.
   - If and how the pane is to be further divided into smaller panes.
3. To divide panels and panes into smaller panes, follow a few simple rules. These rules ensure that a program can access all areas on the panel or pane that it creates:
   - You can divide a panel or pane either horizontally (using a horizontal dividing line) or vertically (using a vertical dividing line).
   - Panes created by a horizontal division must be linked together from top to bottom.
   - Panes created by a vertical division must be linked together from left to right.
   - Panes that are divided again must be linked to the first pane of its subpanes. The original pane in this case is not a part of the presented panel, but it is needed to define the structure of the panel.
   - Panes created by a horizontal division have a fixed horizontal dimension that is the same as its parent pane.
   - Panes created by a vertical division have a fixed vertical dimension that is the same as its parent pane.
   - Specify the variable dimension for a pane as being in one of three categories:

Fixed	For a *fixed* pane, specify the number of rows or columns, including any border, to assign to the pane.
Fractional	For a *fractional* pane, specify the percentage of the available space to assign to the pane.
Floating	For a *floating* pane, do not specify a size. The floating pane shares the available space equally with any other floating panes that the program creates.

   The linkage of the panes forms a tree structure. The root of the tree is a panel description. All other elements in the tree are pane descriptions.

How to Create Panels and Panes

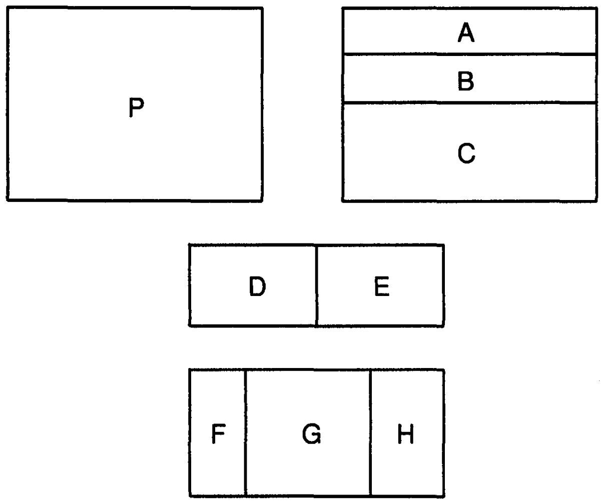
Procedure

To create a panel, perform the following steps:

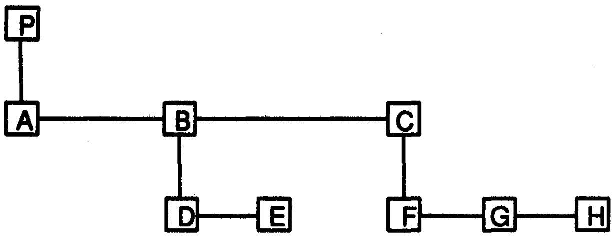
1. Define panel **P** using the **ecbpls** routine with a link to pane **A**.
2. Divide the panel (**P**) with two horizontal splits into three panes. Use the **ecbpns** routine to define the three panes with the following links:
   a. **A** – No links
   b. **B** – Linked to **A** and **D**
   c. **C** – Linked to **B** and **F**
3. Divide pane **B** with a single vertical split into two panes. Use the **ecbpns** routine to define the two panes with the following links:
   a. **D** – No links
   b. **E** – Linked to **D**
4. Divide pane **C** with two vertical splits into three panes. Use the **ecbpns** routine to define the three panes with the following links:
   a. **F** – No links
   b. **G** – Linked to **F**
   c. **H** – Linked to **G**

The following diagrams show the final panel appearance after the preceding steps have been completed. Links in the Panel and Pane Structure, and Creating Panes in the Panel. Horizontal lines show the links within a pane; vertical lines show links to the parent panel or pane.

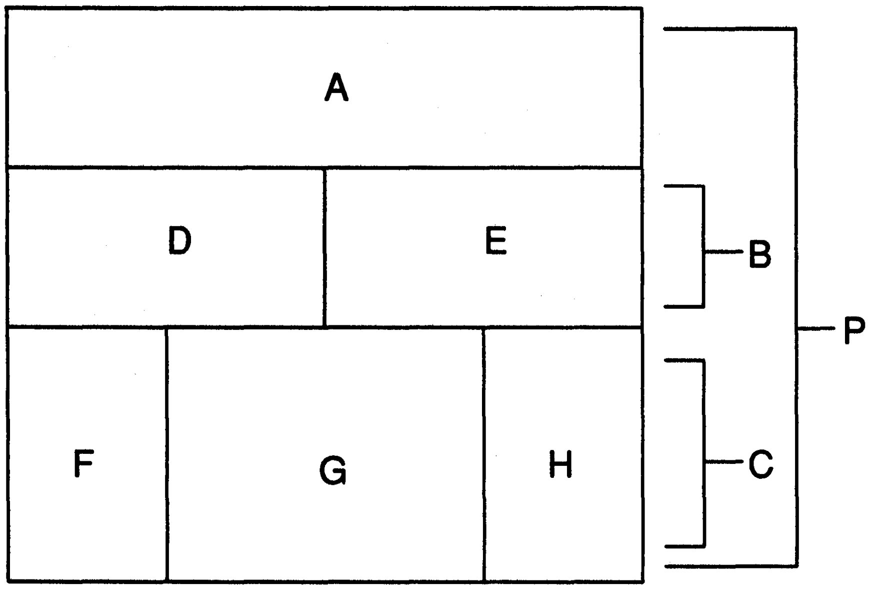
Creating Panes in the Panel



Links in the Panel and Pane Structure



Example Panel Final Appearance



The following is a Sample Program from which the Final Panel was created:

```
#include  <cur01.h>
#include  <cur05.h>

main()

{

        pane   *A, *B, *C, *D, *E, *F, *G, *H ;
        panel *P ;

initscr () ;

A = ecbpns (24,80,NULL,NULL,0,2500,Pdivszp,Pbordry NULL,NULL);

D = ecbpns (24,80,NULL,NULL,0,0,    Pdivszf,Pbordry,NULL,NULL);
E = ecbpns (24,80,D,    NULL,0,0,   Pdivszf,Pbordry,NULL,NULL);

B = ecbpns (24,80,A,D, Pdivtyh,3000, Pdivszf,Pbordry,NULL,NULL);

F = ecbpns (24,80,NULL,NULL,0,0,    Pdivszf,Pbordry,NULL,NULL);
G = ecbpns (24,80,F,    NULL,0,5000,Pdivszf,Pbordry,NULL,NULL);
H = ecbpns (24,80,G,    NULL,0,3000,Pdivszf,Pbordry,NULL,NULL);

C = ecbpns (24,80,B, F,Pdivtyh, 0, Pdivszf,Pbordry,NULL,NULL);

P = ecbpls (24,80,0, 0,NULL, Pdivtyv, 0, Pdivszf,Pbordry,A );

ecdvpl ( P );
ecdfpl ( P, FALSE );
ecshpl ( P );
ecrfpl ( P );

endwin();
}        /* end main program */
```

How to Use the Extended Curses Library

Procedure

1. To use the library, define the types and variables that the routines use. The file **cur01.h** contains all of the definitions that are needed for the library routines for most common uses. Include this file in the program by putting the following statement at the top of the program source:

   ```
   #include <cur01.h>
   ```

2. To use the library routines for panel and pane management (those routine names begin with the letters **ec**), use the following statement in the program source to include a larger set of definitions:

   ```
   #include <cur05.h>
   ```

3. To use the library routines for programs that use global variables defined to represent the information taken from the terminal file, include the following statement in the program source:

   ```
   #include <cur00.h>
   ```

The **cur00.h** header file also contains include statements for the following header files: **stdio.h**, **sgtty.h**, and **cur01.h**, but you do not have to specify these header files separately in the program.

4. To compile a program with the **cc** command, specify the two additional libraries on the command line:

   ```
   cc myproc.c -lcur
   ```

   The above example compiles the program `myproc.c`, with the linked output going to **a.out**.

How to Set Up the Extended Curses Environment

Procedure

To use the library routines, the program must set up the operating conditions for the program. Perform the following actions in the program, if they apply, in the order that they appear:

1. Perform all necessary actions to load the program and make sure that it is operating successfully.
2. To change the defined size of the terminal, set the variable `LINES` and `COLS` to new values.
3. Use the routine **initscr** to get information about terminal characteristics, and to allocate memory for **stdscr** and **curscr**. Call **initscr** before calling any routines that affect windows. If the program uses a window routine before **initscr**, the program will not run.
4. Check the value that **initscr** returns to see if the screen setup was successful. If this value is 0 (FALSE or ERR), then **initscr** could not get enough memory for the needed windows.
5. Use any needed terminal status changing routine, such as **nl** or **crmode**.
6. Create any new windows with the **newwin** or **subwin** routine.
7. Create panels using **ecbpls**, **ecbpns**, or **ecdfpl**.
8. Define or change the characteristics of the windows as needed. For example, the routine **scrollok** allows the window to scroll, or the routine **leaveok** leaves the cursor at the position of the last change.

The program can now work with the windows that it has defined. When the program is done, use the routine **endwin** to clean up before exiting the program. This routine restores terminal modes to what they were when the program first started.

Curses Programming Example

The following example program, **twinkle.c**, uses the extended curses library routines (**libcur.a**) to create a series of displays on the screen.

```
#include        <cur00.h>
#include <signal.h>

#define NCOLS 80
#define NLINES 24
#define MAXPATTERNS 11
```

```
struct locs
{
   char  y, x;
};

typedef struct locs  LOCS;

LOCS  layout[ NCOLS * NLINES ];  /* current board layout */

int   pattern,                   /* current pattern number */
      numstars;                  /* numbers of stars in ptern */

main()
{
   char *getenv();
   int die();

   srand( getpid() );            /* initialize random sequence */
   initscr();
   signal( SIGINT, die );
   noecho();
   leaveok( stdscr, TRUE );
   scrollok( stdscr, FALSE );

   for( ;; )
   {
      makeboard();               /* make the board setup */
      puton( '*' );              /* put on '*'s */
      system( "sleep 2" );
      erase();
      refresh();
   }
}

/*
**  On program exit, move the cursor to the lower left corner by
**  direct addressing, since current location is not certain.
**  We say we used to be at the upper right corner to obtain
**  absolute addressing.
*/

die()
{
   signal( SIGINT, SIG_IGN );
   mvcur( LINES/2, COLS/2, 0, 0 );
   wclear( curscr );
   wrefresh( curscr );
   endwin();
   exit(0);
}

/*
**  Make the current board setup.  It picks a random pattern and
**  calls ison() to determine if the character is on that pattern
**  or not.
*/

makeboard()
{
   reg int  y, x;
   reg LOCS  *lp;

   pattern = rand() % MAXPATTERNS;
   lp = layout;
   for( y = 0; y < NLINES; y++ )
   {
      for( x = 0; x < NCOLS; x++ )
      {
         if( ison( y, x ))
         {
            lp -> y = y;
            lp++ -> x = x;
         }
      }
   }
   numstars = lp - layout;
}

/*
**  Return TRUE if( y, x ) is on the current pattern.
*/

ison( y, x )

reg int y, x;

{
   switch( pattern )
   {
      /*
      ** Alternating lines:
      */
      case 0:
         return !( y & 01 );
      /*
      ** Box:
      */
      case 1:
         if( y < 3 || y >= NLINES - 3 )
            return TRUE;
         return( x < 4 || x >= NCOLS - 4 );
      /*
      ** Cross:
      */
      case 2:
         return( ( x + y ) & 01 );
      /*
      ** Bar across center:
      */
      case 3:
         return( y >= 9 && y <= 15 );
      /*
      ** Alternating columns:
      */
      case 4:
         return !( x & 02 );
      /*
      ** Bar down center:
        */
        case 5:
           return( x >= 36 && x <= 44 );
        /*
        ** Bar across and down center:
        */
        case 6:
          return( ( y >= 9 && y <= 15 ) || ( x >= 37 && x <= 43 ));
        /*
        ** Bar across and down center, in a box:
        */
        case 7:
           if( y < 3 || y >= NLINES - 3 )
              return TRUE;
           if( x < 4 || x >= NCOLS - 4 )
              return TRUE;
           return( ( y >= 10 && y <= 14 )||( x >= 36 && x <= 44 ));
        /*
        ** Asterisk:
        */
        case 8:
           if( abs( x - y ) <= 2 || abs( NLINES - ( x + y )) <= 2 )
              return TRUE;
           if( abs( ( NLINES/2 ) - x ) <= 2 )
              return TRUE;
           return( abs( ( NLINES/2 ) - y ) <= 1 && x <= NLINES );
        /*
        **  Ellipse:
        */
        case 9:
           return
           (
              (
                (( float ) (( x-40 ) * ( x-40 )) ) / 1521 +
                (( float ) (( y-12 ) * ( y-12 )) ) / 121
              ) <= 1
           );
        /*
        ** Circle:
        */
        case 10:
           return
              (
                 (
                   (( float ) (( x-28 ) * ( x-28 )) ) / 729 +
                   (( float ) (( y-12 ) * ( y-12 )) ) / 121
                 ) <= 1
              );
   }     /* end of switch( pattern ) */
}        /* not reached */

puton(ch)
reg char ch;
{
   reg LOCS  *lp;
   reg LOCS  *end;
   LOCS      temp;
   reg  int     r;
   end = &layout[ numstars ];
   for( lp = layout; lp < end; lp++ )
   {
      r = rand() % numstars;
      temp = *lp;
      *lp = layout[ r ];
      layout[ r ] = temp;
   }

   for( lp = layout; lp < end; lp++ )
   {
      mvaddch( lp -> y, lp -> x, ch );
      refresh();
   }
}  /* end of twinkle */
```

Screen Attributes Sample Program

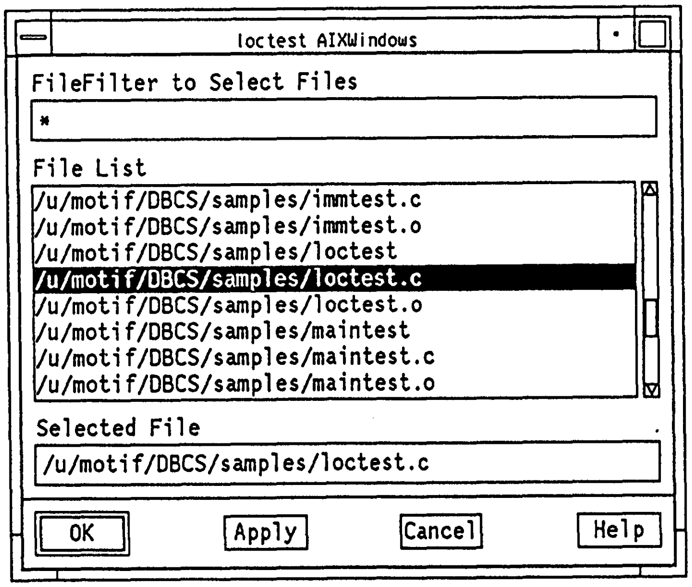
The following is a sample code fragment showing how to use the display constants to change the default set of attributes:

```
#include <cur00.h>
#include <cur03.h>

int     attrs[] =
{
        _dBOLD, _dBLINK,
        _dF_WHITE, _dF_RED, _dF_BLUE, _dF_GREEN,
        _dF_BROWN, _dF_MAGENTA, _dF_CYAN, _dF_BLACK,
        _dB_BLACK, _dB_RED, _dB_BLUE, _dB_GREEN,
        _dB_BROWN, _dB_MAGENTA, _dB_CYAN, _dB_WHITE,
        _dREVERSE, _dINVISIBLE, _dDIM, _dUNDERSCORE,
        NULL
};

main( )
{

sel_attr(attrs);
initscr( );
if( REVERSE == NORMAL ) REVERSE = F_BLACK | B_WHITE;
if( INVISIBLE == NORMAL ) INVISIBLE = F_BLACK | B_BLACK;
if( DIM == NORMAL ) DIM = F_BLACK | BOLD;
if( UNDERSCORE == NORMAL ) UNDERSCORE = F_WHITE | B_RED;
STANDOUT = REVERSE;

          <rest of program>

endwin( );
}  /* end main */
```

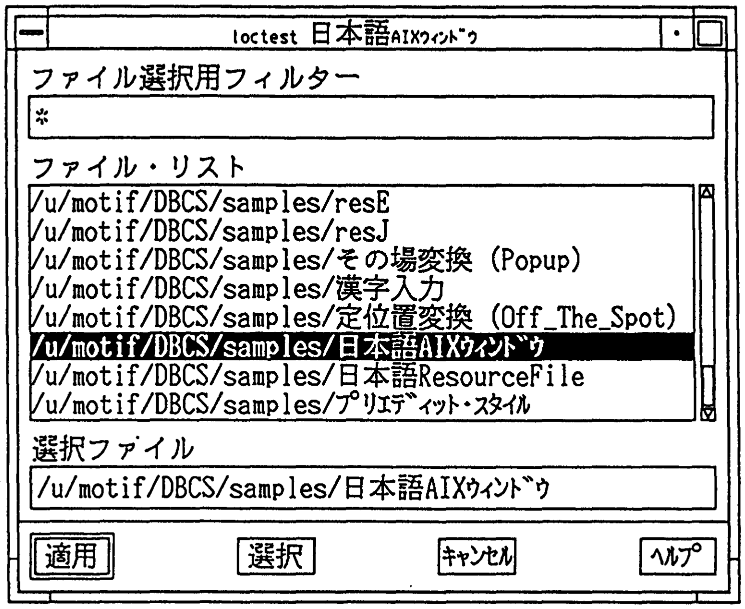
The routines only define 8 bits of unique attribute information. Selecting foreground color, backrgound color or font requires either 1, 2 or 3 bits depending upon the number of colors or fonts in the list. 1 bit for 2 or fewer, 2 bits for 3 or 4, and 3 bits for 5 to 8. Each character attribute takes 1 bit. However, the attribute names passed to **wcolorout** are variables, so that you can make combinations from the other attributes as shown in the last part of the preceding sample program. If a requested attribute (that is not the terminal default) is equal to NORMAL, then it is either not supported by the terminal, or there is not enough space in the window structure for its mask.

Appendix A: DBCS Sample Source Code

A Locale–Sensitive Application: loctest.c



```
loctest AIXWindows
FileFilter to Select Files
*
File List
/u/motif/DBCS/samples/immtest.c
/u/motif/DBCS/samples/immtest.o
/u/motif/DBCS/samples/loctest
/u/motif/DBCS/samples/loctest.c
/u/motif/DBCS/samples/loctest.o
/u/motif/DBCS/samples/maintest
/u/motif/DBCS/samples/maintest.c
/u/motif/DBCS/samples/maintest.o
Selected File
/u/motif/DBCS/samples/loctest.c
OK    Apply    Cancel    Help
```



```
loctest 日本語AIXｳｨﾝﾄﾞｳ
ファイル選択用フィルター
*
ファイル・リスト
/u/motif/DBCS/samples/resE
/u/motif/DBCS/samples/resJ
/u/motif/DBCS/samples/その場変換 (Popup)
/u/motif/DBCS/samples/漢字入力
/u/motif/DBCS/samples/定位置変換 (Off_The_Spot)
/u/motif/DBCS/samples/日本語AIXｳｨﾝﾄﾞｳ
/u/motif/DBCS/samples/日本語ResourceFile
/u/motif/DBCS/samples/ﾌﾟﾘｴﾃﾞｨｯﾄ・ｽﾀｲﾙ
選択ファイル
/u/motif/DBCS/samples/日本語AIXｳｨﾝﾄﾞｳ
適用    選択    ｷｬﾝｾﾙ    ﾍﾙﾌﾟ
```

A Locale-sensitive Application: **loctest.c**

Notes:

1. This locale–sensitive application can output the following languages without recompilation:
   - English

- Japanese

2. It can also dynamically select an appropriate resource file corresponding to its run–time environment as specified by the environment variable **LANG**.

   - English

   LANG is set to `En_US.pc850`, indicating one of the AIX Input Methods.

   $HOME/En_US.pc850/loctest provides language dependent data for this application.

   - Japanese

   LANG is set to `Jp_JP.pc932`, indicating one of the AIX Input Methods.

   $HOME/Jp_JP.pc932/loctest provides language dependent data for this application.

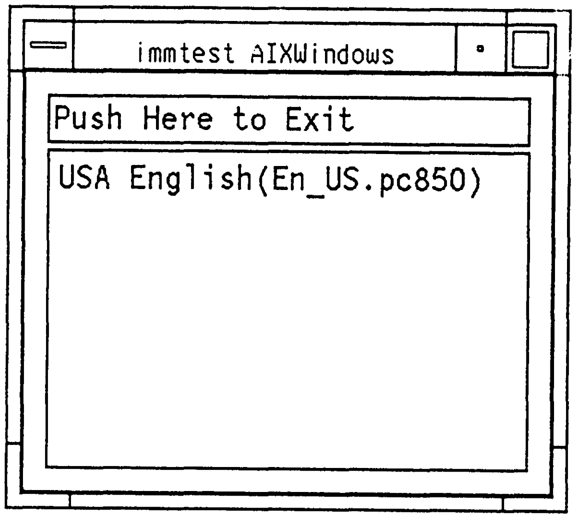
Sample Source Code: loctest.c

```
/*
* Sample Source Code – loctest.c
*
* Locale–sensitive application
*
* goJ.loc or goE.loc as script files to set environment.
* resJ.loc or resE.loc as resource file for NLS
*
*/

/************************/
/* include files       */
/************************/
#include <locale.h>
#include <stdio.h>

#include <X11/Xatom.h>
#include <X11/Intrinsic.h>
#include <X11/Shell.h>

#include <Xm/Xm.h>
#include <Xm/FileSB.h>
#include <Xm/mbRes.h>

/*************/
/* constants */
/*************/
#define BOX_APPLY 0
#define BOX_CANCEL 1
#define BOX_HELP 2
#define BOX_OK 3

/**************************/
/* global   declarations */
/**************************/
void box_CB();
void bye();

/****************/
/* main program */
```

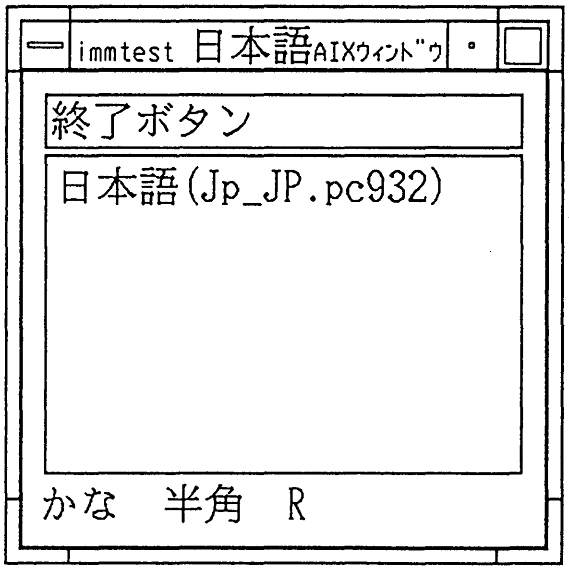
```
/****************/
int main(argc,argv)
int argc;
char **argv;
{
  /*******************/
  /* local variables */
  /*******************/
  Widget toplevel,box;
  char *locale,*language;
  Arg arg[10];
  int i;

  /*************/
  /* get LANG  */
  /*************/
  language = (char *)getenv("LANG");
  printf("LANG = %s.\n",language);

  /**************/
  /* get locale */
  /**************/
  locale = (char *)setlocale(LC_CTYPE,"");
  printf("locale = %s.\n",locale);

  /**********************/
  /* initialize toolkit */
  /**********************/
  toplevel = XtInitialize(argv[0], "loctest", NULL,
                          NULL, &argc, argv);

  /***************************/
  /* create FileSelectionBox */
  /***************************/
  i = 0;
  box = XmCreateFileSelectionBox(toplevel,
                                 "File Selection Dialog",arg,i);


  XtAddCallback(box, XmNokCallback,     box_CB, BOX_OK);
  XtAddCallback(box, XmNcancelCallback, box_CB, BOX_CANCEL);
  XtAddCallback(box, XmNapplyCallback,  box_CB, BOX_APPLY);
  XtAddCallback(box, XmNhelpCallback,   box_CB, BOX_HELP);
  XtManageChild(box);

  /******************/
  /* realize window */
  /******************/
  XtRealizeWidget(toplevel);
  XtMainLoop();
}

/************/
/* Callback */
/************/
void bye()
{
  exit(0);
}
```

```
/*************/
/* Callback */
/*************/
void box_CB(w,client_data,call_data)
Widget w;
caddr_t client_data;
caddr_t call_data;
{
  switch((int)client_data)
  {
    case BOX_CANCEL:
      printf("CANCEL Callback is called.\n");
      break;
    case BOX_APPLY:
      printf("APPLY Callback is called.\n");
      break;
    case BOX_HELP:
      printf("HELP Callback is called.\n");
      break;
    case BOX_OK:
      printf("OK Callback is called.\n");
      break;
    default:
      printf("UNKNOWN Callback is called.\n");
      break;
  }
}
```

A Program that Creates an XmInputMethod Widget Interface Directly: immtest.c



USA English(En_US.pc850)



日本語(Jp_JP.pc932)

かな　半角　R

An Application with an XmInputMethod Widget: **immtest.c**

Notes:

1. This locale–sensitive application can output the following languages without recompilation:
   - English
   - Japanese
2. It can also dynamically select an AIX Input Method and a resource file corresponding to its run–time environment as specified by the environment variables **LANG** and **XENVIRONMENT.**
   - English

LANG is set to `En_US.pc850`, indicating one of the AIX Input Methods.

XENVIRONMENT is set to `resE`, which is the user–defined file name. No **statusWindow** structure or **preEditWindow** structure is created by the English Input method `En_US.pc850`.

- Japanese

LANG is set to `Jp_JP.pc932`, indicating one of the AIX Input Methods.

XENVIRONMENT is set to `resJ`, which is the user–defined file name. A **statusWindow** structure and a **preEditWindow** structure are created by the Japanese Input Method `Jp_JP.pc932`.

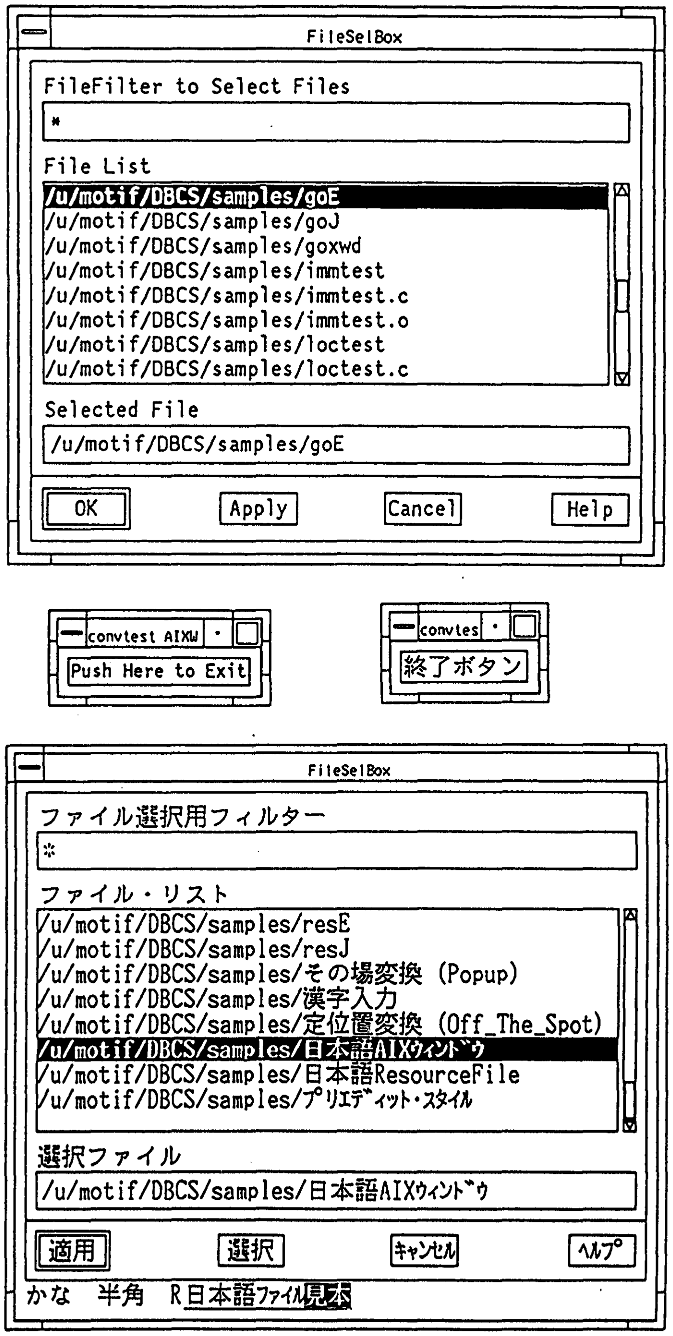
Sample Source Code: immtest.c

```
/* sample source code – immtest.c
*
*
* Direct creation of InputMethod widget by
* XtCreateManagedChild()
*
* goJ.imm or goE as script file to set up environment
* resJ.imm or resE as resource file
*
*/

/*****************/
/* include files */
/*****************/
#include <stdio.h>
#include <X11/StringDefs.h>
#include <X11/Intrinsic.h>
#include <X11/keysym.h>
#include <Xm/Label.h>
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>
#include <Xm/Text.h>
#include <Xm/IMM.h>
#include <locale.h>

/*************************/
/* global  declarations */
/*************************/
void process_IM();
void bye();

/********************/
/* global variables */
/********************/
Widget  imw;

/****************/
/* main program */
/****************/
int main(argc, argv)
int argc;
char **argv;
{
  /*****************/
```

```
/* local variables */
/*******************/
Widget toplevel, box;
Widget textwidget;
Widget rc;
Widget exitbutton;
Widget immwidget;
Arg args[10];
int n;
char *locale;

/**************/
/* set locale */
/**************/
locale = (char *)setlocale(LC_CTYPE, "");
fprintf(stderr,"locale=%s\n",locale);

/**********************/
/* initialize toolkit */
/**********************/
toplevel = XtInitialize(argv[0], "Immtest",
                        NULL, NULL, &argc, argv);
n = 0 ;
XtSetArg (args[n], XmNallowShellResize, True);  n++;
XtSetArg (args[n], XmNallowOverlap,    False);  n++;
XtSetValues (toplevel, args, n);

/*XSynchronize(XtDisplay(toplevel), True); */

/***********************/
/* create input method */
/***********************/

immwidget = XtCreateManagedWidget("mmwidget",
                                 xmInputMethodWidgetClass,
                                 toplevel, NULL, 0);

/******************************/
/* create widget as work area */
/******************************/
n = 0;
XtSetArg (args[n], XmNallowOverlap, False); n++;
rc = XtCreateManagedWidget("rc",
                           xmRowColumnWidgetClass,
                           immwidget, args, n);

n = 0;
exitbutton = XtCreateManagedWidget("Exit Button",
                                   xmPushButtonWidgetClass,
                                   rc, args, n);
XtAddCallback(exitbutton, XmNactivateCallback, bye, NULL);

n = 0;
XtSetArg(args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
XtSetArg(args[n], XmNwidth, 300);  n++;
XtSetArg(args[n], XmNheight, 200); n++;
textwidget = XtCreateManagedWidget("textwidget",
                                   xmTextWidgetClass,
                                   rc, args, n);
```

```
  /****************************************/
  /* tell work area to input method widget */
  /****************************************/
  n = 0;
  XtSetArg(args[n], XmNworkWindow, rc); n++;
  XtSetValues(immwidget, args, n);

  XtRealizeWidget(toplevel);
  XtMainLoop();
}

/******************************/
/* callback for button widget */
/******************************/
void bye()
{
  fprintf(stderr,"Exit callback is called.\n");
  exit(1);
}
```

A Program that Uses the XmCreateFileSelectionDialog Subroutine to Create an XmInputMethod Widget Indirectly: convtest.c



```
FileSelBox
FileFilter to Select Files
*
File List
/u/motif/DBCS/samples/goE
/u/motif/DBCS/samples/goJ
/u/motif/DBCS/samples/goxwd
/u/motif/DBCS/samples/immtest
/u/motif/DBCS/samples/immtest.c
/u/motif/DBCS/samples/immtest.o
/u/motif/DBCS/samples/loctest
/u/motif/DBCS/samples/loctest.c
Selected File
/u/motif/DBCS/samples/goE
OK    Apply    Cancel    Help
```

```
convtest AIXW
Push Here to Exit
```

```
convtes
終了ボタン
```

```
FileSelBox
ファイル選択用フィルター
*
ファイル・リスト
/u/motif/DBCS/samples/resE
/u/motif/DBCS/samples/resJ
/u/motif/DBCS/samples/その場変換 (Popup)
/u/motif/DBCS/samples/漢字入力
/u/motif/DBCS/samples/定位置変換 (Off_The_Spot)
/u/motif/DBCS/samples/日本語AIXｳｨﾝﾄﾞｳ
/u/motif/DBCS/samples/日本語ResourceFile
/u/motif/DBCS/samples/ﾌﾟﾘｴﾃﾞｨｯﾄ・ｽﾀｲﾙ
選択ファイル
/u/motif/DBCS/samples/日本語AIXｳｨﾝﾄﾞｳ
適用    選択    ｷｬﾝｾﾙ    ﾍﾙﾌﾟ
かな  半角  R日本語ﾌｧｲﾙ見本
```

An Application with an XmInputMethod Widget: **convtest.c**

Notes:

1. This locale–sensitive application can output the following languages without recompilation:
   - English
   - Japanese

2. It can also dynamically select an AIX Input Method and a resource file corresponding to its run–time environment as specified by the environment variables **LANG** and **XENVIRONMENT**.

   - English

   LANG is set to `En_US.pc850`, indicating one of the AIX Input Methods.

   XENVIRONMENT is set to `resE`, which is the user–defined file name. No **statusWindow** structure or **preEditWindow** structure is created by the English Input method `En_US.pc850`.

   - Japanese

   LANG is set to `Jp_JP.pc932`, indicating one of the AIX Input Methods.

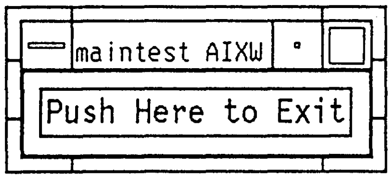
   XENVIRONMENT is set to `resJ`, which is the user–defined file name. A **statusWindow** structure and a **preEditWindow** structure are created by the Japanese Input Method `Jp_JP.pc932`.

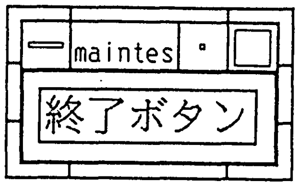
Sample Source Code: convtest.c

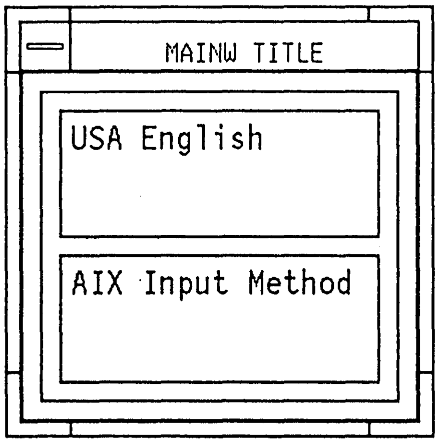
```
/* sample source code – convtest.c
*
*
* Implicit creation of InputMethod widget by
* Convenience Dialog XmCreateFileSelectionDialog()
*
* goJ.loc or goE.loc as script file to set environment
* resJ.loc or resE.loc as resource file
*
*/
/*****************/
/* include files */
/*****************/
#include <stdio.h>
#include <X11/StringDefs.h>
#include <X11/Intrinsic.h>
#include <X11/keysym.h>
#include <Xm/Label.h>
#include <Xm/PushB.h>
#include <Xm/BulletinB.h>
#include <Xm/FileSB.h>
#include <Xm/SelectioB.h>
#include <Xm/DialogS.h>
#include <Xm/XmP.h>
#include <Xm/IMM.h>
#include <locale.h>

/*************/
/* constants */
/*************/
#define BOX_APPLY  0
#define BOX_CANCEL 1
#define BOX_HELP   2
#define BOX_OK     3

/*************************/
/* global  declarations */
/*************************/
void bye();
void box_CB();

/*********************/
/* global variables */
/*********************/
Widget toplevel;

/****************/
/* main program */
/****************/
int main(argc, argv)
int argc;
char **argv;
{
  /*******************/
  /* local variables */
  /*******************/
  Widget topshell;
  Widget exit_button;
  Widget box;
  Arg args[10];
  int n;


  /****************************************/
  /* Make this application locale-senstive */
  /****************************************/
  setlocale(LC_CTYPE, "");

  /*********************/
  /* Initialize window */
  /*********************/
  topshell = XtInitialize(argv[0], "convtest",
                          NULL, NULL, &argc, argv);
  n = 0;
  XtSetArg (args[n], XmNallowShellResize, True); n++;
  XtSetValues (topshell, args, n);

  /***************************************/
  /* Create BulletinBoard as work window */
  /***************************************/
  n = 0;
  XtSetArg (args[n], XmNwidth,  300);  n++;
  XtSetArg (args[n], XmNheight, 380);  n++;
  XtSetArg (args[n], XmNx, 200);  n++;
  XtSetArg (args[n], XmNy, 200);  n++;
  toplevel = XtCreateManagedWidget("work area",
                                   xmBulletinBoardWidgetClass,
                                   topshell, args, n);

  /*****************************/
  /* Create PushButton for Exit */
  /*****************************/
  n = 0;
  XtSetArg (args[n], XmNx, 10); n++;
  XtSetArg (args[n], XmNy, 10); n++;
  exit_button = XtCreateManagedWidget ("Exit Button",
                                       xmPushButtonWidgetClass,
                                       toplevel, args, n);
  XtAddCallback (exit_button, XmNactivateCallback, bye, NULL);

  /***************************************/
  /* Create FileSelectionDialog by using */
  /* XmCreateFileSelectionDialog(). This */
  /* function creates an InputMethod     */
  /* widget implicitly.                  */
  /***************************************/
  n = 0;
  XtSetArg (args[n], XmNx, 20); n++;
  XtSetArg (args[n], XmNy, 20); n++;
  XtSetArg (args[n], XmNpreEditType, XmPREEDIT_UNDER);  n++;

  XtSetArg (args[n], XmNinputMethodWidget,
                     XmDEFAULT_INPUT_METHOD);  n++

  XtSetArg (args[n], XmNdialogTitle,
            XmStringCreate("FILESEL TITLE",
                           XmSTRING_DEFAULT_CHARSET) ); n++;
  box = XmCreateFileSelectionDialog (toplevel,
  "FILESEL WIDGET",
  args, n);
  XtAddCallback(box, XmNokCallback, box_CB, BOX_OK);
  XtAddCallback(box, XmNcancelCallback, box_CB, BOX_CANCEL);
  XtAddCallback(box, XmNapplyCallback, box_CB, BOX_APPLY);
  XtAddCallback(box, XmNhelpCallback, box_CB, BOX_HELP);

  XtManageChild(box);
  XtRealizeWidget(topshell);
  XtMainLoop();
}

/****************************************/
/* callback for FileSelectionBox widget */
/****************************************/
void box_CB(w,client_data,call_data)
Widget w;
caddr_t client_data;
caddr_t call_data;
{
  switch((int)client_data)
  {
    case BOX_CANCEL:
      printf("CANCEL Callback is called.\n");
      break;
    case BOX_APPLY:
      printf("APPLY Callback is called.\n");
      break;
    case BOX_HELP:
      printf("HELP Callback is called.\n");
      break;
    case BOX_OK:
      printf("OK Callback is called.\n");
      break;
    default:
      printf("UNKNOWN Callback is called.\n");
      break;
  }
}
```

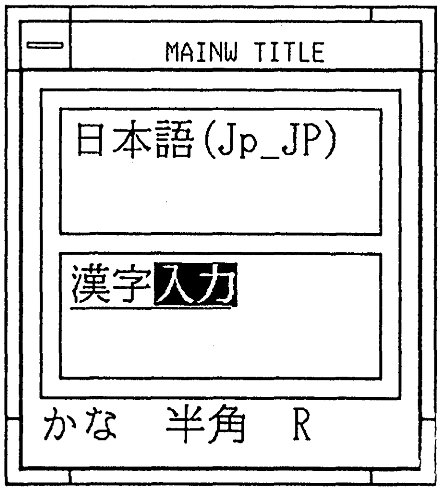
```
/******************************/
/* callback for button widget */
/******************************/
void bye()
{
  printf("Exit Callback is called.\n");
  exit(1);
}
```

A Program that Uses the XmCreateMainWindow Subroutine to Create an XmInputMethod Widget Indirectly: maintest.c









An Application with an XmInputMethod Widget: **maintest.c**

Notes:

1. This locale–sensitive application can output the following languages without recompilation:
   - English
   - Japanese
2. It can also dynamically select an AIX Input Method and a resource file corresponding to its run–time environment as specified by the environment variables **LANG** and **XENVIRONMENT**.
   - English

   LANG is set to `En_US.pc850`, indicating one of the AIX Input Methods.

XENVIRONMENT is set to `resE`, which is the user–defined file name. No **statusWindow** structure or **preEditWindow** structure is created by the English Input method `En_US.pc850`.

- Japanese

LANG is set to `Jp_JP.pc932`, indicating one of the AIX Input Methods.

XENVIRONMENT is set to `resJ`, which is the user–defined file name. A **statusWindow** structure and a **preEditWindow** structure are created by the Japanese Input Method `Jp_JP.pc932`.

Sample Source Code: maintest.c

```
/* sample source code – maintest.c
*
*
* Implicit creation of InputMethod widget by
* Convenient Dialog XmCreateMainWindow()
*
* goJ.imm or goE as Script file to set environment
* resJ.imm or resE as resource file
*
*/

/*****************/
/* include files */
/*****************/
#include <stdio.h>
#include <X11/StringDefs.h>
#include <X11/Intrinsic.h>
#include <X11/keysym.h>
#include <Xm/Label.h>
#include <Xm/PushB.h>
#include <Xm/Text.h>
#include <Xm/BulletinB.h>
#include <Xm/MainW.h>
#include <Xm/DialogS.h>
#include <Xm/IMM.h>
#include <locale.h>

/***********************/
/* global declarations */
/***********************/
void bye();


/********************/
/* global variables */
/********************/
Widget toplevel;

/****************/
/* main program */
/****************/
void main(argc, argv)
unsigned int argc;
char  **argv;
{
  /******************/
```

```
/* local variables */
/*******************/
Widget topshell;
Widget shell;
Widget exit_button;
Widget work_area1;
Widget main_window;
Widget text_widget1;
Widget text_widget2;
Arg args[10];
int n;
static char kanji1[] = {0x8A, 0xBF, 0x8E,0x9A,0x00};
static char kanji2[] = {0x95, 0x5C, 0x8E,0xA6,0x00};

/*****************************************/
/* Make this application locale—sensitive */
/*****************************************/
setlocale(LC_CTYPE, "");

/*********************/
/* Initialize window */
/*********************/
topshell = XtInitialize(argv[0], "maintest",
                        NULL, NULL, &argc, argv);
n = 0;
XtSetArg (args[n], XmNallowShellResize, True); n++;
XtSetArg (args[n], XmNallowOverlap, False); n++;
XtSetValues (topshell, args, n);

/****************************************/
/* Create BulletinBoard as a work window */
/****************************************/
n = 0;
toplevel = XtCreateManagedWidget("work area 1",
                                 xmBulletinBoardWidgetClass,
                                 topshell, args, n);

/******************************/
/* Create PushButton for Exit */
/******************************/
n = 0;
exit_button = XtCreateManagedWidget("Exit Button",
                                    xmPushButtonWidgetClass,
                                    toplevel, args, n);
XtAddCallback(exit_button, XmNactivateCallback, bye, NULL);

/****************************/
/* Create application shell */
/****************************/
n = 0;
XtSetArg (args[n], XmNallowShellResize, True); n++;
shell = XtCreatePopupShell ("MAINW WIDGET",

xmDialogShellWidgetClass,
xmTransientShellWidgetClass,

toplevel, args, n);

/**********************************************/
/* Create MainWindow using XmCreateMainWindow(). */
/* This function creates the InputMethod widget   */
/* implicitly.                                    */
/************************************************/
n = 0;

XtSetArg (args[n], XmNinputMethodWidget,
 XmDEFAULT_INPUT_METHOD); n++

XtSetArg (args[n], XmNborderWidth, 2); n++;
main_window = XmCreateMainWindow(shell,
                               "MAINW WIDGET", args, n);
XtManageChild(main_window);

/*************************************/
/* Create BulletinBoard as work area */
/*************************************/
n=0;
work_area1 = XmCreateBulletinBoard(main_window,
                                  "work area 2", args, n);

/**********************************************/
/* Add BulletinBoard as work area of main window*/
/**********************************************/
n = 0;
XtSetArg (args[n], XmNworkWindow, work_area1); n++;
XtSetValues (main_window, args, n);
XtManageChild (work_area1);

/***************************************/
/* Create a text child of the work area */
/***************************************/
n = 0;
XtSetArg (args[n], XmNwidth, 200); n++;
XtSetArg (args[n], XmNheight, 80); n++;
text_widget1 = XmCreateText (work_area1, "text #1", args, n);

/*****************************************/
/* Create 2nd text child of the work area */
/*****************************************/
n = 0;
XtSetArg (args[n], XmNy, 100); n++;
XtSetArg (args[n], XmNwidth, 200); n++;
XtSetArg (args[n], XmNheight, 80); n++;
XtSetArg (args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
text_widget2 = XmCreateText (work_area1, "text #2", args, n);

XtManageChild (text_widget1);
XtManageChild (text_widget2);

/****************************************************/
/* set title on parent of mainw (imw) to MAINW TITLE */
/****************************************************/
n = 0;
XtSetArg (args[n], XmNdialogTitle,
          XmStringCreate("MAINW TITLE",
                         XmSTRING_DEFAULT_CHARSET) );  n++;
XtSetValues(XtParent(main_window), args, n);

/**********************/
/* Realize top shell */
```

```
  /***********************/
  XtRealizeWidget(topshell);
  XtMainLoop();
}

/******************************/
/* callback for button widget */
/******************************/
void bye()
{
  printf("Exit Callback is called.\n");
  exit(1);
}
```

Suggested Reading

Related Information

Step–by–step procedures relating to these samples are discussed in How to Support a National Language in a libXm Application.

Appendix B: AIXwindows User Interface Language (UIL) Compiler Diagnostic Messages

The following diagnostics messages are produced by the UIL Compiler:

Note: Messages are listed alphabetically by IDENT code. The following strings are used to represent data that varies from message to message:

String	**Data Represented**
%c	Character
%d	Decimal number
%s	String

IDENT Code: arg_count

procedure %s was previously declared with %d parameters

Severity: Error

The declaration of the marked procedure specified a different number of parameters than are present in this procedure reference.

User Action

Check that you are calling the correct subroutine. If you intend to call the procedure with a varying number of parameters, omit the argument list in the procedure declaration.

IDENT Code: arg_type

found %s value – procedure %s parameter must be %s value

Severity: Error

The declaration of the marked procedure specified a different type of parameter than is present in this procedure reference.

User Action

Check that you are passing the correct parameter to the correct subroutine. If you intend to call the procedure with varying parameter types, declare the procedure specifying any for the type of the parameter.

IDENT Code: backslash_ignored

unknown escape sequence "\%c" – ignored

Severity: Error

A backslash was followed by an unknown escape character. The backslash (\) is the escape character in UIL. A selected set of single characters can follow a backslash such as \n for newline or \\ to insert a backslash. The character following the backslash was not one of the selected set.

User Action

If you want to add a backslash, use \\..

IDENT Code: bug_check

internal error: %s

Severity: Severe

The Compiler diagnosed an internal error.

User Action

Submit a software problem report.

IDENT Code: circular_def

widget %s is part of a circular definition

Severity: Error

The indicated widget is referenced as a descendant of itself, either within its own definition or within the definition of one of the widgets in the widget tree that it controls.

User Action

Change the definition of the indicated widget so that it is not a descendant of itself.

IDENT Code: control_char

unprintable character \%d\ ignored

Severity: Error

The Compiler encountered a illegal control character in the UIL specification file. The decimal value of the character is given between the backslash (\) characters

User Action

Replace the character with the sequence specified in the message (for example, \3 if the internal value of the control character is 3). UIL provides several built–in control characters such as \n and \r for newline and carriage return.

IDENT Code: create_proc

creation procedure is not supported by the %s widget

Severity: Error

You specified a creation procedure for a AIXwindows Toolkit widget. You can specify a creation procedure only for a user–defined widget.

User Action

Remove the procedure clause following the widget type.

IDENT Code: create_proc_inv

creation procedure is not allowed in a %s widget reference

Severity: Error

You specified a creation procedure when referencing an widget. You can specify a creation procedure only when you declare the widget.

User Action

Remove the procedure clause following the widget type.

IDENT Code: create_proc_req

creation procedure is required in a %s widget declaration

Severity: Error

When defining a user–defined widget, you must specify the name of the creation subroutine for creating an instance of this widget.

User Action

Insert a procedure clause following the widget type in the widget declaration. You also need to declare the creation procedure using a procedure declaration. For example:

```
procedure my_list_box_creation_proc();
            widget list_box:
             user_defined procedure my_list_box_creation_proc()
             { arguments ... };
```

IDENT Code: ctx_req

context requires a %s – %s was specified

Severity: Error

At the point marked in the specification, one type of object (such as a widget) is required and your specification supplied a different type of object (such as value).

User Action

Check for misspelling or that you have referred to the intended widget.

IDENT Code: d_add_source

additional UIL source file: %s was ignored

Severity: Error

More than one source file was specified. Only the first source file will be compiled.

User Action

Compile additional source files using separate invocations of the Compiler.

IDENT Code: d_dupl_opt

duplicate option \ "%s" \ was ignored

Severity: Warning

The same command line option has been repeated more than once (for example, the **–o** option or the **–v** option)

User Action

Remove duplicate command line option.

IDENT Code: d_miss_opt_arg

%s missing following \ "%s" \ option

Severity: Error

You used a command line option that requires a parameter and you did not provide that parameter.

User Action

Omit the option or provide the parameter.

IDENT Code: d_no_source

no source file specified

Severity: Severe

No source file was specified to compile.

User Action

Specify the name of a UIL specification file to compile.

IDENT Code: d_unknown_opt

unknown option \ “%s” \ was ignored

Severity: Warning

An unknown option has been used in the Compiler command line.

User Action

Check what you typed on the command line.

IDENT Code: dup_letter

Color letter used for prior color in this table

Severity: Error

Each of the letters used to represent a color in a color table must be unique. If not, that letter in a icon would represent more than one color (each pixel can have only one color associated with it at a time). The letter marked has been assigned to more than one color.

User Action

Choose which color the letter is to represent and remove or assign a new character to any duplicates.

IDENT Code: dup_list

%s %s already specified for this %s %s

Severity: Error

A widget or gadget declaration can have at most one arguments list, one callbacks list, and one controls list.

User Action

If you want to specify multiple lists of arguments, controls, and callbacks, you can do so within one list. For example:

```
arguments { arguments_list1; arguments_list2; };
```

IDENT Code: gadget_not_sup

%s gadget is not supported – %s widget will be used instead

Severity: Warning

The indicated widget type does not support a gadget variant; only a widget variant is supported for this widget type. The UIL Compiler ignores the gadget indication, and creates widgets of this widget type.

User Action

Specify that this widget type is a widget instead of a gadget.

IDENT Code: icon_letter

row %d, column %d: letter \"%c"\ not in color table

Severity: Error

You have specified a color to be used in an icon that is not in the color table for that icon. The invalid color is identified in the message by displaying the letter used to represent that color between the backslashes (\). This letter was not defined in the specified color table.

User Action

Either add the color to the color table for that icon or use a character representing a color in the color table. The default color table defines " " as background and "*" as foreground.

IDENT Code: icon_width

row %d must have same width as row 1

Severity: Error

The icons supported by UIL are rectangular (that is, x pixels wide by y pixels high). As a result, each of the strings used to represent a row of pixels in an icon must have the same length. The specified row does not have the same length as the first row.

User Action

Make all the strings in the icon subroutine the same length.

IDENT Code: inv_Module

invalid Module structure – check UIL Module syntax

Severity: Error

The structure of the UIL Module is incorrect.

User Action

If there are any syntax errors reported, fix them and recompile. For example, if the error occurs before the first widget declaration (that is, before your value and widget declarations), check the syntax of the Module header for unwanted semicolons (;) after the Module clauses. If the error occurs at the end of the Module, check that the Module concludes with the keywords "end module;".

IDENT Code: list_item

%s item not allowed in %s %s

Severity: Error

The indicated list item is not of the type required by the list. Arguments lists must contain argument entries, callbacks lists must contain callback entries, controls lists must contain control entries, and procedures lists must contain callback entries.

User Action

Check the syntax for the type of list entry that is required in this context and change the indicated list item.

IDENT Code: listing_open

error opening listing file: %s

Severity: Severe

The Compiler could not create the listing file noted in the message.

User Action

Check that you have write access to the directory you specified to hold the listing file.

IDENT Code: listing_write

error writing to listing file: %s

Severity: Severe

The Compiler could not write a line into the listing file noted in the message.

User Action

Check to see that there is adequate space on the disk specified to hold the listing file.

IDENT Code: name_too_long

name exceeds 31 characters – truncated to: %s

Severity: Error

The UIL Compiler encountered a name longer than 31 characters. The Compiler truncated the name to the leftmost 31 characters.

User Action

Shorten the name in the UIL Module source.

IDENT Code: names

place names clause before other Module clauses

Severity: Error

The case sensitivity clause, if specified, must be the first clause following the name of the Module. You have inserted another Module clause before this clause.

User Action

Reorder the Module clauses so that the case sensitivity clause is first. (It is acceptable to place the version clause ahead of the case sensitivity clause; this is the only exception.)

IDENT Code: never_def

%s %s was never defined

Severity: Error

Certain UIL widgets such as gadgets and widgets can be referred to before they are defined. The marked widget is such an widget, however, the Compiler never found the widget declaration.

User Action

Check for misspelling. If the Module is case sensitive, the spellings of names in declarations and in references must match exactly.

IDENT Code: no_uid

no UID file was produced

Severity: Informational

If the Compiler reported error or severe diagnostics (that is, any of the diagnostic abbreviations starting with %UIL–E or %UIL–F), a UID file is not created. This diagnostic informs you that the Compiler did not produce a UID file.

User Action

Fix the problems reported by the Compiler.

IDENT Code: non_pvt

value used in this context must be private

Severity: Error

A private value is one that is not imported or exported. In the context marked by the message, only a private value is legal. Situations where this message is issued include: defining one value in terms of another; and parameters to subroutines. In general, a value must be private when the Compiler must know the value at compilation time. Exported values are disallowed in these contexts, even though a value is present, because that value could be overridden at run time.

User Action

Change the value to be private.

IDENT Code: not_impl

%s is not implemented yet

Severity: Error

You are using a feature of UIL that has not been implemented.

User Action

Try an alternate technique.

IDENT Code: null

a NULL character in a string is not supported

Severity: Warning

You have created a string that has an embedded null character. Strings are represented in a UID file and in many AIXwindows Toolkit data structures as null terminated strings. So, although the embedded nulls will be placed in the UID file, AIXwindows Toolkit subroutines may interpret an imbedded null as the terminator for the string.

User Action

Be very careful using embedded nulls.

IDENT Code: obj_type

found %s %s when expecting %s %s

Severity: Error

Most parameters take values of a specific type. The value specified is not correct for this parameter.

User Action

The message indicates the expected type of parameter. Check that you have specified the intended value and that you specified the correct parameter.

IDENT Code: operand_type

%s type is not valid for %s

Severity: Error

The indicated operand is not of a type that is supported by this operator.

User Action

Check the definition of the operator and make sure the type of the operand you specify is supported by the operator.

IDENT Code: out_of_memory

Compiler ran out of virtual memory

Severity: Severe

The Compiler ran out of virtual memory.

User Action

Reduce the size of your application.

IDENT Code: out_range

value of %s is out of range %s

Severity: Error

The value specified is outside the legal range of its type.

User Action

Change the UIL Module source.

IDENT Code: prev_error

compilation terminated – fix previous errors

Severity: Severe

Errors encountered during the compilation have caused the Compiler to abort.

User Action

Fix the errors already diagnosed by the Compiler and recompile.

IDENT Code: previous_def

name %s previously defined as %s

Severity: Error

The name marked by the message was used in a previous declaration. UIL requires that the names of all widgets declared within a Module be unique.

User Action

Check for a misspelling. If the Module is case sensitive, the spellings of names in declarations and in references must match exactly.

IDENT Code: single_letter

color letter string must be a single character

Severity: Error

The string associated with each color in a color table must hold exactly one character. You have specified a string with either fewer or more characters.

User Action

Use a single character to represent each color in a color table.

IDENT Code: single_occur

%s %s supports only a single %s %s

Severity: Warning

You have specified a particular clause more than once in a context where that clause can only occur once. For example, the version clause in the Module can only occur once.

User Action

Choose the correct clause and delete the others.

IDENT Code: src_limit

too many source files open: %s

Severity: Severe

The Compiler has a fixed limit for the number of source and include files that it can process. This number is reported in the message.

User Action

Use fewer include files.

IDENT Code: src_null_char

source line contains a null character

Severity: Error

The specified source line contains a null character. The Compiler ignores any text following the null character.

User Action

Replace each null character with the escape sequence "\".

IDENT Code: src_open

error opening source file: %s

Severity: Severe

The Compiler could not open the UIL specification file listed in the message.

User Action

Check that the file listed in the message is the one you want to compile, that it exists, and that you have read access to the file. If you are using a large number of include files, you may have exceeded your quota for open files.

IDENT Code: src_read

error reading next line of source file: %s

Severity: Severe

The Compiler could not read a line of the UIL specification file listed in the message.

User Action

In the listing file, this message should appear following the last line the Compiler read successfully. First check that the file you are compiling is a UIL specification file. If it is, the file mostly likely contains corrupted records.

IDENT Code: src_truncate

line truncated at %d characters

Severity: Error

The Compiler encountered a source line greater than 132 characters. Characters beyond the 132 character limit were ignored.

User Action

Break each source line longer than 132 characters into several source lines. Long string literals can be created using the concatenation operator.

IDENT Code: submit_spr

internal error – submit an SPR

Severity: Severe

The Compiler diagnosed an internal error.

User Action

Get a listing and look where the error is being issued. Try fixing any faulty syntax in this area. If you are unable to prevent this error, submit a software problem report.

IDENT Code: summary

errors: %d warnings: %d informationals: %d

Severity: Informational

This message lists a summary of the diagnostics issued by the Compiler, and appears only when diagnostics have been issued.

User Action

Fix the problems reported. You can use the –I option qualifier to suppress informational and warning diagnostics that you have determined to be harmless.

IDENT Code: supersede

this %s %s supersedes a previous definition in this %s %s

Severity: Informational

A parameter or callback list has either a duplicate parameter or duplicate reason.

User Action

This is not necessarily an error. The Compiler is alerting you to make sure that you intended to override the value of a prior parameter. This informational message can be suppressed using the –I option.

IDENT Code: syntax

unexpected %s token seen – parsing resumes after \"%c"\

Severity: Error

At the point marked in the Module, the Compiler found a construct such as a punctuation mark, name, or keyword when it was expecting a different construct. The Compiler continued analyzing the Module at the next occurrence of the construct stated in the message.

User Action

Check the syntax of your UIL Module at the point marked by the Compiler. If the Module specifies case–sensitive names, check that your keywords are in lowercase characters.

IDENT Code: too_many

too many %ss in %s, limit is %d

Severity: Error

You exceeded a Compiler limit such as the number of fonts in a font table or the number of strings in a translation table. The message indicates the limit imposed by the Compiler.

User Action

Restructure your UIL Module.

IDENT Code: uid_open

error opening UID file: %s

Severity: Severe

The Compiler could not create the UID file noted in the message. A UID file holds the compiled user interface specification.

User Action

Check that you have write access to the directory you specified to hold the UID file. If you have a large number of source and include files, check that you have not exceeded your open file quota.

IDENT Code: undefined

%s %s must be defined before this reference

Severity: Error

The widget pointed to in the message was either never defined or not defined prior to this point in the Module. The Compiler requires the widget to be defined before you refer to the widget.

User Action

Check for a misspelling of the name of the widget, a missing declaration for the widget, or declaring the widget after its first reference. If names in the Module are case sensitive, the spellings of the name in the declaration and in the reference must match exactly.

IDENT Code: unknown_charset

unknown character set

Severity: Error

The message is pointing to a context where a character set name is required. You have not specified the name of a character set in that context.

User Action

Check for misspelling.

If you specified case–sensitive names in the Module, check that the character set name is in lowercase characters.

IDENT Code: unknown_seq

unknown sequence \"%s"\ ignored

Severity: Error

The Compiler detected a sequence of printable characters it did not understand. The Compiler omitted the sequence of characters listed between the quotation marks (" ").

User Action

Fix the UIL Module source.

IDENT Code: unsupported

the %s %s is not supported for the %s %s

Severity: Warning

Each widget or gadget supports a specific set of parameters, reasons, and children. The particular parameter, reason, or child you specified is not supported for this widget or gadget.

User Action

If a widget creation subroutine accepts a parameter that UIL rejects, this does not necessarily indicate that the UIL Compiler is in error. Widget creation subroutines ignore parameters that they do not support, without notifying you that the parameter is being ignored.

IDENT Code: unterm_seq

%s not terminated %s

Severity: Error

The Compiler detected a sequence that was not properly terminated, such as a string literal without the closing quotation mark.

User Action

Insert the proper termination characters.

IDENT Code: wrong_type

found %s value when expecting %s value

Severity: Error

The indicated value is not of the specific type required by UIL in this context.

User Action

Check the definition of the subroutine or clause.

Index

Reader's Comment Form

AIX User Interface Programming Concepts for RISC System/6000

SC23–2209–01

Please use this form only to identify publication errors or to request changes in publications. Your comments assist us in improving our publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your IBM-approved remarketer. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box. Be sure to print your name and address below.

Page	Comments

Please contact your IBM representative or your IBM–approved remarketer to request additional publications.

Please print

Date	________
Your Name	________________________
Company Name	________________________
Mailing Address	________________________
	________________________
	________________________
Phone No.	( ) ________________ Area Code

No postage necessary if mailed in the U.S.A

Tape Please Do Not Staple Tape

Fold Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Internal Zip 3603
11400 Burnet Rd.
Austin, Texas 78758–3493

Fold Fold

SC23-2209-01